Reinforcement Learning in Video Games Using Nearest Neighbor Interpolation and Metric Learning

Matthew S. Emigh, Evan G. Kriminger, Austin J. Brockmeier, *Member, IEEE*, José C. Príncipe, *Fellow, IEEE*, and Panos M. Pardalos

***Abstract*—Reinforcement learning (RL) has had mixed success when applied to games. Large state spaces and the curse of dimensionality have limited the ability for RL techniques to learn to play complex games in a reasonable length of time. We discuss a modification of Q-learning to use nearest neighbor states to exploit previous experience in the early stages of learning. A weighting on the state features is learned using metric learning techniques, such that neighboring states represent similar game situations. Our method is tested on the arcade game Frogger, and it is shown that some of the effects of the curse of dimensionality can be mitigated.**

***Index Terms*—Games, metric learning, nearest neighbor, reinforcement learning.**

I. Introduction

LEARNING to successfully solve or "win" a game is the process of discovering the right actions to achieve goals or subgoals. As players learn to play a game they begin to associate actions or sequences of actions with particular game scenarios. Gameplay can be considered a decision process in which a player must decide, based on his current knowledge of the game world, what actions or sequences of actions will best lead to the game's objective. More generally, the player learns a policy that maps any given game scenario to an action permitted by the game world.

Traditional artificial intelligence methods, such as finite state machines, hard code policies based on expert knowledge of the game and its goals. Such a policy is unable to adapt to changing circumstances. Furthermore, the policy is limited by the knowledge of the AI designer.

Reinforcement learning (RL) [1], [2] is a natural paradigm for a machine to mimic the biological process of learning to complete objectives. It is applied to sequential decision problems, in which the outcome of an action may not be immediately apparent. With RL, credit can be assigned to the actions leading to an outcome, which provides feedback in the form of a reward. The feedback that occurs in response to actions is combined with previous experience. Thus, RL serves as a principled means of learning policies for decision making directly from experience. RL has been successfully applied to games, most famously with Tesauro's world-class backgammon player, TD-gammon [3]. RL has also been applied to games such as Tetris [2] and chess [4], [5].

Most approaches described in the references above are limited in their application to other games. While RL agents learn their policies and are thus not tied to any particular application, this generality of RL has not been effectively exploited for games. This is the result of the difficulties in choosing the relevant variables to represent a game scenario for the RL algorithm. If an agent is not given enough information about the state of the game, the results of actions cannot be attributed to the proper cues from the game, and thus the agent will never learn. However, when game scenarios are represented with many variables, the problem often becomes so large that the agent requires an unreasonable amount of experience to test actions in all important game scenarios. This problem is known as the "curse of dimensionality" [6], and it motivates finding efficient representations of game scenarios, or "feature selection". While feature selection is well-studied for other machine learning tasks, such as classification [7], in RL it is often done heuristically or with expert knowledge [1].

Even with careful feature selection, there are still too many game scenarios for an agent to exhaustively explore all possibilities. When a previously unseen scenario arises, it is important to exploit previous experience by interpolating amongst similar scenarios [8], [9] . For this paper, we use the simple nearest neighbor interpolation method for its simplicity in both implementation and computation. Previous examples of nearest neighbor interpolation being used with RL methods include [10] and [11]. Gordon [9] proved that for fitted dynamic programming and RL methods, k-nearest neighbor is a stable function approximator. The appropriate choice for this interpolation turns out to be highly dependent on the way a game scenario is represented. Thus, we will look at various representations and their effect on the ability of RL to learn in terms of learning speed and goodness of the final policy. We will use a (Python) clone of the classic 1981 arcade game, "Frogger," as shown in Fig. 1, as our test bed. Similar work on another Frogger clone has been performed by Cobo [12], using learning from demonstration (LfD) and state abstraction combined with RL methods. Wintermute [13] experimented with another similar game, "Frogger II," using imagery to predict future states and Q-learning to assign a value to abstracted states.

Manuscript received January 23, 2014; revised August 19, 2014; accepted October 29, 2014. Date of publication November 10, 2014; date of current version March 15, 2016. This work was supported by Air Force Office of Scientific Research Grant FA9550-13-1-0142.

M. S. Emigh, E. G. Kriminger, and A. J. Brockmeier are with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611 USA (e-mail: matt@cnel.ufl.edu; evankriminger@gmail.com; ajbrockmeier@gmail.com).

J. C. Príncipe is with the Department of Electrical Engineering and Biomedical Engineering, University of Florida, Gainesville, FL 32611 USA (e-mail: principe@cnel.ufl.edu).

P. M. Pardalos is with the Department of Industrial and Systems Engineering, University of Florida, Gainesville, FL 32611 USA (e-mail: pardalos@ufl.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCIAIG.2014.2369345

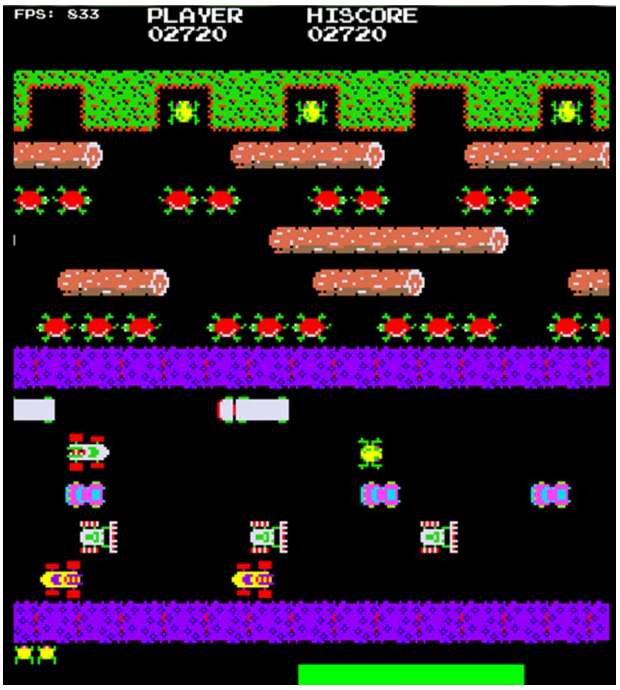


Fig. 1. Screenshot of Frogger game. In a normal game, the player controls the frog using a joystick or keyboard. The only controls are up, down, left, and right. The goal of the game is to maneuver the frog to the five home positions (shown at the top of the screen), without allowing the frog to be hit by one of the cars (bottom half of screen), or drown in the river (top half of screen). The player navigates the river by jumping on the logs and turtles. Each time the player maneuvers the frog to the one of the home positions, a new player-controlled frog is respawned at the bottom of the screen. Once all five home positions are filled, a new level begins and process is repeated. Pitfalls for the player to watch out for include turtles periodically diving underneath the water, and a crocodile that fills one of the home positions at random intervals.

In the rest of this paper we will provide a background on reinforcement learning, then introduce a novel approach to reinforcement learning. Our method consists of a nearest neighbor approach to learn the value functions of RL combined with a metric learning method designed to provide more weighting to features that are highly relevant to the game task [14]. Experimental results are provided for the game of Frogger, providing a case study for feature extraction in games.

II. Background

A. Markov Decision Processes

Reinforcement learning is formulated for problems posed as Markov Decision Processes (MDP). An MDP is a model for a certain class of stochastic control problems. A (discounted) MDP can be represented mathematically as a 5-tuple $(\mathcal{X}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{X}$ is the set of all possible states (state space), $\mathcal{A}$ is the set of all possible actions (action space), $\mathcal{P}$ is the set of transition probabilities, $\mathcal{R}$ is the set of rewards, and γ is the discount factor. Since the available actions are typically state-dependent, we define $\mathcal{A}(x) = \{a \in \mathcal{A} | a \text{ is allowed in } x\}$. When used to model a game world, the state space $\mathcal{X}$ represents all possible game scenarios, and the action space $\mathcal{A}(\cdot)$ is the set of possible actions the player can take in any given scenario. A state or action occurring at time t is denoted by X_t and A_t, where $X_t \in \mathcal{X}$ and $A_t \in \mathcal{A}$. When a player takes an action a in a certain game state x, this leads to a new game state x'. This new state is a random variable, and a probability is assigned to the transition as follows: $P(x, a, x') = P(X_{t+1} = x' | X_t = x, A_t = a)$. Note that the next state probability is conditioned only on the current state and action. This is known as the Markov property, and is the primary assumption for modeling with an MDP.

Associated with each state transition is a reward. We write R_{t+1} to denote the reward received when the state transitions from X_t to X_{t+1}. Since state transitions are random, rewards R_{t+1} are as well. In games, the rewards of each state transition are usually naturally defined. For instance, winning the game results in a positive reward, while all other steps result in zero reward.

Since typically only a small fraction of possible state transitions receive a nonzero reward, credit must be assigned to actions that indirectly led to the rewarded state. This is achieved by evaluating actions with the rewards that are received at current *and* future times, which is formalized in a criterion known as the Q-value, or value function. The value function is defined as the sum of expected future discounted rewards

$$Q^{\pi}(x, a) = E\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | X_0 = x, A_0 = a\right], 0 \leq \gamma < 1 \tag{1}$$

starting in state x and performing action a. γ is a discounting factor, which decays rewards received further in the future. This is the discounted value function; there are also value functions based on time-averaged rewards and on finite time-horizon sums of rewards.

The expectation in (1) is taken over the transition probabilities of the MDP and under the policy π, which is a mapping from states X to actions A. The policy may be stochastic or deterministic. If the policy is stochastic, then it is a mapping from the states X to a distribution over A. The MDP is considered to be solved when the policy maximizes the expected future sum of rewards from any state $X \in \mathcal{X}$. We call $\pi^{\star}$ the optimal policy for the MDP. Every MDP has at least one optimal policy $\pi^{\star}$[2].

For a more concrete example, consider chess from the point of view of one of the players. The state space $\mathcal{X}$ is the set of all possible board configurations, while a state at time t, X_t, is the configuration of the board after both the player and opponent have performed t moves. The (restricted) action space $\mathcal{A}(x)$ is the set of legal moves the player can take while the chess pieces are in configuration x. Since at any time t, the player does not know what the opponent's next move will be, the transition to the next board configuration is stochastic. If a move leads to a state in which the player captures a chess piece, a positive reward can be assigned to that transition. If a piece belonging to the player is captured, that transition can be assigned a negative reward. Similarly, checkmating the opponent can lead to a very large positive reward, while being checkmated produces a very large negative reward. Solving this problem amounts to selecting the best action in any given state in order to maximize future expected rewards.

It is intractable to calculate the value functions of MDPs directly because (1) requires expectations over all future times. The tools of dynamic programming (DP) [6] allow the value

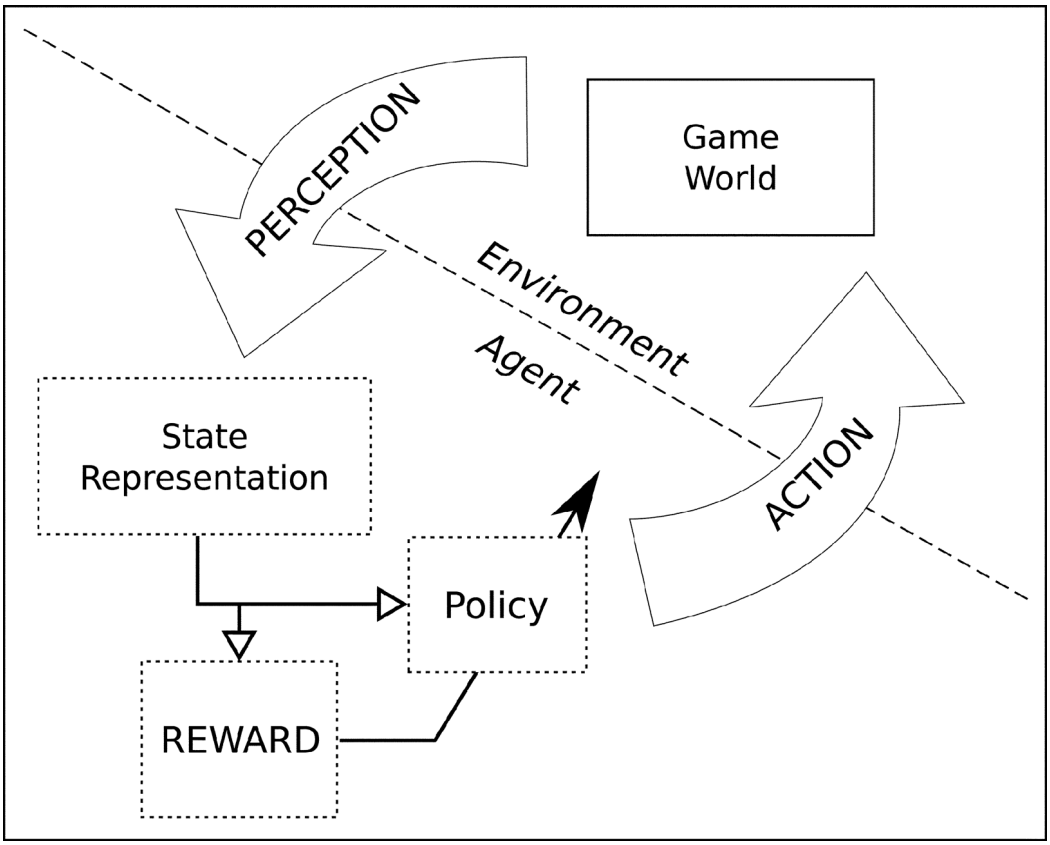


Fig. 2. The perception-action-reward cycle describes the processing of an agent playing a game. The player forms an internal state representation of the gaming environment through perception. Based on the current game state the player's policy determines which action to take. After each action, or inaction, the gaming environment transitions from its current state to the next, yielding a reward. The player forms modifies the policy based on the observation of this transition.

functions to be computed efficiently by using recursion. If (1) is rewritten as

$$Q^{\pi}(x,a) = R_1 + E\left[\sum_{t=1}^{\infty} \gamma^t R_{t+1} | X_0 = x, A_0 = a\right]$$

it is clear that the summation now represents the value function at the next state $\gamma Q^{\pi}(X_1, A_1)$. We can thus rewrite (1) as

$$Q^{\pi}(x_t, a_t) = E[R_{t+1} + \gamma Q^{\pi}(X_{t+1}, A_{t+1})] \tag{2}$$

where R_{t+1} is dependent on x_t and a_t.

In (2) the expectation is now over only a single state transition. DP computes the value functions using iterative methods, and the optimal policies can be found by choosing actions which maximize the value function at each state. However, even with the recursion of DP, the state transition probabilities as well as the rewards for each transition must be known. For real problems with very large state spaces, this is an unreasonable assumption.

B. Reinforcement Learning

Reinforcement learning is an area of machine learning which focuses on techniques and algorithms to learn the solution to MDPs using information gained from interacting with the environment. Many RL algorithms can be viewed in the framework of the biologically inspired perception-action-reward cycle (PARC), see Fig. 2.

Q-learning [15] is an RL algorithm that learns the optimal Q-function, $Q^{\star}(x,a)$, for each state-action pair. The optimal policy $\pi^{\star}$ can be written in terms of the optimal Q-function as $\pi^{\star}(x) = \arg\max_a Q^{\star}(x,a)$ Q-learning solves $E[R_{t+1} + \gamma \max_a Q(X_{t+1}, a)]$ using stochastic approximation [16], thus removing the need to calculate expectations and learning Q-functions directly from observed $(X_t, A_t, R_{t+1}, X_{t+1})$. The classical Q-learning algorithm is defined by the following update:

$$Q(X_t, A_t) \leftarrow Q(X_t, A_t) + \alpha\delta$$
$$\delta = R_{t+1} + \gamma \max_{a'} Q(X_{t+1}, a') - Q(X_t, A_t) \tag{3}$$

where α is a small learning rate, and $Q(\cdot,\cdot)$ is the current Q-function estimate.

Suppose $x, y \in \mathcal{X}$ and $a \in \mathcal{A}$, the action a is taken in x, and the state transition $x \to y$ is observed. Furthermore, suppose reward r_{xy} is received. Then the term $r_{xy} + \gamma \max_{a'} Q(y, a')$ in (3) is a bootstrapped observation of $Q^{\star}(x,a)$. The sample averages of these observations will converge to $Q^{\star}(x,a)$ for all x, a if each possible state transition is observed an infinite number of times. Q-learning is an on-line algorithm which uses these observations to stochastically update its estimate of $Q^{\star}(x,a)$.

The update in (3) is known as tabular Q-learning, since a Q-value for each state-action pair is stored in a table and updated each time that pair is visited by the agent. For problems with large or continuous state-action spaces there are so many

states (state-actions), that there is not enough experience to provide good estimates of the Q-functions in the table. In this case, a function approximator which maps state-actions to Q-functions is learned from the data. However, theoretical guarantees for Q-learning in the general case, in which no assumptions are made about the sampling distribution of the observed state-action pairs are limited. The majority of work in this area has focused on the linear case, where $Q_\theta(x,a) = \theta^T\phi(x,a)$, and θ is a d dimensional parameter vector and $\phi(x,a)$ is the feature vector representation of the state-action pair [1]. For practical problems, nonlinear function approximators such as neural networks can be employed to learn the value function, however these methods can be quite unreliable. A tradeoff between the reliability of tabular methods and the interpolation of function approximation is sought in this paper.

Intimately related to the problem of mapping state-actions to value functions is the problem of selecting the feature mapping ϕ. This vector represents the concatenation of different variables corresponding to features extracted from the environment. While many of these feature variables may be relevant to the decision process, some of them may be extraneous, whereas others may best be used in functional combinations. Feature selection consists of including, excluding, weighting, or combining the original features to make a new set of features. Thus, feature extraction modifies the original state space such that it is suitable for use by reinforcement learning algorithms. For instance, in video games, the original state can be represented as the pixels of the screen. There are far too many states in this state space, so the extraneous information in the form of the background or overly complicated sprites would need to be removed for RL to be effective. Success has been had in reducing the dimensionality of the state space using deep networks [17].

III. Nearest Neighbor Interpolation

The initial stages of RL can be very difficult, as actions must be made based on poorly estimated quantities. For tabular Q-learning, random actions must be made when there is no prior experience for that state. Function approximation methods are capable of interpolating Q-values when new states are encountered, but this may take some time before producing reliable quantities. To mitigate these effects, we propose Q-learning using the Q-values from nearest neighbor states to serve as surrogates for action selection and as targets for Q-learning updates when data is unavailable for new states. The algorithm is a variant of tabular Q-learning algorithm described in Table I.

Nearest neighbor regression is a simple method for interpolating the value of a function given a set of input and output examples. Consider the set of points $\{x_k, f(x_k)\}_{k=1}^N$, where $x_k \in \mathcal{X}$ and $f : \mathcal{X} \to \mathcal{Y}$. For any $x \in \mathcal{X}$ we can approximate the value of $f(x)$ as $f(x_n)$ where $x_n = \arg\min_{x_k} d(x, x_k)$ and $d(\cdot,\cdot)$ is a distance function.

Suppose states $\{x_k\}_{k=1}^N$ have been visited, thus there exist table entries for the Q-values of these states at the actions that were taken in them. For step 6 of Table I, if an entry for x in $Q(x,a)$ does not exist, typically an action is selected randomly. Using nearest neighbor interpolation, we can instead use the Q-values of the nearest recorded state. This helps guide actions early in learning so that when only a few points are recorded, action selection is not completely random. Table II shows a modified step 6 to reflect this modification.

TABLE I
Q-Learning With ϵ-Greedy Exploration

1)	Initialize parameters γ, α, ϵ
2)	Repeat:
3)	$\epsilon_0 \leftarrow$ random number between 0 and 1
4)	$x \leftarrow$ current state
5)	if $\epsilon_0 > \epsilon$ then:
6)	$a \leftarrow \arg\max_{a'} Q(x,a')$
7)	else:
8)	$a \leftarrow$ random action
9)	Take action a, observe state transition
10)	$x' \leftarrow$ new state
11)	$r \leftarrow$ reward associated with state transition
12)	$\delta \leftarrow r + \gamma \max_{a'} Q(x',a') - Q(x,a)$
13)	$Q(x,a) \leftarrow Q(x,a) + \alpha \cdot \delta$

TABLE II
Q-learning Variant With Nearest Neighbor Action Selection

6a)	if $\max_a Q(x,a)$ exists, then:
6b)	$a \leftarrow \arg\max_{a'} Q(x,a')$
6c)	else:
6d)	$x_0 \leftarrow$ nearest neighbor to x
6e)	$a \leftarrow \arg\max_{a'} Q(x_0,a')$

Furthermore Q-values can be updated using nearest neighbor interpolation. That is, (3) can be modified so that if X_{t+1} has never been visited, the nearest neighbor to X_{t+1} can be used. Table III gives an updated Q-learning algorithm which includes using nearest neighbor for both action selection and update.

Our approach relies on the distance function d to provide a measure of similarity between states. Typically, nearest

TABLE III
Q-LEARNING USING NN WITH ϵ-GREEDY EXPLORATION

1)	Initialize parameters γ, α, ϵ
2)	Repeat:
3)	$\epsilon_0 \leftarrow$ random number between 0 and 1
4)	$x \leftarrow$ current state
5)	if $\epsilon_0 > \epsilon$ then:
6a)	if $\max_a Q(x, a)$ exists, then:
6b)	$a \leftarrow \arg\max_{a'} Q(x, a')$
6c)	else:
6d)	$x_0 \leftarrow$ nearest neighbor to x
6e)	$a \leftarrow \arg\max_{a'} Q(x_0, a')$
7)	else:
8)	$a \leftarrow$ random action
9)	Take action a, observe state transition
10)	$x' \leftarrow$ new state
11)	$r \leftarrow$ reward associated with state transition
12a)	if $\max_a Q(x', a)$ exists, then:
12b)	$\delta \leftarrow r + \gamma \max_{a'} Q(x', a') - Q(x, a)$
12c)	else:
12d)	$x'_1 \leftarrow$ nearest neighbor to x'
12e)	$\delta \leftarrow r + \gamma \max_{a'} Q(x'_1, a') - Q(x, a)$
13)	$Q(x, a) \leftarrow Q(x, a) + \alpha \cdot \delta$

neighbor is performed using the Euclidean metric, i.e., if x and x' are feature vectors, and x_i represents the ith element of x, then $d(x, x') = \sqrt{\sum_i (x_i - x'_i)^2}$. However, not all features of the state are equally important, and if less important features dominate the Euclidean distance, then neighboring states may not be similar relative to the task at hand. It is therefore important that we use *metric learning* to learn a weighted Euclidean metric. That is, we learn a weight vector w, such that $d_w(x, x') = \sqrt{\sum_i (w_i(x_i - x'_i))^2}$. The weights can be seen as weighting the importance of features.

IV. METRIC LEARNING FOR TASK-DEPENDENT FEATURE WEIGHTING

Instead of viewing feature selection as a preprocessing step, which changes the state vector, we consider changing the underlying distance function in the state space. By changing the underlying metric of the state space, different features can be emphasized, ignored, or combined. Through metric learning [18]–[20], the parameters of a parameterized metric are learned based on additional information relevant to the task.

Since nearest neighbor regression relies on the distance function, we directly modify the distance function to improve the regression as an alternative to feature selection. Good features for regression are ones whose similarity/dissimilarity implies similarity/dissimilarity in the dependent variable. In the case of reinforcement learning, the state representation is the regressor and the Q-value is the dependent variable.

For linear regression the relationship between the independent and dependent variables amounts to correlation, but more generally it can be quantified by statistical dependence [20]. We use a metric learning algorithm named "Centered Alignment Metric Learning (CAML)" [21] that uses a kernel-based statistical dependency measure [22] as the objective function. We briefly restate the key portions of the methods [21] for completeness.

Our objective for metric learning is to maximize the dependence between the state space representation and the Q-function evaluations. Centered kernel alignment is used to evaluate the statistical dependence in terms of kernel functions. Kernel functions are bivariate measures of similarities. The dependence between random variables is measured by how often similarity in one variable corresponds to similarity in the other variable. Somewhat confusingly this can be stated as "how similar are kernel evaluations (a measure of similarity) for the two variables?" Fortunately, this has a clear answer: just like the correlation coefficient for pairs of real valued variables, a natural measure of similarity between these kernel functions is the expected value of their centered and normalized inner product.

Let $\kappa_x(.,.)$ and $\kappa_y(.,.)$ denote the kernel functions defined in $\mathcal{X} \times \mathcal{X}$ and $\mathcal{Y} \times \mathcal{Y}$, respectively. Centered alignment is written as

$$\rho_{\kappa_x,\kappa_y}(x, y) = \frac{\mathrm{E}_{x,y}\mathrm{E}_{x',y'}[\tilde{\kappa}_x(x, x')\tilde{\kappa}_y(y, y')]}{\sqrt{\mathrm{E}_x\mathrm{E}_{x'}[\tilde{\kappa}_x^2(x, x')]\mathrm{E}_y\mathrm{E}_{y'}[\tilde{\kappa}_y^2(y, y')]}} \tag{4}$$

$$\begin{aligned}\tilde{\kappa}_i(z, z') &= \langle \tilde{\phi}_i(z), \tilde{\phi}_i(z') \rangle \\ &= \langle \phi_i(z) - \mathrm{E}_z[\phi_i(z)], \phi_i(z') - \mathrm{E}_{z'}[\phi_i(z')] \rangle \\ &= \kappa_i(z, z') - \mathrm{E}_{z'}[\kappa_i(z, z')] - \mathrm{E}_z[\kappa_i(z, z')] \\ &\quad + \mathrm{E}_{z,z'}[\kappa_i(z, z')]\end{aligned} \tag{5}$$

where x', y', z' are random variables independent and identically distributed to the random variables x, y, z.

The centered alignment $\rho_{\kappa_x,\kappa_y}(x, y)$ is a measure of statistical dependence between x and y. For positive-definite-symmetric kernels, $\rho_{\kappa_x,\kappa_y} \in [0, 1]$[22]. Centered alignment is essentially a normalized version of the Hilbert-Schmidt Information Criterion [23].

Given a dataset $\{x_i, y_i\}_{i=1}^n$, define kernel matrices X and Y, with the elements at i, j being $X_{ij} = \kappa_x(x_i, x_j)$ and $Y_{ij} =$

$\kappa_y(y_i, y_j)$. An empirical estimate of the centered alignment can be computed directly from these kernel matrices as

$$\hat{\rho}(X,Y) = \frac{\langle \tilde{X}, \tilde{Y} \rangle}{\sqrt{\langle \tilde{X}, \tilde{X} \rangle \langle \tilde{Y}, \tilde{Y} \rangle}} = \frac{\langle \tilde{X}, \tilde{Y} \rangle}{\| \tilde{X} \|_2 \| \tilde{Y} \|_2} \quad (6)$$

where $\tilde{X}$ and $\tilde{Y}$ are the *centered* kernel matrices. The centered kernel is computed as

$$\tilde{X}_{ij} = X_{ij} - \frac{1}{n}\sum_{a=1}^{n} X_{aj} - \frac{1}{n}\sum_{b=1}^{n} X_{ib} + \frac{1}{n^2}\sum_{a=1}^{n}\sum_{b=1}^{n} X_{ab}. \quad (7)$$

Using matrix multiplication $\tilde{X} = HXH$ where $H = I - 1/n\mathbf{1}\mathbf{1}^T$ is the empirical centering matrix, I is the $n \times n$ identity matrix, and $\mathbf{1}$ is a vector of ones. The computational complexity of the centered alignment between two $n \times n$ kernel matrices is $\mathcal{O}(n^2)$.

We now consider centered alignment as an objective for explicitly optimizing the coefficients of the weighted Euclidean distance on the space of the independent variable x. If we use the Gaussian kernel, the weights w are present in the argument of the exponential

$$\begin{aligned} k_w(x, x') &= \exp(-d_w(x, x')^2) \\ &= \exp\left(-\sum_i w_i (x_i - x'_i)^2\right). \end{aligned}$$

Any kernel size is absorbed into the weights, so an additional parameter is unnecessary. The unweighted Gaussian kernel is used on the dependent variable

$$k_y(y, y') = \exp\left(-\frac{1}{2\sigma^2} d(y, y')^2\right)$$

as we are not learning a weighting on this space.

As an optimization technique, we use stochastic gradient ascent over small batches taken randomly from the set of available examples. The optimization of the centered alignment over the weights can be stated as

$$\underset{\mathbf{w} \geq 0}{\text{maximize}} \log(\rho_{\kappa_w, \kappa_y}(x, y)) \quad (8)$$

where we take the logarithm of ρ to simplify calculation of the gradient. Taking the logarithm does not change the global optima, and using the logarithm of ρ as the objective resulted in better performance empirically.

When the empirical estimate of centered alignment is substituted the explicit objective function is

$$\begin{aligned} f(w) &= \log(\hat{\rho}(X_w, Y)) \\ &= \log(\mathrm{tr}(X_w H Y H)) - \log(\sqrt{\mathrm{tr}(X_w H X_w H)}) \\ &\quad - \log(\mathrm{tr}(Y H Y H)) \end{aligned} \quad (9)$$

where X_w is the kernel matrix of the independent variables (with the dependence on the weighting present in the kernel k_w made explicit) and Y is the kernel matrix of the dependent variables.

For Q-learning with metric learning, we find a weighting on the state variables to maximize the dependence between the weighted state and the Q-values for those states, under the same action. In other words, two states are similar if performing the same action leads to similar outcomes. This requires a set of visited states and their approximate Q-values, acquired from the initial stages of Q-learning without any feature weighting. Consider the subset $\{x_i, \hat{Q}(x_i, a)\}_{i=1}^{n_a}$ of state-value pairs for a specific action, a. Let $f_a(w)$ be the log of the empirical estimate of centered alignment for this set, defined by (9). A new objective function can be constructed, which takes all actions into account:

$$f(w) = \sum_{a \in \mathcal{A}} f_a(w). \quad (10)$$

Finding the weights w which maximize (10), results in an improved feature weighting. Using this weighting, improved Q-values can be estimated, which can then be used to learn a new weighting in an iterative fashion.

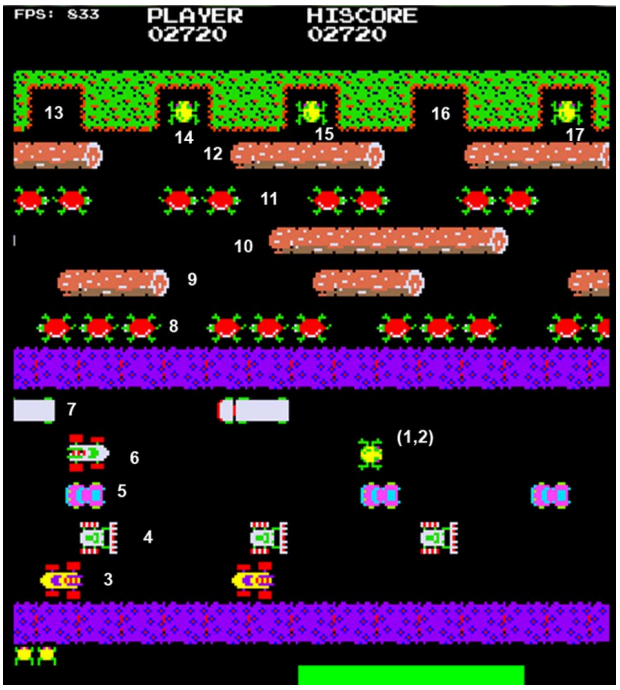


Fig. 3. Visualization of "holistic" feature construction. The first and second features are the x- and y-locations of frog respectively. Features 3–7 are the car sprites x-locations. Features 8–12 are the turtle and log sprite's x-locations. Features 13–17 indicate whether the corresponding home location is empty or filled.

V. Methods

A. Test Bed

We chose the classic arcade game of Frogger as a test bed for our algorithms. Frogger is an interesting learning problem because it contains elements of the classical maze problem, but with moving obstacles and multiple goals. For a screenshot and description of Frogger, refer to Fig. 1.

We use two different feature sets for the state vector of RL. The first feature set, as seen in Fig. 3 will be referred to as "holistic" features, as they are selected globally from the game map without reference to the frog. The variables included in this feature set include the frog's x- and y- positions, the sprites' x-positions, and the status of each of the home positions.

Fig. 4. Visualization of "local" feature construction. Features 1–4 and 6–9 indicate whether a sprite is in the corresponding square-shaped box. Feature 5 is the y-location of the frog. Not pictured here are features 10–14, which (like the holistic feature construction method) indicate whether the home locations are empty or filled.

To save memory, the location of only one sprite per row was used. The location of each sprite in a row is a deterministic function of any one of the sprites in that row. That is, the sprite positions in a row are always fixed in relation to each other. Including the location of more than one sprite per row is redundant information. Since each frog jump is 32 pixels, we discretized the x- and y- position information to multiples of 32.

The second feature set, which we will refer to as "local" features, is visualized in Fig. 4. A set of 32×32 pixel boxes surround the frog. If a sprite is in a box, either fully or partially, the corresponding feature is set to one; otherwise it is set to zero. The fifth feature gives the frog's y-location. Furthermore, like the holistic features, the local feature set keeps track of which home locations are empty or filled.

We use the following reward structure: $+1$ for the frog reaching a home, -1 for the frog dying, and $+5$ for completing a game by reaching all five homes. Furthermore, to speed up learning, a small positive reward of $+0.1$ was given for reaching the midpoint and a small negative reward of -0.1 was given for moving down from the midpoint.

For each of the experiments we use the following terminology: We call each time the frog either reaches a home or dies an "episode," since in either case the agent has reached a terminal state. Each time the player or machine maneuvers five frogs into each of the homes, we say a game has been completed. Each experiment is started with a set number of episodes. After a game is completed the game resets to its initial configuration and learning continues.

B. Results

We evaluate five variants of the Q-learning algorithm based on the speed of learning a policy. For each experiment, we set the discount factor γ to 0.9, the learning rate α to 0.1, and the exploration parameter ϵ to 0.1. These values were chosen based on experimentation and knowledge from previous Q-learning applications. The first variant is the classical tabular Q-learning algorithm. The next two, which we label "NN—No weights" and "NN—Weighted" use nearest neighbor action selection as shown in Table II. The final two use nearest neighbor action selection as well as the nearest neighbor update as shown in Table III. The *weighted* variants weight features according to weights as determined by CAML. The *unweighted* variants simply weight each feature equally with a weighting of one.

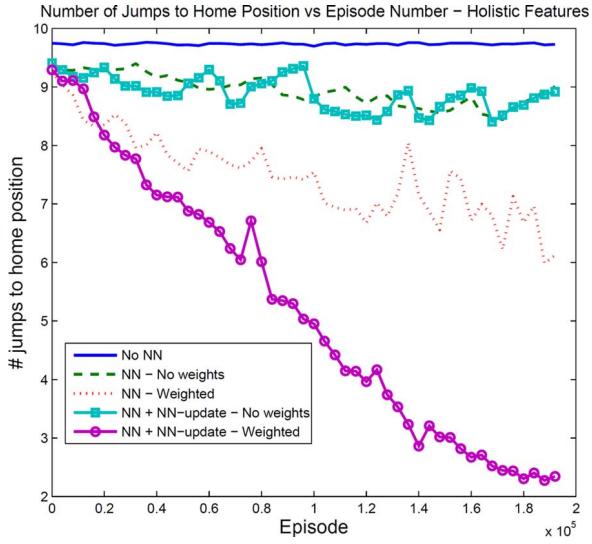


Fig. 5. (Holistic features) Plot of the minimum number of jumps to home for each episode while learning, convolved with a 5000-length moving average filter, for 5 variants of Q-learning and feature weighting. This measures how close the frog made it to its home before dying in each episode. The low-pass filtering was performed because of extremely large episode-to-episode fluctuations in the data. Each episode corresponds to one frog life.

To determine the weights for each feature we first ran the classical Q-learning algorithm to obtain a table of states and their corresponding Q-function estimates. We then ran the CAML algorithm on this data to learn the weighting.

C. Holistic Features

We ran each Q-learning variant with holistic features for 200 000 episodes. Fig. 5 shows the minimum number of jumps away from the home goals the frog reached for each episode, smoothed with a length-5000 moving average filter for clarity. This number changes radically between episodes in part due to the fact that a random action is taken 10% of the time ($\epsilon = 0.1$). It is apparent that the standard tabular Q-learning algorithm fails to learn an improved policy. This occurs because the state space is so large that states are rarely visited more than once, and as a result most actions must be random ones. The unweighted nearest neighbor variants perform better because action can be selected based on the experience of nearby states even when the current state is new.

Fig. 7 shows weights learned for each feature. It should be noted that the frog x- and y- locations are weighted higher than the rest of the features. Weighting these two features makes intuitive sense, because the configuration of the car sprites is irrelevant if the frog is in a completely different position. The "home" features are given the least weight, implying that the optimal action for most states is not dependent on which combination of homes are filled. For example, the moving upwards is a good starting move regardless of where the frog needs to be in the distant future. This weighting provided by metric learning is similar to one that was hand-selected for good performance experimentally.

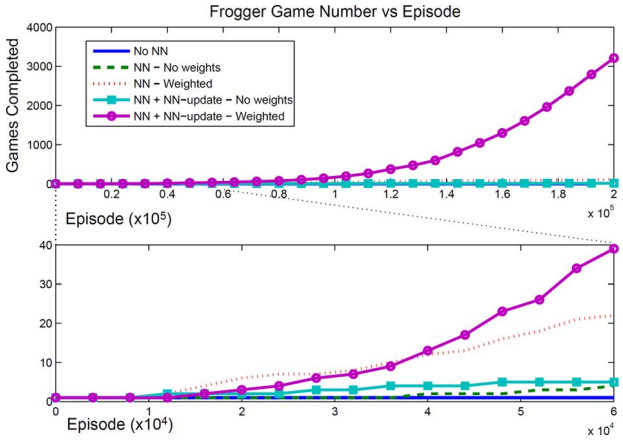


Fig. 6. (Holistic Features) Cumulative number of games won per episode while learning for five variants of Q-learning and feature-weighting. Each episode corresponds to one frog life. The top subplot shows learning over 200 000 episodes. The bottom subplot shows the same learning curves magnified over the first 60 000 episodes.

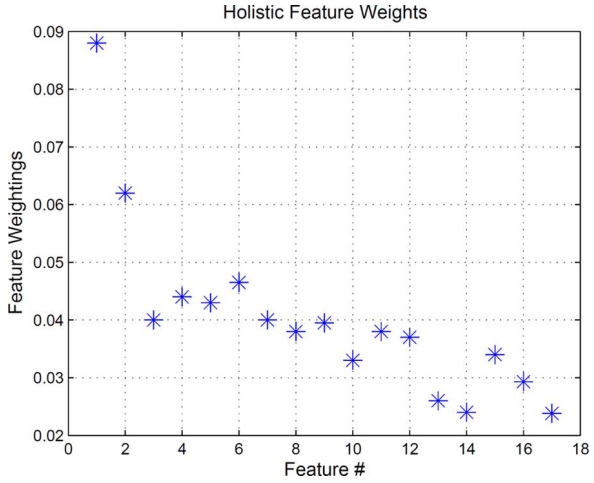


Fig. 7. (Holistic Features) Feature weightings learned using state and Q-value data and CAML. The first and second features are the x- and y- locations of frog respectively. Features 3–7 are the car sprites x-locations. Features 8–12 are the turtle and log sprite's x-locations. Features 13–17 are indicate whether the corresponding home location is empty or filled.

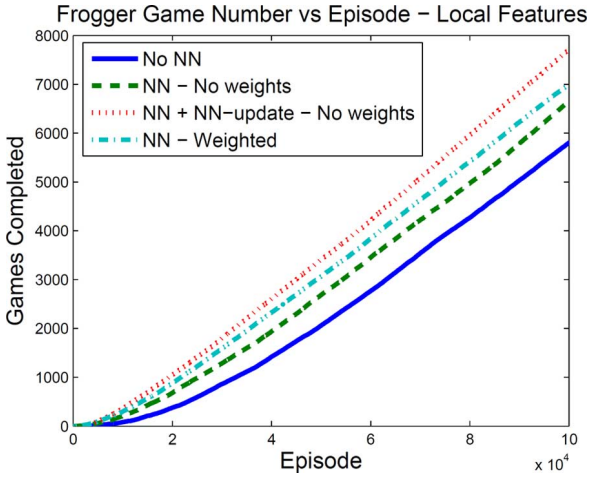


Fig. 8. (Local Features) Cumulative number of games won per episode while learning for four variants of Q-learning and feature-weighting. Each episode corresponds to one frog life.

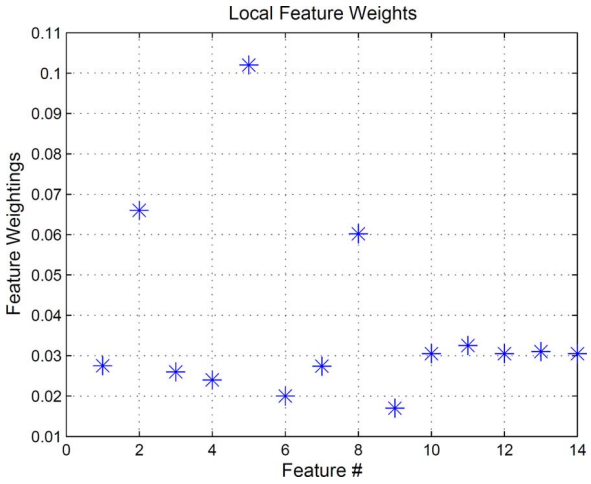


Fig. 9. (Local Features) Feature weightings learned using state and Q-value data and CAML. Features 1–4 and 6–9 indicate whether a sprite is in the corresponding square-shaped box. Feature 5 is the y-location of the frog. Features 10–14 indicate whether the home locations are empty or filled.

Fig. 6 shows the cumulative number of games won versus the number of episodes spent learning. Since using the algorithm shown in Table III is so much more successful than the other variants, we show only the first 60 000 episodes in the bottom view. These results strictly correspond with the results shown in Fig. 5.

Figs. 5 and 6 clearly show that using either or both weighted features and nearest neighbor updates improves the speed of learning. The weights ensure that two states which lead to similar outcomes are themselves similar with respect to our metric, while states that lead to different outcomes are far away. This helps in selecting a suitable action when no "best" action is known. The nearest neighbor update allows previously unvisited states to quickly take on the Q-values of similar states.

D. Local Features

We ran each Q-learning variant with local features for 200 000 episodes. Fig. 8 shows the number of games won versus the number of episodes. The difference in results between Q-learning variants for local features is much smaller than for holistic features because the local feature state space is much smaller. Even so, using weighted features or the algorithm presented in Table III boosts learning by a noticeable amount.

We do not show weighted features with the algorithm in Table III because that variant's results are almost identical to the unweighted case. Weighted versus unweighted is much less significant for this feature set because, unlike the holistic feature set, there are no variables that are significantly more important than the others for determining the correct action.

Figs. 9 and 10 give a visualization of feature importance, as learned by CAML, for the local features in the same style as Fig. 4. It can be seen that the most highly weighted feature is the frog's y-position. After that, the positions above and below the frog are assigned the highest weights. Also notice the features to the left of the frog are more highly weighted than the corresponding features on the right. This can likely be attributed to the fact that more sprites are moving from left to right than from right to left.

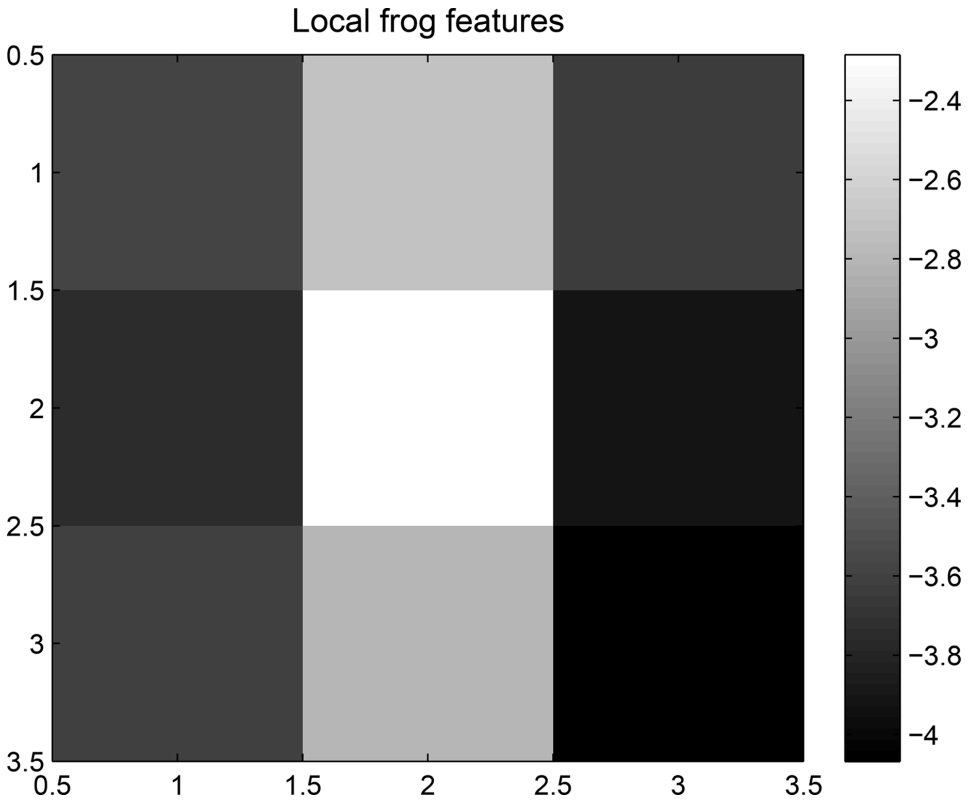


Fig. 10. Visualization of the weightings for local features.

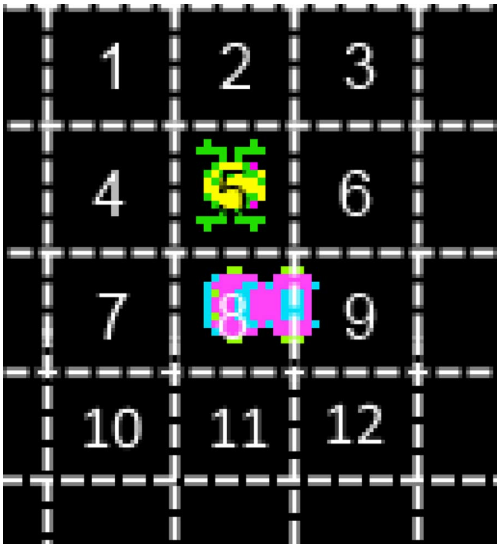


Fig. 11. Visualization of "local" feature construction with irrelevant features. Features 1–4 and 6–12 indicate whether a sprite is in the corresponding square-shaped box. Feature 5 is the y-location of the frog. Features 10–12 are less relevant in determining the action the agent needs to take. Not pictured here are features 13–17, which indicate whether the home locations are empty or filled.

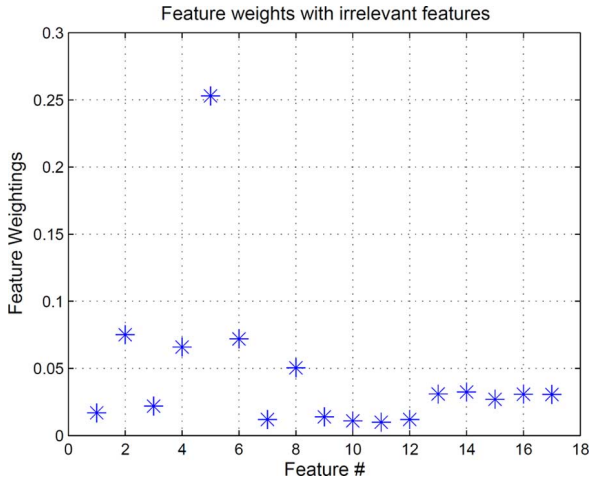


Fig. 12. (Local + Irrelevant Features) Feature weightings learned using state and Q-value data and CAML. Features 1–4 and 6–12 indicate whether a sprite is in the corresponding square-shaped box. Feature 5 is the y-location of the frog. Features 13–17 indicate whether the home locations are empty or filled.

E. Irrelevant Features

Comparing the two feature sets, it is clear that using local features results in faster learning. This is because the local feature set contains a relatively small number of states which consists of information required to select a good policy. Contrariwise, the holistic feature set contains a huge amount of information irrelevant to the agent's goals. For example (Fig. 3), the location of a log at the top of the screen is not very important to the agent controlling the frog at the bottom of the screen. CAML is unable to completely collapse any dimensions from the holistic feature set because each feature becomes important at some point in the frog's trajectory. However, if we add features truly (or almost) irrelevant to the agent's goals, CAML will ensure these features play a small or no part in action selection and value updates. Fig. 11 gives a visualization of a modified local feature set. Three additional features were added which indicate whether a sprite is present two jumps below the

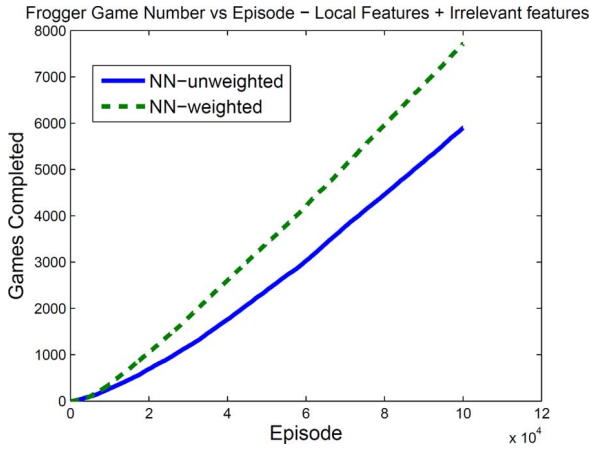


Fig. 13. (Local + Irrelevant Features) Cumulative number of games won per episode while learning for four variants of Q-learning and feature-weighting. Each episode corresponds to one frog life.

frog. After 10 000 iterations, we computed feature weights using CAML (shown in Fig. 12). The "irrelevant" features 10–12 have a low weighting, along with features 1,3,7,9. Fig. 13 shows the number of games won versus the number of episodes using the nearest neighbor update for weighted and unweighted features. Clearly, the weighted features helped speed up learning.

VI. Conclusion

We have shown that using nearest neighbor states to bootstrap value function updates in Q-learning can speed up convergence. This approach is particularly useful for game learning tasks, where the state space is very large, but discrete. Standard tabular Q-learning fails in this scenario because states are rarely revisited, and therefore previous experience cannot be exploited. With a Euclidean metric used to find the nearest neighbor, the features of the state must be selected very carefully, since even features that are uncorrelated with the game task will be weighted with equal importance. The features can instead be weighted such that the state representation and Q-values are highly dependent. With this weighted metric, neighboring states are also similar with respect to the task (reflected in the Q-value). With this approach, state features can be selected more easily, since irrelevant ones will be deemphasized by the metric learning. Our method was tested on the classic arcade game Frogger. Two feature sets were used. With a local feature set consisting of the sprites in the immediate vicinity of the frog, nearest neighbor based Q-learning successfully learned how to navigate the frog. When irrelevant features were added to the local feature set, metric learning learned to eliminate or decrease the importance of the irrelevant features when making nearest neighbor decisions and updates. With a holistic feature set consisting of sprites throughout the entire game map, metric learning greatly improved the learning performance. Our approach for weighting features was used in combination with a simple nearest neighbor approximation. The authors predict similar success when used with RL methods such as kernel-based RL [8] and RL with Case Based Reasoning (CBR) [10].

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. Cambridge, U.K.: Cambridge Univ Press, 1998, vol. 1.
[2] D. P. Bertsekas and J. N. Tsitsiklis, "Neuro-dynamic programming (optimization and neural computation series, 3)," *Athena Scientif.*, vol. 7, pp. 15–23, 1996.
[3] G. Tesauro, "Td-gammon, a self-teaching backgammon program, achieves master-level play," *Neural Comput.*, vol. 6, no. 2, pp. 215–219, 1994.
[4] S. Thrun, "Learning to play the game of chess," in *Adv. Neural Inf. Process. Syst. (NIPS) 7*, G. Tesauro, D. Touretzky, and T. Leen, Eds. Cambridge, MA, USA: MIT Press, 1995.
[5] J. Baxter, A. Tridgell, and L. Weaver, "Learning to play chess using temporal differences," *Mach. Learn.*, vol. 40, no. 3, pp. 243–263, 2000.
[6] R. Bellman, *Dynamic Programming*, 1st ed ed. Princeton, NJ, USA: Princeton Univ. Press, 1957.
[7] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *J. Mach. Learn. Res.*, vol. 3, pp. 1157–1182, 2003.
[8] D. Ormoneit and S. Sen, "Kernel-based reinforcement learning," *Mach. Learn.*, vol. 49, no. 2-3, pp. 161–178, 2002.
[9] G. J. Gordon, "Stable function approximation in dynamic programming, DTIC Document," Tech. Rep. CMU-CS-95-103, 1995.
[10] T. Gabel and M. Riedmiller, "CBR for state value function approximation in reinforcement learning," in *Case-based Reasoning for State Value Function Approximation in Reinforcement Learning*. New York, NY, USA: Springer-Verlag, 2005, pp. 206–221.
[11] X. Huang and J. Weng, "Covert perceptual capability develop," presented at the Int. Work. Epigenetic Robot., 2005.
[12] L. C. Cobo, P. Zang, C. L. Isbell, and A. L. Thomaz, "Automatic state abstraction from demonstration," in *Proc.-IJCAI Int. Joint Conf. on Artif. Intell.*, 2011, vol. 22, pp. 1243–1243.
[13] S. Wintermute, "Using imagery to simplify perceptual abstraction in reinforcement learning agents," in *Proc. AAAI Conf. on Artif. Intell.*, Ann Arbor, MI, USA, 2010, vol. 1001, pp. 48 109–2121.
[14] E. Kriminger, A. Brockmeier, L. Sanchez-Giraldo, and J. Principe, "Metric learning for invariant feature generation in reinforcement learning," in *Proc. Multidisciplin. Conf. Reinforce. Learn. Decision Making*, 2013.
[15] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D., King's College, Cambridge, U.K., 1989.
[16] H. Robbins and S. Monro, "A stochastic approximation method," *Ann. Math. Statist.*, pp. 400–407, 1951.
[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep reinforcement learning," presented at the NIPS 2013 Deep Learning Workshop, 2013 [Online]. Available: Arxiv Preprint Arxiv:1312.5602
[18] D. G. Lowe, "Similarity metric learning for a variable-kernel classifier," *Neural Comput.*, vol. 7, no. 1, pp. 72–85, 1995.
[19] E. P. Xing, A. Y. Ng, M. I. Jordan, and S. Russell, "Distance metric learning with application to clustering with side-information," in *Advances in Neural Inform. Processing Syst. 15*, S. T. S. Becker and K. Obermayer, Eds. Cambridge, MA: MIT Press, 2003, pp. 505–512.
[20] K. Fukumizu, F. R. Bach, and M. I. Jordan, "Dimensionality reduction for supervised learning with reproducing kernel Hilbert spaces," *J. Mach. Learn. Res.*, vol. 5, pp. 73–99, 2004.
[21] A. J. Brockmeier, J. S. Choi, E. G. Kriminger, J. T. Francis, and J. C. Principe, "Neural decoding with kernel-based metric learning," *Neural Comput.*, 2014.
[22] C. Cortes, M. Mohri, and A. Rostamizadeh, "Algorithms for learning kernels based on centered alignment," *J. Mach. Learn. Res.*, vol. 13, pp. 795–828, 2012.
[23] A. Gretton, O. Bousquet, A. Smola, and B. Schölkopf, "Measuring statistical dependence with Hilbert-Schmidt norms," in *Algorithmic Learning Theory*. New York, NY, USA: Springer-Verlag, 2005, pp. 63–77.

Matthew S. Emigh received the B.S. and M.S. degrees in electrical engineering from the University of Florida, Gainesville, FL, USA, in 2010 and 2012, respectively. He is currently working toward the Ph.D. degree in electrical and computer engineering.

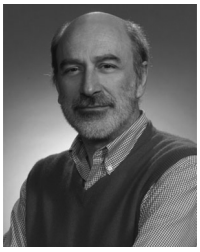
His research interests include machine learning, adaptive signal processing, and information-theoretic learning.

Evan G. Kriminger received the M.S. degree in electrical engineering from the University of Florida, Gainesville, FL, USA, in 2010, and the B.S. degree in engineering science from the University of Miami, Coral Gables, FL, USA, in 2009. He is currently working toward the Ph.D. degree in electrical engineering at the University of Florida.

As a member of the Computational NeuroEngineering Laboratory (CNEL), he has done work on time series analysis, reinforcement learning, and active learning. His dissertation work focuses on active learning methods to discover the true number of classes in open-set classification problems, such as automatic target recognition.

Austin J. Brockmeier (S'05–M'14) received the Ph.D. degree from University of Florida, Gainesville, Fl, USA, in 2014. He received the B.S. degree in computer engineering from the University of Nebraska-Lincoln, Omaha, NE, USA, in 2009.

From 2010 to 2014, he was a Graduate Research Assistant with the Computation NeuroEngineering Laboratory. Currently, he is a Postdoctoral Research Associate in Electrical Engineering and Electronics, University of Liverpool, Liverpool, U.K. His research is the application of signal processing and machine learning to text mining and biomedical signals, specifically neural signal analysis and brain-machine interfaces. He is interested in math, science, and engineering education.



José C. Príncipe (M'83–SM'90–F'00) received the Ph.D. degree in electrical and computer engineering from the University of Florida, Gainesville, FL, USA, in 1979.

He is a Distinguished Professor of Electrical and Computer Engineering with the University of Florida, Gainesville. He is a BellSouth Professor and the Founding Director of the Computational Neuro-Engineering Laboratory, University of Florida. His current research interests are centered in advanced signal processing and machine learning, brain machine interfaces, and the modeling and applications of cognitive systems.

Panos M. Pardalos received the Ph.D. degree in computer and information sciences from the University of Minnesota, Minneapolis, MN, USA.

He is a Distinguished Professor of Industrial and Systems Engineering at the University of Florida, Gainesville. He is the director of the Center for Applied Optimization.

Dr. Pardalos is the Editor-in-Chief of the *Journal of Global Optimization*, and of the journals *Optimization Letters*, *Computational Management Science*, and *Energy Systems*.