Function Minimization for Dynamic Programming Using Connectionist Networks

Leemon C. Baird III
Wright Laboratory
Wright-Patterson Air Force Base, OH 45433-6543

Abstract **Learning controllers based on dynamic programming require some means of storing arbitrary functions and finding global minima within cross sections of those functions. There are many general methods for learning and representing functions, including polynomials, multi-layer perceptrons with backpropagation, and radial basis functions, but these systems do not allow the minima to be found easily. A method is presented here for learning and finding the minima of all cross sections of an arbitrary, smooth function. This method is applicable to any general function approximation system that learns smooth functions from examples. Mathematical properties of this approach are described, applications to learning control are discussed, and simulation results are presented.**

INTRODUCTION

Optimal control of a nonlinear, poorly modeled system is a difficult problem. In recent years, a number of systems have been designed for this problem that are based on dynamic programming [1]-[5],[8],[9],[12]. Such systems generally have two significant properties: they must learn functions from experience, and they must be able to find the minima of these functions. Many well-studied methods are widely used for learning arbitrary functions. These include high-order polynomials, multi-layer perceptrons with backpropagation learning [6],[7], and radial basis functions. Unfortunately, it is often impossible to find the minimum of a function analytically when it is represented using these techniques. The problem to be solved can be summarized as follows. Given a vector $\mathbf{x}$, find the vector $\mathbf{u}$ that minimizes $f(\mathbf{x},\mathbf{u})$, where f is an arbitrary, smooth function. This paper proposes one method for solving the problem by learning a function $F(\mathbf{x},\mathbf{u},p)$ whose domain has one additional dimension. The function $F(\mathbf{x},\mathbf{u},0)$ is trained to be equal to $f(\mathbf{x},\mathbf{u})$, and the function $F(\mathbf{x},\mathbf{u},p)$ for $p>0$ is trained in accordance with a simple differential equation.

Manuscript received August 1, 1992. This research was supported under Task 2312R1 by the Life and Environmental Sciences Directorate of the United States Air Force Office of Scientific Research, and by a Charles Stark Draper Laboratory Fellowship.

MINIMIZATION

The approach presented here will be applied to a discrete problem first, then to the continuous problem. Consider the problem of finding the minimum of the following list of digits:

4 5 2 4 6 9 4 3

One way to solve the problem is to take each pair of digits in the list and write the minimum of that pair on the line above. Repeating this gives the triangular array of digits shown in fig. 1.

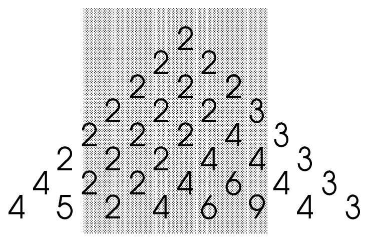

Fig 1. Triangular array of digits.

Given an array generated in this manner, the minimum of the list will always be found at the top. The location of the minimum can be found by starting at the top and repeatedly moving to the minimum of the two adjacent numbers on the line below, as shown by the shaded path in fig. 1. This gradient descent path always ends at the minimum digit in the original list.

The following two conditions uniquely specify each of the values in the triangular array.

- The bottom row of the array must consist of the list to be minimized. In other words, if *A* is the *k*th number on the bottom row of the array, and *N* is the *k*th number in the list to be minimized, then:

$$A = N \tag{1}$$

- For any adjacent numbers *A*, *B*, and *C* in this arrangement:

  A
  B C

  the following condition holds:

$$\min(B - A, C - A) = 0 \tag{2}$$

Given a triangular array initialized to arbitrary values, the two conditions can be used to find the minimum of a list. This is done by selecting a value *A* from the array at random. This value is then modified to satisfy the above conditions. If this is done repeatedly, then the bottom row gradually becomes the same as the list to be minimized, and the upper rows gradually come to reflect the minimum of the bottom row. With probability 1, the array will converge to the desired values.

This approach can be generalized to find the minimum of a continuous function. Instead of the minimum digit of a list, the problem is to find the minimum value of a function *f*(*u*) over the interval [-1,1]. Just as the minimum digit was found using a triangular array of digits, so the minimum of *f* can be found using a function defined over a triangular region. This function can be stored in a connectionist network with two inputs, *u* and *p*, and one output, *F*(*u*,*p*). The network can be any general function approximation system, such as a backpropagation, multi-layer perceptron network.

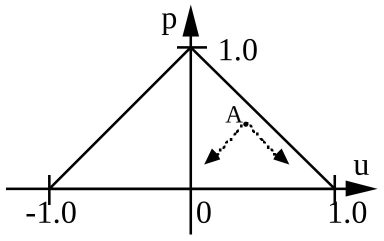

Fig. 2. Triangular domain of function *F*(*u*,*p*).

The inputs, *u* and *p*, are the axes in fig. 2. Given a point on the triangle or within it, the network will output the value *F*(*u*,*p*) associated with that point. Equation (3) is the continuous version of (1).

$$\forall u \in [-1,1] \quad F(u,0) = f(u) \tag{3}$$

In (2), the value at point *A* is determined by the minimum of two differences. These differences are taken between adjacent numbers in diagonal directions. For the continuous problem, those differences are replaced by directional derivatives along the diagonal directions. Define $D_{(u_1,p_1)}F(u,p)$ to be the directional derivative of *F* in the direction (u_1,p_1) evaluated at (*u*,*p*). The continuous version of (2) is (4).

$$\forall p \in (0,1], u \in [-(1-p), 1-p] \quad \min\left(D_{(-1,-1)}F(u,p), D_{(1,-1)}F(u,p)\right) = 0 \tag{4}$$

Equations (3) and (4) describe the properties that the function *F* must have. It can be proven that if the function *f*(*u*) and its first derivative are continuous, then there does exist a function *F* that satisfies (3) and (4). Furthermore, it can be shown that this solution is unique, is continuous, and can be written in closed form as:

$$\forall p \in (0,1], u \in [-(1-p), 1-p] \quad F(u,p) = \min_{v \in [u-p, u+p]} \left(f(v)\right) \tag{5}$$

The value of *F* at a given point is the minimum of *f* over some region. That region is small for small *p* and larger for larger *p*. For *p*=1 the minimum is over the entire domain, [–1,1].

Once *F* has been found, the minimum of *f*(*u*) is simply *F*(0,1). A value of *u* that minimizes *f*(*u*) can be found by the gradient descent process that starts at *F*(0,1) and moves at a steady rate in the negative *p* direction, while following the gradient in the *u* direction. When *p* reaches zero, *u* will be a value that minimizes *f*(*u*).

To find the function *F* over the triangular domain, an approximation of *F* will need to be stored and gradually improved. Assume that *F* is stored in a function approximation system that is continuous and has a continuous gradient at every point. *F* will be improved using (3) and (4), which requires finding the minimum of two directional derivatives. Let *m* be the minimum of the two directional derivatives, which can be calculated quickly using (6).

$$\begin{aligned} m &= \min\left(D_{(-1,-1)}F(u,p), D_{(1,-1)}F(u,p)\right) \\ &= \min\left(\nabla F \cdot \left(\frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}\right), \nabla F \cdot \left(\frac{-1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}\right)\right) \\ &= \frac{1}{\sqrt{2}} \min\left(\frac{\partial F}{\partial u} - \frac{\partial F}{\partial p}, -\frac{\partial F}{\partial u} - \frac{\partial F}{\partial p}\right) \end{aligned}$$

$$m = \frac{1}{\sqrt{2}}\left(-\frac{\partial F}{\partial p} - \left|\frac{\partial F}{\partial u}\right| \right) \tag{6}$$

If **u** is a two dimensional vector and *p* remains a scalar, then *F* becomes a function over the points in a pyramid instead of in a triangle. If **u** and **d** are *n* dimensional vectors and *p* is scalar, then (3) and (4) become (7) and (8).

$$\forall i \in \{1,2,\ldots,n\},\ u_i \in [-1,1] \quad F(\mathbf{u},0) = f(\mathbf{u}) \tag{7}$$

$$\forall i \in \{1,2,\ldots,n\},\ p \in (0,1], u_i \in [-(1-p), 1-p]$$
$$\min_{\mathbf{d} \in \{-1,1\}^n} \left(D_{(\mathbf{d},-1)} F(\mathbf{u},p) \right) = 0 \tag{8}$$

If the function *f*(u) is continuous and has a continuous gradient, then the function *F* that satisfies (7) and (8) exists, is unique, is continuous, and can be written in closed form as:

$$\forall i \in \{1,2,\ldots,n\},\ p \in (0,1], u_i \in [-(1-p), 1-p]$$
$$F(\mathbf{u},p) = \min_{\{\mathbf{v} | v_i \in [u_i - p, u_i + p]\}} (f(\mathbf{v})) \tag{9}$$

The differential equations can be solved by an iterative method, using a network to store the current approximation of *F*. The network used to solve these equations should have *n*+1 inputs (*p* and the elements of **u**), and one output, *F*. The stored function should be repeatedly adjusted at randomly-chosen points so that (7) and (8) are satisfied.

The above discussion assumes there is only one function to minimize, *f*(**u**). There could also be a parameterized family of functions, each of which is to be minimized. In that case, all of the above remains true for each of these functions. The entire family of functions can be stored in a single network and trained simultaneously, possibly leading to useful generalization between functions.

In summary, to find a **u** that minimizes *f*(**x**,**u**) for each possible value of **x,** the following algorithm should be used.

ALGORITHM TO MINIMIZE *f*(**x**,**u**)

```
• Initialize the net to a convenient function (e.g. $F$=0)
• Repeat the following steps
    • Randomly choose a value for $\mathbf{x}$
    • Randomly choose $p$ to be either zero or a value from (0,1],
    • Randomly choose each element $u_i$ from $[-(1-p),1-p]$
    • If $p = 0$ then
        • Train the net to increase $F(\mathbf{x},\mathbf{u},p)$ by an amount
          proportional to $f(\mathbf{x},\mathbf{u})-F(\mathbf{x},\mathbf{u},p)$
    • else
        • Train the net to increase $F(\mathbf{u},p)$ by an amount
          proportional to
```

$$\frac{\partial F(\mathbf{x},\mathbf{u},p)}{\partial p} + \sum_{i=1}^{n} \left| \frac{\partial F(\mathbf{x},\mathbf{u},p)}{\partial u_i} \right| \tag{10}$$

Expression (10) is the multidimensional version of (6), and represents the negative of the minimum of the directional derivatives. The network is adjusted to make this value closer to zero, thus satisfying (8).

As this procedure is repeated, the function *F*(**x**,**u**,0) changes to become equal to *f*(**x**,**u**). The function *F*(**x**,**u**,*p*) for p>0 changes to reflect the minimum of *f*(**x**,**u**). After training, the value *F*(**x**,0,1) is equal to the minimum of *f*(**x**,**u**).

Note that if the plus sign before the summation in (10) is changed to a minus sign, then the algorithm finds the maximum value instead of the minimum value.

SIMULATION RESULTS

Fig. 3 shows simulation results. A three-layer, sigmoidal, backpropagation network was used to learn the function *F*(*u*,*p*). The top graph shows the function to be minimized, *f*(*u*). The bottom graph shows the function *F*(*u*,*p*) after learning. As desired, the network converged to the correct function, with the minimum value of *f*(*u*) at the apex of the triangle, and a valley leading from the apex to the minimizing value of *u*.

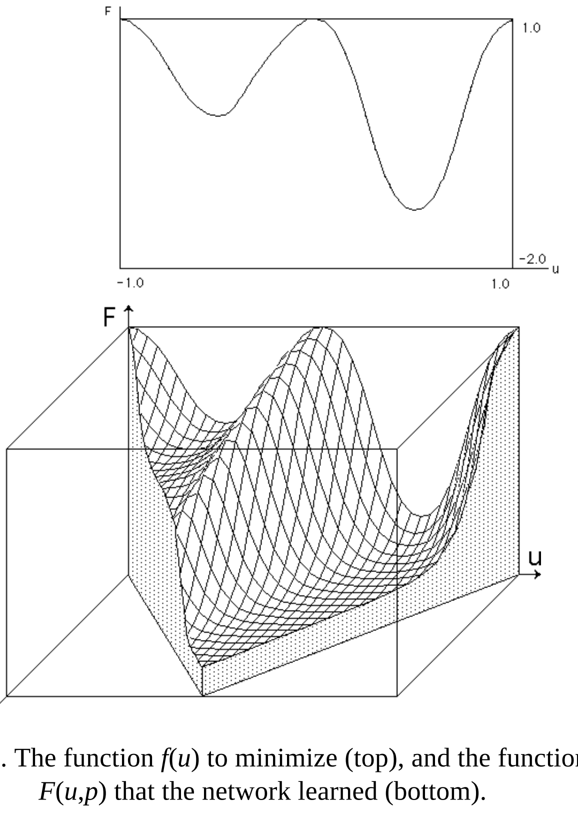

Fig. 3. The function *f*(*u*) to minimize (top), and the function *F*(*u*,*p*) that the network learned (bottom).

One feature of this algorithm is that, when minimizing a function with two local minima, it will learn valleys of low values starting from those two points and heading toward the apex of the triangle, with the lower valley ending at the apex and the higher valley ending before it reaches the apex. This is useful if the function $f(\mathbf{u})$ changes, as is often the case in a dynamic programming system. If $f(\mathbf{u})$ changes slightly so that the other local minimum becomes lower, then all the network must do is extend one valley and shorten the other one. For most types of networks, shifting ridges and valleys is a faster operation than learning to create new valleys. This suggests that this algorithm should be capable of tracking the minimum of a slowly changing function. It has the further advantage of not needing to be reset or started over whenever the function being minimized changes. These are important properties if this system is used as part of a dynamic programming system.

Gradient descent

Once the network has learned F, it is easy to find the minimum of $f(\mathbf{x},\mathbf{u})$ for a given $\mathbf{u}$. It is simply the value $F(\mathbf{x},0,1)$. The $\mathbf{u}$ that yields that minimum value can be found by a gradient descent method. Algorithm GRAD returns that value of $\mathbf{u}$.

Algorithm GRAD($\mathbf{x}$,$\mathbf{u}$,p)

```
• Initialize each $u_i$ to 0
• For $p$ decreasing from 1 to 0 by some small step size $s$,
      loop:
      • Calculate each element of the minimum directional
        derivative vector:
            $$d_i = -s \cdot \mathrm{sgn}\left(\frac{\partial F(\mathbf{x},\mathbf{u},p)}{\partial u_i}\right) \qquad (11)$$
      • $\mathbf{u} \leftarrow \mathbf{u} + \mathbf{d}$
• End loop
• Return $\mathbf{u}$
```

This algorithm is useful after learning is finished, but it might also be useful during learning. Learning might be improved if expression (10) in the learning algorithm were replaced with (12).

$$F(\mathbf{x},\mathrm{GRAD}(\mathbf{x},\mathbf{u},p),0) - F(\mathbf{x},\mathbf{u},p) \qquad (12)$$

This would allow information to flow towards the apex of the triangle more quickly, and might allow learning in fewer training steps. It would, however, increase the computational cost per time step during training, since it involves an entire gradient descent on each step.

Q-Learning

The most natural use of this minimization technique for learning control is as a component of a Q-learning system. Q-learning is a form of dynamic programming developed and proven optimal by Watkins [11]. It requires a number to be stored for each state-action pair $(\mathbf{x},\mathbf{u})$, where $\mathbf{x}$ is the state of the system being controlled at a given point in time, and $\mathbf{u}$ is the control action chosen by the controller in response to that state. Both $\mathbf{x}$ and $\mathbf{u}$ are vectors, possibly high-dimensional vectors. If the system is in state $\mathbf{x}(t)$ and the controller responds with action $\mathbf{u}(t)$, then the resulting state will be $\mathbf{x}(t+1)$, and the system incurs a cost given by the function $C(\mathbf{x}(t),\mathbf{u}(t))$. After performing a given action, the associated value $Q(\mathbf{x}(t),\mathbf{u}(t))$ should be updated to be closer to the value:

$$C(\mathbf{x}(t),\mathbf{u}(t)) + \gamma \cdot \min_{\mathbf{u}}\big(Q(\mathbf{x}(t+1),\mathbf{u})\big) \qquad (13)$$

where γ is the discount factor and lies in the range [0,1].

After the Q-values have converged the controller should always respond to a state $\mathbf{x}$ by choosing the action $\mathbf{u}$ that minimizes $Q(\mathbf{x},\mathbf{u})$. If $\mathbf{x}$ and $\mathbf{u}$ can only take on a finite number of values, and if the Q-values are stored in a table, then Watkins has proven that, if each action is performed sufficiently often in each state during learning, then this method will result in a controller that minimizes the total, discounted cost:

$$\sum_{t=0}^{\infty} \gamma^t C(\mathbf{x}(t),\mathbf{u}(t)) \qquad (14)$$

When the values are continuous, the Q function must be approximated with some function approximation system, and the problem of finding the minimum becomes more difficult. The minimization technique presented here incrementally works on finding the minimum for many states simultaneously. It may also allow useful generalization. Fig. 4 gives an example of a case where useful generalization would be expected to occur.

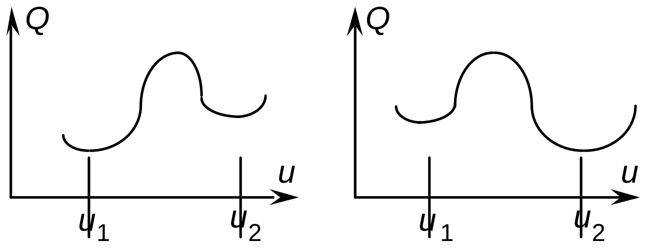

Fig. 4. Q-values for state $\mathbf{x}_1$ (left) and x_2 (right).

In fig. 4, states $\mathbf{x}_1$ and $\mathbf{x}_2$ are assumed to be near each other in state space. The best action in state $\mathbf{x}_1$ is u_1 and in state $\mathbf{x}_2$ is u_2. Using the proposed method, the F function contains valleys leading to both local minima in both states, with one valley slightly lower than the other in each case. If states $\mathbf{x}_1$ and $\mathbf{x}_2$ are near each other, and if it is only trained with data from $\mathbf{x}_1$ and $\mathbf{x}_2$, then it may tend to generalize in the states between $\mathbf{x}_1$ and $\mathbf{x}_2$. As the state moves from $\mathbf{x}_1$ to $\mathbf{x}_2$, the valley at u_1 gradually rises, and the valley at u2 gradually drops. The action recommended by the system will therefore be u_1 up to the point where the valleys are equal, and will be u_2 thereafter. This is a reasonable generalization, and will lead to recommendations that are monotonically improving.

Less useful generalization would be expected from a learning controller using a *policy* network. A policy network simply stores the optimal action u for each **x.** If such a network were trained only for $\mathbf{x}_1$ and $\mathbf{x}_2$, it would recommend u_1 at $\mathbf{x}_1$ and u_2 at $\mathbf{x}_2$. In the intermediate states between $\mathbf{x}_1$ and $\mathbf{x}_2$ it would recommend actions between u_1 and u_2. In this example, these are the worst possible actions. This suggests that if it is likely that the Q functions will have local minima (at least during learning), then a policy network may be a less useful approach to learning control. The method using the function F as proposed here may be better.

FINDING SADDLEPOINTS

Dynamic programming networks have been used successfully to learn good strategies for board games [10], and may be useful for playing differential games. Differential games are games where the control actions are done continuously, such as in two player video games. Each player controls certain elements of **u**, and the entire state is visible to both players and is represented by **x**. For some games the optimal strategy is a function of the state. For other games, the optimal strategy is to pick actions stochastically, where the probabilities are functions of the state. Some games do not even have optimal strategies.

One particularly interesting game always has a deterministic optimal strategy. One player acts as a controller, trying to minimize cost, and the other player acts as a disturbance, trying to maximize cost. On each time step, the controller looks at the state and chooses an action. The disturbance then looks at that state and the controller's action, and chooses its own action. Finally, these two actions are performed, and the state changes. This game is simply another way of viewing *robust control* [1]. The goal of robust control is to build a controller that is optimal, not under the average disturbance conditions, but under worst-case disturbance conditions. "Worst case" is defined as whatever disturbances are worst for the controller built. Thus, if a system is capable of learning to play differential games with deterministic strategies, then it is also able to learn robust, optimal control.

A learning system such as *Q-learning* can be modified to handle games. The primary difference is that instead of finding minima it must find saddlepoints. If player number one controls action u_1 and player number two controls action u_2, then the optimal action for each player in state **x** is the action pair (u_1,u_2), such that:

$$Q(x,u_1,u_2) = \min_{v} \max_{w} Q(\mathbf{x},v,w) \tag{15}$$

In this case Q represents the value of the game if action (u_1,u_2) is performed followed by optimal playing thereafter. A low Q-value is in player one's favor, and a high value is in player two's favor. Equation (15) represents the value of the game if player two has the advantage of knowing $u_1(t)$ before choosing $u_2(t)$, as in the case of robust control. If the *min* and *max* were reversed, then player one would have that advantage. In many games, the value remains unchanged when the *min* and *max* are reversed; in these games, the point (u_1,u_2) is called the saddlepoint. It is assumed here that such a saddlepoint exists. Saddlepoints are generally more difficult to find than minima, yet the method presented here can be extended to find saddlepoints.

Fig. 4 shows the domain of the F function used to minimize $f(u_1,u_2)$. The domain is a pyramid, and after learning the minimum will appear at the top. If the sign of the summation in (10) were changed, then the maximum would appear at the top. Fig. 5 shows the domain for finding the maximum for each possible u_1. For each u_1, there is a triangular region used to find the maximum of $f(u_1,u_2)$. This is analogous to the triangle in fig. 2. During training, the top edge of this prism comes to represent the maxima of each of these cross sections. Fig. 6 shows the single triangle used to find the minima of these maxima. The base of the triangle consists of all of the maxima, and the apex of the triangle comes to represent the minimum of the maxima. Both the lower prism and the upper triangle could be trained simultaneously, and the saddlepoint value would eventually appear at the top. Thus the value at the top will be the value of the game. The (u_1,u_2) value can be found by starting at the top and doing gradient descent in the p and u_1 directions for half the distance, then doing gradient *ascent* in the p and u_2 directions for the rest of the distance. The fact that the new minimization method solves multiple minimization problems in parallel makes it particularly useful for finding saddlepoints. Other minimization methods based on gradient descent, simulated annealing, etc., would be more difficult to adapt to the saddlepoint problem.

CONCLUSIONS

An algorithm has been presented for finding the minimum of each of a family of parameterized functions. Simulation of the algorithm demonstrates its ability to work on a simple problem. The properties of the algorithm appear to make it useful for learning control using dynamic programming techniques, especially Q-learning. It is also extendable to finding saddlepoints, since it can work on an infinite number of minimization problems in parallel. These appear to be fruitful areas for further research.

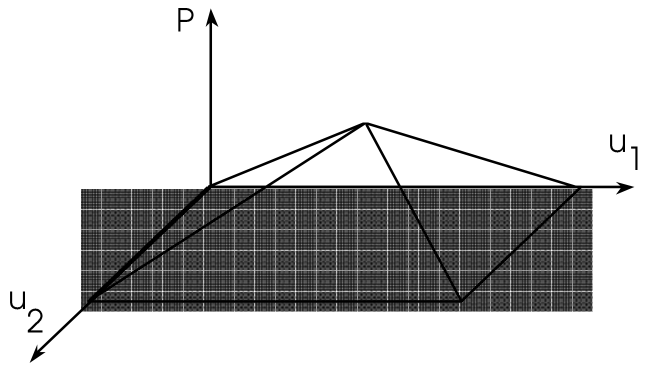

Fig. 4. Domain of $F(u_1,u_2,p)$ when finding the minimum of $f(u_1,u_2)$ over all u_1,u_2.

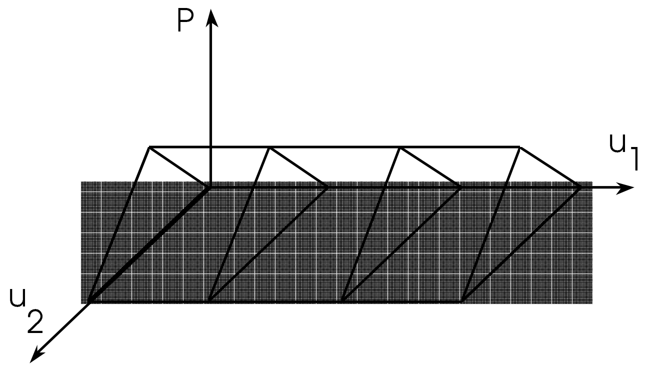

Fig. 5. Domain of $F(u_1,u_2,p)$ when finding the maximum of $f(u_1,u_2)$ for each possible u_1. Each triangle is analogous to the triangle in fig. 2.

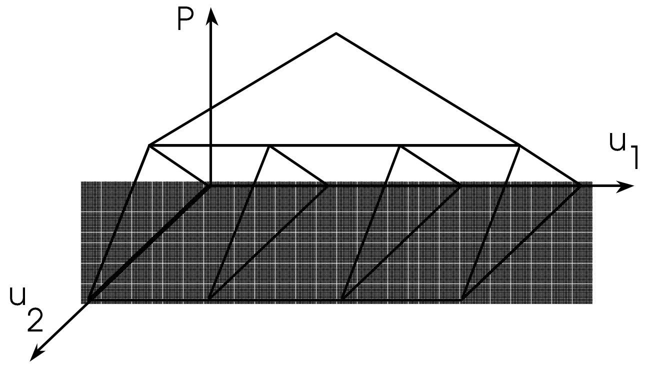

Fig. 6. Domain of $F(u_1,u_2,p)$ when finding the minimum over $u_1$of the maximum of $f(u_1,u_2)$. The upper triangle finds the minimum of the maxima.

References

[1] Baker, W., personal communication, 1991.

[2] Barto, A., "Connectionist Learning for Control: An Overview," COINS Technical Report 89-89, Department of Computer and Information Science, University of Massachusetts, Amherst, September, 1989.

[3] Barto, A., and S. Singh, "Reinforcement Learning and Dynamic Programming," *Proceedings of the Sixth Yale Workshop on Adaptive and Learning Systems,* New Haven, CT, August, 1990.

[4] Barto, A., R. Sutton, and C. Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-13, No. 5, September/October 1983.

[5] Barto, A., R. Sutton, and C. Watkins, "Learning and Sequential Decision Making," COINS Technical Report 89-95, Department of Computer and Information Science, University of Massachusetts, Amherst, September, 1989.

[6] Hornik, K., and M. White, "Multilayer Feedforward Networks are Universal Approximators," *Neural Networks*, Vol. 2, 1989.

[7] Rumelhart, D., G. Hinton, and R. Williams, "Learning Internal Representation by Error Propagation," *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1, Rumelhart, D., and J. McClelland, ed., MIT Press, Cambridge, MA, 1986.

[8] Sutton, R., "Artificial Intelligence by Approximating Dynamic Programming," *Proceedings of the Sixth Yale Workshop on Adaptive and Learning Systems,* New Haven, CT, August, 1990.

[9] Sutton, R., "Learning to Predict by the Methods of Temporal Differences," *Machine Learning*, Kluwer Academic Publishers, Boston, MA, vol. 3: 9-44, 1988.

[10] Tesauro, G., “Neurogammon: a Neural Network Backgammon Program.” IJCNN Proceedings III, 33-39 1990.

[11] Watkins, C., "Learning from Delayed Rewards," Ph.D. thesis, Cambridge University, Cambridge, England, 1989.

[12] Williams, R. and L. Baird, "A Mathematical Analysis of Actor-Critic Architectures for Learning Optimal Controls Through Incremental Dynamic Programming," *Proceedings of the Sixth Yale Workshop on Adaptive and Learning Systems,* New Haven, CT, August, 1990.