Deep learning and reinforcement learning

Jesús Fernández-Villaverde\textsuperscript{1} and Galo Nuño\textsuperscript{2}
June 27, 2021

\textsuperscript{1}University of Pennsylvania

\textsuperscript{2}Banco de España
1. Dynamic programming in continuous time.


3. Heterogeneous agent models.

4. Optimal policies with heterogeneous agents.
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.

• 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, ...
An artificial neural network (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:

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

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

• Compare:

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

with a standard projection:

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

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 multilayer composition of $J > 1$ neural networks:

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

where the $M^{(1)}, M^{(2)}, \ldots$ and $\phi^1(\cdot), \phi^2(\cdot), \ldots$ 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.
• 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 is trained by regression to predict the expected outcome (that is, whether the current player wins) in positions from the self-play data set. The network is then improved by policy gradient learning to maximize the outcome (that is, winning more games) against previous versions of the game. The policy network, from 30 million positions from the KGS Go Server, was used to maximize the likelihood of the human move, $\sigma_s$, selected in state $s$, using a linear probability map over the board. The value network similarly uses many convolutional filters per layer, $\rho_s$, to predict the expected outcome from the perspective of the current player, $\rho_s(s)$. We use a reward function, $r$, that is identical in structure to the SL policy network, $p_\pi$, but outputs a scalar value, represented by a scalar value, $v_\theta(s')$. The RL policy network was trained on KGS, that executes 100,000 simulations per move. Using no search, the RL policy network won 85% of games against Pachi. In comparison, the previous state-of-the-art, based only on supervised learning, won 44.4% at the time of submission. 

We evaluated the performance of the RL policy network against the match version of AlphaGo. The policy network takes a representation of the board position, $s$, to perform a fast rollout to select a move, $a$. The value network, $v_\theta$, maximizes the outcome (that is, winning more games) against previous versions of the game. The value network is trained by regression to predict the expected outcome, $\rho_s(s)$, from the perspective of the current player, $\rho_s(s)$. We use a reward function, $r$, that is identical in structure to the SL policy network, $p_\pi$, but outputs a scalar value, represented by a scalar value, $v_\theta(s')$. The RL policy network was trained on KGS, that executes 100,000 simulations per move. Using no search, the RL policy network won 85% of games against Pachi. In comparison, the previous state-of-the-art, based only on supervised learning, won 44.4% at the time of submission. 

We play random rollouts, the fast rollout policy $p_\sigma$, to select a move, $a$, and a randomly selected move, $a'$. Positions and outcomes were sampled from human expert games. Each position was evaluated by a single forward pass of the value network, $\rho$. The policy network is initialized to the SL policy network, $p_\pi$. The policy network, $p_\rho$, is trained by stochastic gradient ascent to maximize the outcome (that is, winning more games) against previous versions of the game. The policy network is trained on KGS, that executes 100,000 simulations per move. Using no search, the RL policy network won 85% of games against Pachi. In comparison, the previous state-of-the-art, based only on supervised learning, won 44.4% at the time of submission.
A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play.

David Silver1,2, Julian Schrittwieser1, Ioannis Antonoglou1, THorsten Graepel1, Timothy Lillicrap1, Karen Simonyan1, Demis Hassabis1

*These authors contributed equally to this work.

Corresponding author. Email: davidsilver@google.com (D.S.);
dhcontact@google.com (D.H.)
on December 9, 2018 http://science.sciencemag.org/ Downloaded from

A long-standing ambition of artificial intelligence was to enable computers to play challenging games at superhuman levels. In 1997, Deep Blue defeated the world chess champion, Garry Kasparov. More recently, AlphaGo Zero has achieved superhuman performance in the game of Go, by using the same algorithm, a reinforcement learning algorithm, to play games of self-play; these are then used to guide its search. In this paper, we introduce AlphaZero, a more generic version of the AlphaGo algorithm succeeding in multiple domains—chess, shogi, as well as Go, by using the same algorithm and move-ordering heuristics, AlphaZero uses a general-purpose deep neural network (DNN). However, these systems are highly tuned to their domain and cannot be generalized to other games without substantial human effort, whereas general game-playing programs that play substantially human effort, whereas general game-playing programs that play

In terms of game tree complexity, shogi is a substantially harder game than chess (6,400 legal moves versus 30 legal moves). Shogi programs attempt this complexity using handcrafted features and carefully tuned domain-specific adaptations. In (10), these augmentations, focusing on the 2016 Top Chess Engine Championship (TCEC) season 9 world champion Stockfish, have only recently defeated human world chess champions (11, 12). Computer shogi programs, including Deep Blue, use very similar architectures (7). However, these systems are highly tuned to their domain and cannot be generalized to other games without substantial human effort. Other strong chess programs, including Deep Blue, use very similar architectures (7). However, these systems are highly tuned to their domain and cannot be generalized to other games without substantial human effort. Other strong chess programs (13, 1140) remain comparatively weak.

Computer science and mathematical science are highly tuned to their domain and can-they-be generalized to other games without substantial human effort. Other strong chess programs (13, 1140) remain comparatively weak.

For each action $a$ at a position $s$, AlphaZero computes the expected outcome

$$q(s,a)=P_r(a|s)\sum_{s'}P(s'|s,a)v(s')$$

of the game from $s$ and $a$, where $P_r(a|s)$ is the move probability, $\sum_{s'}P(s'|s,a)v(s')$ is the expected outcome of the game from $s$ and $a$, and $v(s')$ is the evaluation of the position in the outcome $s'$. This neural network takes $s$ as an input and outputs a vector of $C$ components $\{p_j\}_{j=1}^C$, the values of the action $a_j$.

A landmark for artificial intelligence was von Neumann’s tabula rasa algorithm, which derived the minimax algorithm. This algorithm succeeds in multiple domains—chess, shogi, as well as Go, by using the same algorithm and move-ordering heuristics, AlphaZero uses a general-purpose deep neural network (DNN). However, these systems are highly tuned to their domain and cannot be generalized to other games without substantial human effort, whereas general game-playing programs that play substantially human effort, whereas general game-playing programs that play

- **Chess**
  - AlphaZero
  - Stockfish

- **Shogi**
  - AlphaZero
  - Elmo

- **Go**
  - AlphaZero
  - AlphaGo Zero
  - AlphaGo Lee
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.
- 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 bullets.

- 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 an ANN will beat an ARIMA(n,p,q) (at least only with macro variables).
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: \( x_1, x_2, \ldots, x_N \). 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  Weights

\[ x_1 \rightarrow \theta_1 \]

\[ x_2 \rightarrow \theta_2 \]

\[ x_3 \rightarrow \theta_3 \]

\[ \vdots \]

\[ x_n \rightarrow \theta_n \]

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

Net input

Activation

Perceptron classification output

\[ \gamma \]
The biological analog
Activation functions

• Traditionally:
  1. A sigmoidal function:
     \[ \phi(z) = \frac{1}{1 + e^{-z}} \]
  3. Hyperbolic tangent.

• Other activation functions have gained popularity recently:
  1. Rectified linear unit (ReLU):
     \[ \phi(v) = \max(0, z) \]
  2. Softplus:
     \[ \phi(v) = \log(1 + e^z) \]
• $\theta_0$ controls the activation threshold.

• The level of the $\theta$ control the activation rate (the higher the $\theta$, the harder the activation).

• Some textbooks separate the activation threshold and a scaling parameter 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 $\theta$ do 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.
Combining neurons into a neural network

- As before, we have $N$ observables: $x_1, x_2, \ldots, x_N$.
- Coefficients (or weights): $\theta_{0,m}$ (a constant), $\theta_{1,m}, \theta_{2,m}, \ldots, \theta_{N,m}$.
- 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:
  \[
  y \equiv g(x; \theta) = \theta_0 + \sum_{m=1}^{M} \theta_{m} \phi (z_m)
  \]
- Also, quasi-linear structure in terms of vectors of observables and coefficients.
- This is known as a single layer network.

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.
\[ n_1(x) = \text{Relu}(-5x - 7.7) \]
\[ n_2(x) = \text{Relu}(-1.2x - 1.3) \]
\[ n_3(x) = \text{Relu}(1.2x + 1) \]
\[ n_4(x) = \text{Relu}(1.2x - .2) \]
\[ n_5(x) = \text{Relu}(2x - 1.1) \]
\[ n_6(x) = \text{Relu}(5x - 5) \]

\[ Z(x) = -n_1(x) - n_2(x) - n_3(x) + n_4(x) + n_5(x) + n_6(x) \]
• 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).

• What about Chebyshev polynomials? Splines? Problems of convergence and extrapolation.

• There is another, yet more subtle curse of dimensionality.
Training the network

- $\theta$ 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?
Minimization

- Minibatch gradient descent (a variation of stochastic gradient descent) is the most popular algorithm.

- Why stochastic? Intuition from Monte Carlos.

- Why minibatch? Intuition from GMM. Notice also resilience to scaling.

- In practice, we do not need a global min (≠ 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.

Figure 2-7. The stochastic error surface fluctuates with respect to the batch error surface, enabling saddle point avoidance.
• Random multi-trial with initialization from a proposal distribution \( \Theta \) (typically a Gaussian or uniform):

\[ \{\theta\}^0 \sim \Theta \]

• \( \theta \) is recursively updated:

\[ \{\theta\}^{i+1} = \{\theta\}^i - \epsilon^i \nabla E (\theta; y_j, \hat{y}_j) \]

where:

\[ \nabla E (\theta; y_j, \hat{y}_j) \equiv \left[ \frac{\partial E (\theta; y_j, \hat{y}_j)}{\partial \theta_0}, \frac{\partial E (\theta; y_j, \hat{y}_j)}{\partial \theta_1}, \ldots, \frac{\partial E (\theta; y_j, \hat{y}_j)}{\partial \theta_{N,M}} \right]^\top \]

is the gradient of the error function with respect to \( \theta \) evaluated at \((y_j, \hat{y}_j)\) until:

\[ \| \{\theta\}^{i+1} - \{\theta\}^i \| < \varepsilon \]

• In a minibatch, you use a few observations instead of just one.
Some details

- We select the learning rate $\epsilon_m > 0$ using some optimality criterium.

- We evaluate the gradient using \textit{back-propagation} (Rumelhart \textit{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.

- This will be particularly important below when we introduce multiple layers.
1. Newton and Quasi-Newton are unlikely to be of much use in practice. Why? However, perhaps your problem is an exception.


3. Genetic algorithms:
   - In fact, much of the research in deep learning incorporates some flavor of genetic selection.
   - Basic idea.
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

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

... 

\[
    y \approx g(x; \theta) = \theta_0^K + \sum_{m=1}^{M} \theta_{m}^K \phi (z_m^{K-1})
\]
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:

\[ y_m = \frac{e^{z_{m-1}}}{\sum_{m=1}^M e^{z_{m-1}}} \]
Application to Economics
• Joint work with George Sorg-Langhans and Maximilian Vogler.

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

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

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.
Error criterion 1

- The HJB error:

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

- The policy function error:

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

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 \( \alpha \).
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:

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

and the complementary slackness error:

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

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

\[
\mathcal{E}(x; \Theta) \equiv \|\text{err}_{HJB}(x; \Theta)\|^2_2 + \|\text{err}_{\alpha}(x; \Theta)\|^2_2 + \|\text{err}_{PF_1}(x; \Theta)\|^2_2 + \|\text{err}_{PF_2}(x; \Theta)\|^2_2 + \|\text{err}_{DF}(x; \Theta)\|^2_2 + \|\text{err}_{CS}(x; \Theta)\|^2_2
\]
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 $\eta_k$ being the learning rate in the $k$-th epoch.
An Example
The continuous-time neoclassical growth model

- 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:

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

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

- We set \( \gamma = 2, \rho = 0.04, F(k) = 0.5 \times k^{0.36}, \delta = 0.05 \).
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. $\text{tanh}(x)$ activation.
4. Normal initialization $\mathcal{N} \left( 0, 4 \sqrt{\frac{2}{n_{\text{input}} + n_{\text{output}}}} \right)$ with input normalization.
Tanh Function

\[ a = \frac{e^z - e^{-z}}{e^z + e^{-z}} \]
(a) Value with closed-form policy
(c) Consumption with closed-form policy
(e) HJB error with closed-form policy
- 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:

$$err_C = \left(U'\right)^{-1} \left(\frac{\partial \tilde{V}(k; \Theta^V)}{\partial k}\right) - \tilde{C}(k; \Theta^C)$$
(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.
CHAPTER 9. CONVOLUTIONAL NETWORKS

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?
Reinforcement Learning

An Introduction
second edition

Richard S. Sutton and Andrew G. Barto
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*. 
A policy-based method I

- Proposed by Thathachar and Sastry (1985).

- A very simple method that uses the averages $Q_n(a)$ of rewards $R_i(a)$, $i = \{1, \ldots, n\}$, 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:

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

  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 "\( \varepsilon \)-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 action-value estimates converge to the true values. This of course implies that the probability of selecting the optimal action converges to greater than 1/10, 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 \( \varepsilon \)-greedy action selection, for the case of two actions and \( \varepsilon = 0.5 \), what is the probability that the greedy action is selected?

The 10-armed Testbed To roughly assess the relative effectiveness of the greedy and \( \varepsilon \)-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, \( q^*(a), a = 1, \ldots, 10 \), were selected according to a normal distribution with mean zero and unit variance, and then the actual rewards were selected according to a mean \( q^*(a) \) unit variance normal distribution, as suggested by these gray distributions.
The 10-armed testbed was 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.
Chapter 2: Multi-armed Bandits

2.6 Optimistic Initial Values

All the methods we have discussed so far are dependent to some extent on the initial action-value estimates, $Q(a)$. 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 $\beta$, 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 $q^*(a)$ 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 $Q(a) = +5$, for all $a$. For comparison, also shown is an $\varepsilon$-greedy method with $Q(a) = 0$. 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 100%.
where \( \ln t \) denotes the natural logarithm of \( t \) (the number that \( e^{\pi} \approx 2.71828 \) 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.
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 softmax 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 the average reward over first 1000 steps
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 \text{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 \text{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.
Appendix A
Direction set methods

- Suppose a function $f$ is roughly approximated as a quadratic form:
  \[
  f(x) \approx \frac{1}{2} x^T A x - b^T x + c
  \]
  
  $A$ is a known, square, symmetric, positive-definite matrix.

- Then $f(x)$ is minimized by the solution to
  \[
  A x = b
  \]

- We can, in general, calculate $f(P)$ and $\nabla f(P)$ for a given $N$-dimensional point $P$.

- How can we use this additional information?
• An tempting (but not very good) possibility: steepest descent method.

• Start at a point $P_0$. As many times as needed, move from point $P_i$ to the point $P_{i+1}$ by minimizing along the line from $P_i$ in the direction of the local downhill gradient $-\nabla f(P_i)$.

• Risk of over-shooting.

• To avoid it: perform many small steps (perhaps with line search) $\Rightarrow$ Not very efficient!
Steepest descent method
Conjugate gradient method

- A better way.

- In $\mathbb{R}^N$ take $N$ steps each of which attains the minimum in each direction, w/o undoing previous steps’ progress.

- In other words, proceed in a direction that is somehow constructed to be conjugate to the old gradient, and to all previous directions traversed.
Conjugate gradient method
Algorithm - linear

1. Let $d_0 = r_0 = b - Ax_0$.

2. For $i = 0, 1, 2, \ldots, N - 1$ do:
   
   \begin{itemize}
   \item $\alpha_i = \frac{r_i^T r_i}{d_i^T A d_i}$.
   
   \item $x_{i+1} = x_i + \alpha_i d_i$.
   
   \item $r_{i+1} = r_i - \alpha_i A d_i$.
   
   \item $\beta_{i+1} = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$.
   
   \item $d_{i+1} = r_{i+1} + \beta_{i+1} d_i$.
   \end{itemize}

3. Return $x_N$. 
Algorithm - non-linear

1. Let $d_0 = r_0 = -f'(x_0)$.

2. For $i = 0, 1, 2, ..., N - 1$ do:
   - Find $\alpha_i$ that minimizes $f'(x_i + \alpha_i d_i)$.
   - $x_{i+1} = x_i + \alpha_i d_i$.
   - $r_{i+1} = -f'(x_{i+1})$.
   - $\beta_{i+1} = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$ or $\beta_{i+1} = \max \left\{ \frac{r_{i+1}^T (r_{i+1} - r_i)}{r_i^T r_i}, 0 \right\}$.
   - $d_{i+1} = r_{i+1} + \beta_{i+1} d_i$.

3. Return $x_N$.  

Go Back