A Universal Robot Control System using Reinforcement Learning with Limited Feedback

Joshua Gruenstein Michael Truell

May 8, 2016

Abstract

A robot control system was developed that could be taught tasks through reinforcement learning. The system, nicknamed "Fido", was designed to be universal regardless of inputs and outputs, robot kinematics, and processing capability. In addition, Fido was built to learn with limited feedback, allowing humans to train Fido in a minimal amount of time. This was achieved through the training of artificial neural networks with a wire-fitted interpolator following the Q-learning reinforcement learning algorithm and an intelligent action selection policy that utilizes a probabilistic approach to exploration. Functionality was first tested and evaluated in simulation. Next, hardware implementations of differing kinematics, sensors, and central processors were constructed. Fido successfully converged on all given tasks in simulation and in hardware within very few reward iterations while maintaining impressively low latency, demonstrating its potential as a comprehensive robot control system.

1 Intro

The most prevalent control system used in mobile robotics is a procedurally programmed expert system (Biggs & MacDonald, 2003). Such systems use linear conditional logic in order to emulate a desired behavior. However, such systems are limited in numerous respects. First, they can only perform the specific task for which they were programmed to accomplish; the entire software must be rewritten in order to change the target task. Second, they rely on a knowledge of the inputs and outputs to the robot (such as sensors and motor control) in order to function. The purpose of Fido was to solve both of these problems, allowing a universal general control system for robots that can be trained on tasks using reinforcement learning.

We chose to approach this problem with artificial neural networks; function appropriators modeled after nature with the capability to take in a large number of inputs to produce an output. Neural networks are commonly used to solve tasks that are challenging using traditional rule-based programming, making them perfect for our task. The control system was named Fido for the name's connotations to training an intelligent organism.

2 Background

The human brain is composed of billions of neurons, interconnected electrically excitable cells that form the basis of intelligence. Each neuron has dendrites that receive electrical signals from other neurons. If the sum of these electrical signals is greater than a certain threshold value, the neuron fires, propagating a voltage down its axon and out of its synapses. These synapses can be considered the output of the neuron and are themselves connected to the dendrites of other neurons. The interconnections of these neurons form a massive network, where a huge number of inputs are processed in parallel to a set of outputs. The purpose of artificial neural networks is to simulate the mathematical properties of these neurons in order to approximate complex functions.

2.1 Single Artificial Neuron

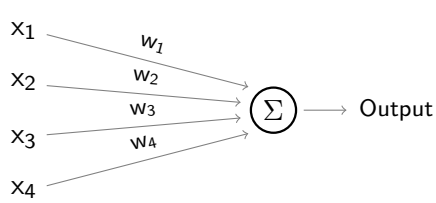

Figure 1: Single Neuron Diagram

An artificial neuron is simply a mathematical model of a biological neuron, and therefore its functionality is very similar. Each artificial neuron has multiple inputs, each one the output of a different neuron. In addition, each neuron has a weight for each input, a bias term, an activation function, and one output. The output of one neuron becomes the input

of other connected neurons. The output of a neuron is expressed mathematically for n inputs x and weights w, a bias term b, and activation function $O(x)$:

$$\text{activation} = \sum_{i=0}^{n} x_i w_i + b$$
$$\text{output} = O(\text{activation})\,.$$

A weight is a positive or negative number that governs the impact of of its respective input on the neuron's single output. The bias term is a number added to the summation of weights and inputs, allowing us to make affine transformations on the domain of the activation function.

In biologically inspired neurons, the activation function is a binary step function. If the activation function's input is less than a certain threshold, the output is zero. If it is greater than the threshold, the output is one. However, a binary output can be somewhat limiting for many applications of neural networks. Many of Fido's outputs are gradient rather than binary; as an example, Fido can set the brightness of a light-emitting diode. Because of this, alternate activation functions with gradient outputs are used. One such function is the sigmoid function. In addition to being continuous, the sigmoid function is differentiable, allowing the network to be trained, and nonlinear, allowing the network to model a broader range of functions. The sigmoid function is expressed as such:

$$O(a) = \frac{1}{1 + e^{-\frac{a}{p}}}\,, \tag{1}$$

for each output O, activation a, and constant p. The sigmoid activation function can also be graphed as below.

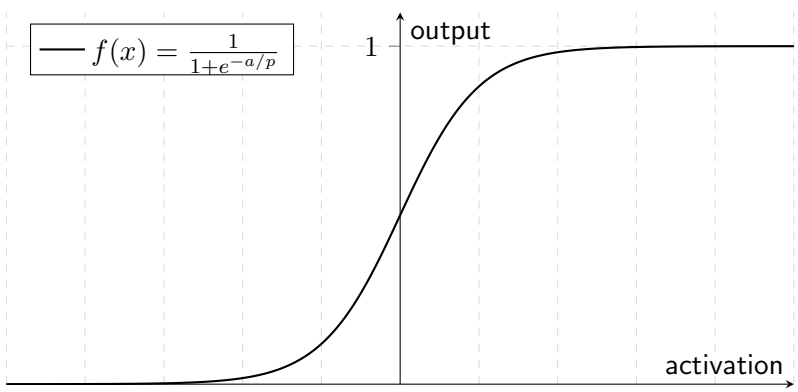

Figure 2: Sigmoid Function Graph

2.2 Feed-forward Neural Network

A traditional method of arranging artificial neurons in a neural network is called a feed-forward network. Neurons are connected as previously described: the output of each neuron

is connected to one of the inputs of another neuron. These neurons are organized into layers, as described in Figure 3.

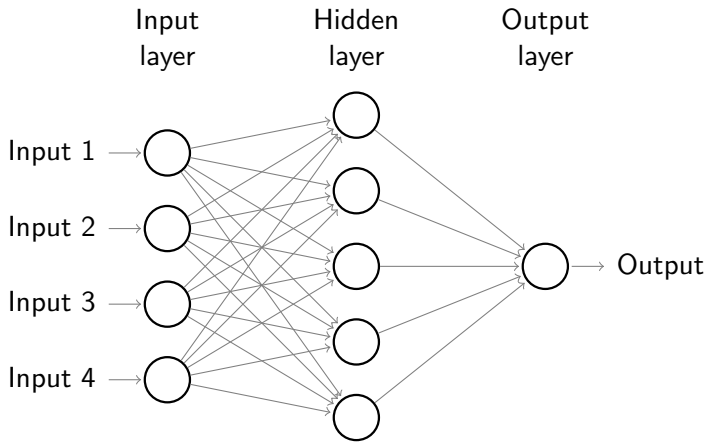

Figure 3: Single Output Feed-forward Network

The output of each neuron in a layer becomes an input of every neuron in the next layer. In this way, the original inputs of the neural network are "fed forward" layer by layer, starting from the first layer (the input layer), passing through any number of middle layers (hidden layers), and ending with the last layer (the output layer). The outputs of the neurons in the output layer are the outputs of the whole network.

Fido is a feed-forward network. Sensor inputs such as light and sound are sent into the input layer. The outputs of output layer neurons are used to change the speed of Fido's motors, the color of Fido's LEDs, and the sound of Fido's buzzers.

2.3 Backpropagation

Supervised learning is the modification of the weights of a neural network's neurons in order to reach a specific output from a specific set of inputs. One such method of adjusting neural networks weights is called back propagation (Werbos, 1974).

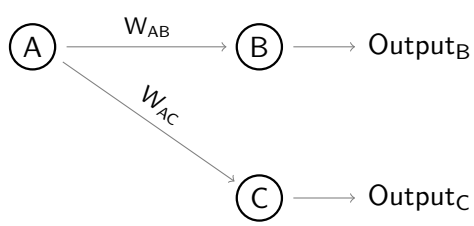

Figure 4: Single Branch in a Back Propagation Neural Network

Neural networks learning through back propagation are trained through example. Example inputs (known as training sets) are linked to a particular target output, forming a training pair. The first step of training is to pass in an arbitrary training set to a neural network with randomly generated weights. The error δ for an output neuron is defined as

the difference between the actual output and the target output for that training set. As an example we will use the single connection described in Figure 4. The neural network in this figure has only an input layer consisting of Neuron A and an output layer consisting of Neurons A and B. W_{AB} and W_{AC} are the weights between the neurons. We first determine the Neuron B error δ_B as $\text{Target}_B - \text{Output}_B$. Next we modify weight W_{AB} as follows, defining W_{AB}^{+} as the newly trained weight:

$$W_{AB}^{+} = W_{AB} + (\delta_B \text{Output}_A)\,. \tag{2}$$

The same approach could be taken for Neuron C. However, this method cannot calculate an improved weight for hidden layer neurons such as Neuron A, as there is no predefined target. Therefore we must calculate the error of A indirectly by back propagating from the two output layer neurons, for which the error is already known:

$$\delta_A = W_{AB}\delta_B + W_{AC}\delta_C\,. \tag{3}$$

Once the error for Neuron A is obtained, its trained weights can be calculated in the same fashion as done for Neuron B. This pattern can be repeated throughout a larger neural network in order to converge upon the network's target output.

2.4 Q-Learning

Reinforcement learning seeks to find the optimal action to be undertaken for a given state through trial and error. In the context of Fido, an action could be the playing of a note or driving straight forwards, while the state could be the amount of light detected by the robot or how near the robot is to another object. Once an action is performed, a reward and a new state are given back to the reinforcement learning algorithm. As actions are performed over time, the reinforcement learning algorithm sharpens its ability to receive reward.

Q-Learning (Watkins, 1989) is a popular reinforcement learning algorithm that works by learning an action-value function Q that takes a state-action pair as an input and outputs the expected utility value of performing that action in that state. This utility value is know as the Q-value. The Q-value is a combination of immediate reward and expected future reward. Every reward iteration the Q-value of an state-action pair is updated as such:

$$Q(s,a) := Q(s,a)(1-\alpha) + \alpha(R + \gamma \max Q(s_{t+1}, a))\,, \tag{4}$$

where a is the action carried out, s is the initial state, R is the reward received, and s_{t+1} is the new state. α is the learning rate of the algorithm. The learning rate determines the rate of convergence by diminishing or amplifying the changes made to the Q-value each reward iteration. γ is the devaluation factor, which determines the weight given to future rewards. A devaluation factor approaching $\gamma = 0$ will force the algorithm to only value immediate reward, while a devaluation factor approaching $\gamma = 1$ will make it focused on high long term reward. A table is used to model the Q function, storing past state-action pairs and each pair's respective Q-value.

3 Learning

3.1 Wire-Fitted Q-Learning

The Q-Learning algorithm had to be immediately modified to be able to work in continuous state-action spaces for it to be suitable for Fido. Conventional Q-Learning is discrete. No relation is made between states or actions, and every action for each state must be performed individually in a noisy feedback system to determine its Q-value. However, Fido will work in a large, continuous state-action spaces where relations made between (state-action, Q-value) pairs can drastically reduce the number of reward iterations needed for convergence. An example of a task that would benefit from continuity is teaching Fido to adjust the speed of its motors based on the intensity of light that the robot detects. There is an obvious gradient relation between the (state-action, Q-value) pairs in this task and with a limited number of Q-values known, it is possible to correctly model Q.

To accommodate continuous action-spaces, we coupled a wire-fitted moving least squares interpolator with our feed-forward neural network as described in (Gaskett, Wettergreen, & Zelinsky, 1999).

Feed-forward neural networks can generalize between states in Q-Learning problems with discrete actions as described in (Rummery, 1995). To extend this implementation to a continuous action space, our feed-forward neural network outputs discrete "wires" when given a state. Each wire consists of an action with its respective Q-value for the state given to the neural network. These wires may be interpolated to model Q, allowing us to get the Q-value of any action performed in a state given as an input to the network. The interpolator used in Fido is a wire-fitted moving least squares interpolator used in the context of a memory-based learning system (Baird & Klopf, 1993).

The wire-fitting function calculates Q-value of an action $\hat{a}$ for a state s given a set of n actions a each with a respective Q-value q as such:

$$Q(a,s) = \frac{\sum_{i=0}^{n} \frac{q_i}{||\hat{a} - a_i||^2 + c(q_{max} - q_i) + k}}{\sum_{i=0}^{n} \frac{1}{||\hat{a} - a_i||^2 + c(q_{max} - q_i) + k}}, \tag{5}$$

where q_{max} is the greatest Q-value among the set of Q-values q, and k is a small value that avoids division by zero. c is the smoothing factor. The greater the smoothing factor, the smoother the interpolated function.

Figure 5 is an example of interpolation on a set of wires. The graph shows the value of one-dimensional actions plotted against their respective Q-values. The wire-fitting function has few properties that make it especially suited for Fido.

Every update to the Q-value requires that q_{max} is computed and the action that produces q_{max} is needed for common action selection policies. As proved in (Baird & Klopf, 1993), the wire with the greatest Q-value is the interpolation point with the greatest Q-value, therefore q_{max} is the maximum Q-value out of the set of wires given to the wire-fitting function. This makes it extremely computationally cheap to compute Q-value, allowing Fido's latency to stay minimal.

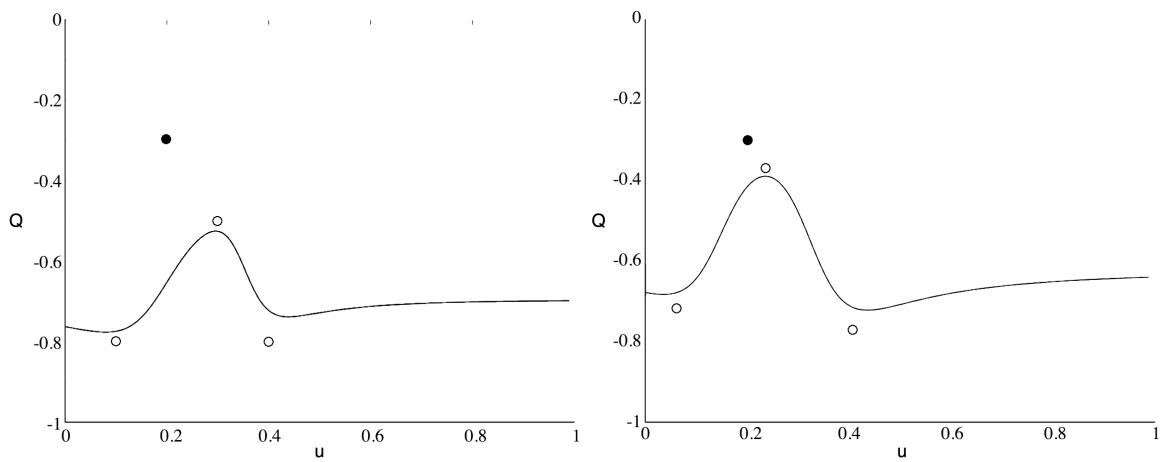

Figure 5: Moving Least Squares Interpolator (adapted from Gaskett, Wettergreen, & Zelinsky, 1999)

The wire-fitting function is derivable. This allows us to update our wires, and therefore our model of Q, using gradient descent. Gradient descent is an optimization algorithm that looks to find the local minimum of a function by modifying each of its parameters, one by one. The update function for a parameter is calculated as such:

$$a = a - \gamma \Delta F(a)\,, \tag{6}$$

where a is the parameter to be updated, γ is the learning rate, and $\Delta F(a)$ is the partial derivative of the function to be minimized F with respect to a. In the case of Fido, once the reward and new state for an action-state pair is received and an updated Q-value $\hat{q}$ is calculated using Equation 4, we are trying to minimize the wire-fitting function's error at predicting $\hat{q}$ when given the wires for Fido's previous state. This error can be calculated as:

$$(\hat{q} - q)^2\,, \tag{7}$$

where q is the old Q-value. Using Equation 7 as our function to be minimized and the partial derivative of the wire-fitting function with respect to each action vector and each q-value, we may compute the partial derivatives of our cost function. Using these, new wires may be calculated for Fido's previous state using gradient descent.

Fido's neural network may be trained is trained to output these wires using the Adadelta training algorithm. Adadelta is a variant of backpropgation that dynamically adjusts its learning rate to an optimum value, allowing for lower latency. This increases the universality of Fido, since backpropagation's learning rate is task specific.

3.2 Uncertainty Value

Fido continuously calculates the "uncertainty" u of its system. Such a value should lower as Fido learns a task and rise if Fido's task changes. This value is useful in adjusting a number hyperparameters that are detailed in the next few sections, such as the exploratory level of

the system. After receiving reward and updating its wires using stochastic gradient descent, Fido calculates its uncertainty value as the means squared error between its original set of wires and its newly updated set of wires. In this way, Fido uncertainty value is proportional to the disparity between the current predictions of its model and new information presented to Fido during the current learning iteration.

3.3 Action Selection

Q-learn requires that actions are selected to be performed. There are a number of approaches to choosing this action, and each has a large affect on the behavior of the learning implementation. The most common approach is to simply pick the action with the best Q-value for the current state. However this strategy stifles exploration of the state-action space, increasing the time of convergence on a task, hurting the retrain-ability of the model, and giving a bias towards an actor's starting policy, which is random. This problem is compounded in the case of Fido due to the large state and action spaces that Fido must explore. Another common method of action selection is to choose each action randomly for a set number of reward iterations, and then to switch to choosing the action with the highest Q-value. This plan improves upon the first by allowing for a period of exploration, but is not suited for Fido. During its lifetime, Fido must have the ability to learn new tasks and be retrained by the operator providing feedback, and so, must continuously explore its state space.

Fido selects actions probabilistically using a Boltzmann or soft-max distribution of probability. The likelihood that an action $\hat{a}$ will be chosen from a set of n actions a for a state s is given as:

$$p(\hat{a}) = \frac{e^{\frac{Q(s,\hat{a})}{T}}}{\sum_{i=0}^{n} e^{\frac{Q(s,a_i)}{T}}}. \tag{8}$$

T is the temperature, or exploration level. As T approaches infinity, a random action is chosen. As T tends toward 0, the best action is chosen. Fido keeps T at a constant value around $T \approx 0.15$ throughout its lifetime to encourage occasional, continued exploration.

T is set proportional to Fido's uncertainty value, so that Fido rapidly explores when it is first initialized or is being retrained. This allows Fido to converge on tasks in fewer reward iterations than if T were set as a constant. As Fido learns a task, T will approach 0. This means that Fido will accurately perform a task when sure of its actions, increasing Fido's average reward over other systems.

3.4 Experience Replay

As a reinforcement learning system proceeded throughout a problem, the effect of former learning iterations decreases with time as the system updates the parameters of its model to lower its error on new learning iterations. This is wasteful since some iterations will present the system with critical information, such as tagging an unlikely action with high reward.

Fido stores each learning iteration's initial and final states, action, and reward, collectively known as an "experience." Every learning iteration, Fido trains its model on a few past experiences, which are sampled probabilistically according to their age. Weight is given to newer experiences. The number of experiences sampled is proportional to the inverse of

Fido's uncertainty value, as past histories will become invalid if the parameters of a problem change. This practice, called "experience replay," lowers the number of learning iterations required for convergence by utilizing all of the information available to Fido.

3.5 Dynamic Model Architecture

The optimal size of a neural network is directly related to the complexity of the function that such a neural network has to model. This requires that Fido dynamically size its network, since a constant neural network architecture would never be able to converge on some tasks. Likewise, Fido must dynamically optimize the number of wires outputted by its neural network, since more complicated tasks require a larger number wires to effectively model their action-reward function.

The most accurate method of model architecture optimization would be a brute force search. When using such a method, all the possible permutations of Fido's model within a range would be generated. Each possible model would be trained on a sample of Fido's experiences for a finite number of epochs. The model with the lowest error on the sample of experiences would be selected as Fido's current model.

However, such a method is extraordinarily latent and since Fido needs to be able to run on low power systems, is infeasible. Stochastic gradient descent would be a natural choice for such a search, but the relationship between Fido's error and its model architecture is not easily derivable.

Instead, Fido performs a simultaneous perturbation stochastic approximation to adjust its model architecture. SPSA is gradient descent method designed for discrete systems that requires only two samples of the cost function to compute its derivative. Its update function is similar to 7 and is given as:

$$u_{n+1} = u_n - a_n \hat{g_n}(u_n) \tag{9}$$

where n is the current epoch; u is our model parameters, the number of wires and neurons; a is a positive sequence of numbers that approach 0 as n approaches 0; and $\hat{g_n}$ is an estimation of the derivative of the cost function. SPSA defines $\hat{g_n}$ as:

$$\hat{g_n}(u_n) = \frac{J(u_n + c_n\Delta_n) - J(u_n - c_n\Delta_n)}{2c_n(\Delta_n)_i}. \tag{10}$$

where Δ_n is a random perturbation vector and J is the cost function. For Fido, J is the error of the the model m_n that is defined by u_n on a sample of experiences after a period of training.

To optimize the sampling of J, Fido creates m_n's neural network by adding and subtracting the Fido's current network since Fido's model at any given time will have a relatively low J. Fido subtracts "unimportant" neurons using a pruning algorithm so as to decrease $J(m_n)$. Our pruning algorithm estimates the effect of pruning weight w_{ij} on J. This estimate using values recorded during backpropagation or Adadelta and is given as:

$$\sum_{b=0}^{B} \frac{\delta E}{\delta w_{ij}} \Delta w_{ij}(b) \frac{w_{ij}^f}{w_{ij}^f - w_{ij}^i} \tag{11}$$

Neurons whose weights have the smallest impact on J should be pruned.

4 Implementation

Fido was implemented in the C++ programming language with minimal use of the standard library. This was to enable cross-platform functionality, lightweight and speedy performance, and the ability to run on microcontrollers if necessary. Class structures were created for neurons and neural networks making extensive use of the "std::vector" sequence container, however to facilitate the goal of minimal standard library reliance a custom vector template class was created. In order to test Fido both a simulation environment and a hardware implementation were constructed and written minimal driver suites for interaction with inputs and outputs.

4.1 Simulation

An asynchronous simulator was created using C++ and the SFML graphics library in order to efficiently test Fido's performance on different types of robots. Multiple inputs were modeled for simulation with outlets for control both by a human operator using sliders and by programmed handlers using a bridge class. Sensors included a microphone, light sensor, infrared light sensor, inertial measurement unit, and three axes of radio receivers allowing measurement from a radio beacon. Multiple tasks were trained using both operator input and autonomous "taskmaster" programs to regulate positive and negative feedback.

Fido's outputs were chosen similarly. A buzzer of varying tone and frequency can play sounds and a multicolor LED can be lit to any red-green-blue color combination. Two motors allow movement using one of two kinematic configurations: differential drive similar to that of a tank, or holonomic control for each axis of movement. Appropriate kinematics for each model including acceleration and friction were implemented with help from (Dudek & Jenkin, 2000).

The black rectangle in the upper right corner of the simulator is the robot, having been moved as part of training. The red dot near the rectangle is a graphical representation of a radio beacon. As adjusting sliders to represent the location of a radio beacon relative to the robot would be impractical, we decided to implement a beacon that could be placed and dragged by right clicking on the simulator. Simulated sensor readings of beacon strength on two axes are gathered using an inverse square law, as applies to radio waves in general. These readings are then displayed in the sliders and can be manually altered as well. The radio beacon can be removed by a human operator by pressing the "p" key in the simulator environment. This was especially helpful in the task of training Fido to follow a radio beacon.

4.2 Hardware

A hardware implementation of Fido was created to test real-life trainability and applicability. As the implementation's "brain" we chose the Intel Edison embedded Linux board. Although Fido was designed to run on microcontrollers without the standard library, a Linux board was chosen to reduce prototyping time. However, segments of Fido's code-base have been tested on a Cortex-M4 microcontroller. The hardware implementation was also given a

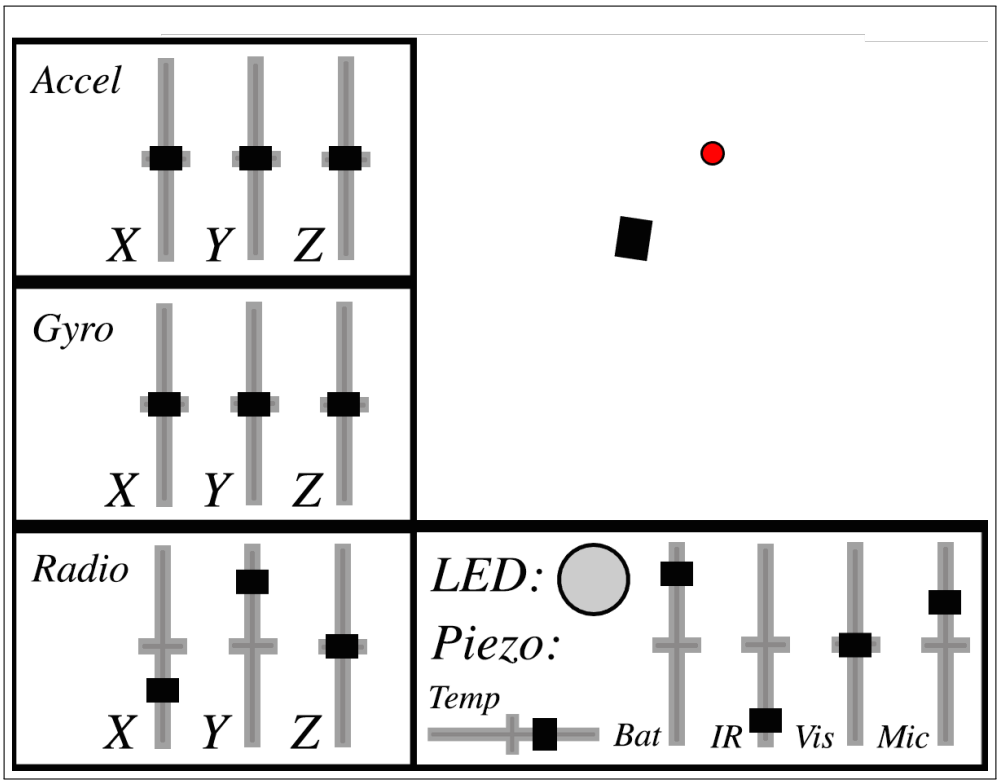

Figure 6: Screenshot of the Fido Simulator Graphical User Interface

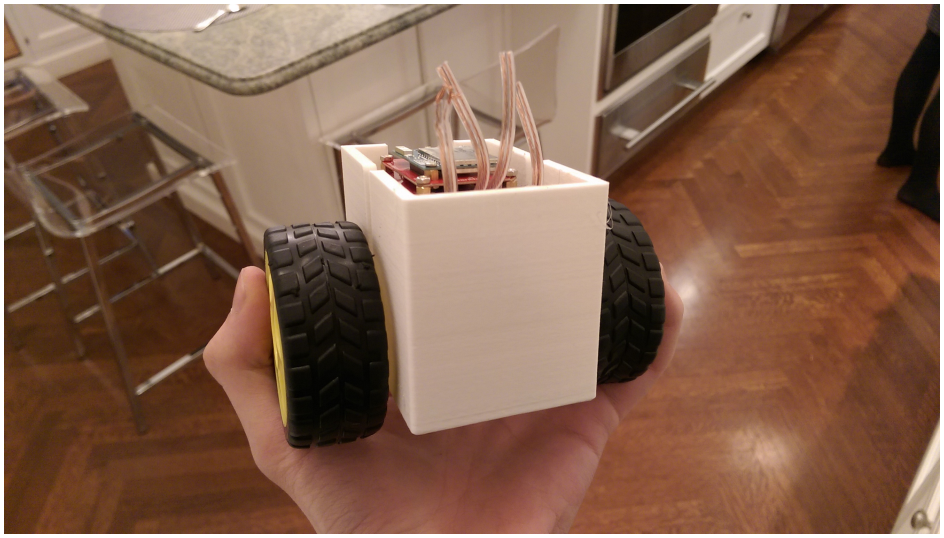

Figure 7: Fido Hardware Implementation with Head-Cap Removed

selection of inputs and outputs: an inertial measurement unit, a light sensor, a microphone, and differential drive. The chassis was modeled in AutoCAD and 3D printed through the Horace Mann School computer science department.

Next, an Android application was created to assist in Fido's training. The application allows an operator to create or select a new neural network to be trained for a specific task, or "trick." The application also allows for the administration of virtual reward. The application communicates with Fido over Bluetooth LE for portable development and demonstration.

5 Results

5.1 Data Collection from Simulation

To test Fido's effectiveness at learning with limited feedback, Fido was first trained on a number of different tasks through our simulator using reward values delegated through software. Data was collected regarding Fido's latency and number of reward iterations needed for convergence.

5.1.1 Simulation Experiments

Fido's first and simplest task, dubbed "Flash," was to set the brightness value of an LED to a value proportional to the amount of light that Fido sensed. Each reward iteration, Fido's neural network was given the intensity of visible light that Fido detected and was asked for the brightness value of Fido's LED. Fido was then given a reward value equal to $1 - |b - v|$ where b was the brightness value of Fido's LED ranging from 0 to 1 and v was the intensity of visible light that Fido detected ranging from 0 to 1. Fido's feed-forward neural network had 3 hidden layers each with 10 neurons and outputted 4 wires to the wire-fitted interpolator. As for all of the tasks, Fido's neural network's input and hidden layers used a sigmoid activation function, while its output layer used a linear activation function that simply outputted its input. The exploration constant in equation 8 was held at 0.2 for the duration of the task.

"Float," Fido's second task, challenged our learning implementation to direct a robot to point. Each time it was told to select an action, Fido specified the robot's vertical and horizontal velocity between +10 and -10 pixels. This emulates a holonomic drive systems, where motor outputs directly correlate to movement on the x and y axes. At the start of each trial, Fido and the point were placed randomly on a boundless plane within 768 pixels of one another. Fido was fed the ratio of its x displacement to its y displacement from its target point as the state. Every fourth action that Fido made was chosen as a reward iteration, and Fido was given a reward value corresponding to its last action. This reward value was the difference between Fido's distance away from the target point before performing the action and Fido's distance from the target point after performing the action. Fido completed each trial when it was within 70 pixels of the point. Fido's feed-forward neural network had 3 hidden layers each with 10 neurons and outputted 3 wires to the wire-fitted interpolator. The exploration constant in equation 8 was held at 0.15.

Fido's next task, nicknamed "Drive," required that it direct a robot to point by controlling the motors of a differential drive system. At the start of each trial, Fido and the point were

placed randomly on a boundless plane within 768 pixels of one another. Fido was fed the ratio of its x displacement to its y displacement from its target point as well as its rotation. Every fourth action that Fido made was chosen as a reward iteration, and Fido was given a reward value corresponding to its last action. This reward value was the difference between Fido's distance away from the target point before performing the action and Fido's distance from the target point after performing the action. Fido completed each trial when it was within 70 pixels of the point. Fido's feed-forward neural network had 4 hidden layers each with 10 neurons and outputted 5 wires to the wire-fitted interpolator. The exploration constant in equation 8 was held at 0.2.

Fido's last challenge, called "Follow," was to perform the classic robotic task of line following by controlling the motors of the simulator's differential drive system. At the start of each trial, Fido and a line with a random rotation were placed arbitrarily on a boundless plane within 768 pixels of one another. Fido was fed the ratio of its x displacement to its y displacement from the closest point on the line as well as its rotation. Every second action that Fido made was chosen as a reward iteration. An action's reward value was the difference between Fido's distance away from the line before performing the action and Fido's distance from the line after performing the action. Fido completed each trial when it had stayed within 60 pixels of the line for 10 consecutive actions (5 reward iterations). Fido's feed-forward neural network had 4 hidden layers each with 10 neurons and outputted 4 wires to the wire-fitted interpolator. The exploration constant in equation 8 was held at 0.2.

5.1.2 Simulation Findings

Each task was run 500 times to gather the data show in Table 1. The reward iterations values shown in the above data table were the medians of the data collected. The median is shown instead of the mean to discount a few large outliers that were present in data. The time data shown above is the mean of the data collected.

Task	Learning Iterations	Action Selection (ms)	Training Time (ms)
Flash	6	0.	6
Float	14	1	6
Drive	17	1	11
Line Follow	21	2	10.

Table 1: Number of Learning Iterations, Action Selection Time, and Training Time Per Iteration for Fido Simulation Tasks

The data collected through the simulator demonstrates the prowess of the Fido control system with computer-delegated reward in multiple configurations and situations. Fido was able to master both a holonomic and differential-drive motor control system (the "float" and "drive" tasks), proving its hardware agnostic capabilities. All tasks showed low numbers of reward iterations and low latency, allowing Fido to learn quickly and efficiently. Even the task that was most difficult for Fido, "Follow," took a median of 21 reward iterations, well within the patience of a human.

5.2 Data Collection in Person

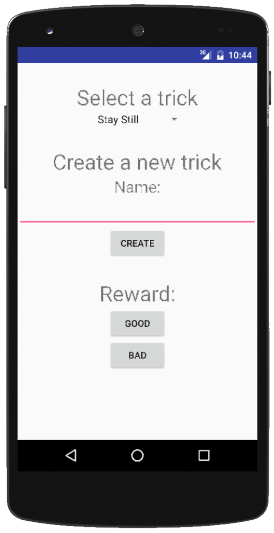

Figure 8: Android Training Application Screenshot

Data was next collected from Fido's hardware implementation to gage performance with human given feedback in real world situations. Tasks were configured through the Android application previously discussed, allowing easy setting of hyper parameters such as neural network structure. The Android application also allowed for a more direct form of administering positive and negative reinforcement.

5.2.1 Physical Experiments

The first task given to Fido's hardware implementation was "Stay Still," and as the name suggests Fido's sole responsibility for this task was to not move. Fido's feed-forward neural network had 3 hidden layers each with 10 neurons and outputted 3 wires to the wire-fitted interpolator. The exploration constant in equation 8 was held at 0.5. Fido was then given a reward value equal to $1 - 2|m|$ where m was the speed of Fido's motors ranging from -1 to 1.

Fido's hardware implementation tested on one more task, "Drive to Point." As on the simulator, Fido was tasked with moving to a point by controlling a differential drive system. At the start of each trial, Fido and the point were placed randomly on a smooth, hardwood surface within 0.75 meters of one another. Fido was told the ratio of its x displacement to its y displacement from its target point as well as its rotation. Every fourth action that Fido made was chosen as a reward iteration, and Fido was given a reward value corresponding to its last action. This reward value was chosen by the tester based on whether Fido moved toward the point or not. Fido completed each trial when it was within 10 cm of the point. Fido's feed-forward neural network had 4 hidden layers each with 10 neurons and outputted 5 wires to the wire-fitted interpolator. The exploration constant in equation 8 was held at 0.2.

5.2.2 Physical Findings

25 trials were done for each task to gather the data shown in Table 2. As with the simulation, reward iterations values shown are the medians of the data collected while time data shown is the mean of the data collected.

Task	Learning Iterations	Action Selection (ms)	Training Time (ms)
Stay Still	3	1	43.5
Drive to Point	18	4	65

Table 2: Number of Learning Iterations, Action Selection Time, and Training Time Per Iteration for Fido Hardware Tasks

Performance was slower on the hardware implementation than in simulation due to the limited computation power of the Intel Edison compute model compared to modern desktop computers. However, the results still demonstrated Fido's real world applicability through its ability to perform outside of a simulation environment.

6 Conclusion

A general robotic control system nicknamed Fido was developed that learned tasks with limited feedback. Fido couples the training of artificial neural networks with a wire-fitted moving least squares interpolator to achieve a continuous state-action space Q-learning reinforcement algorithm implementation. Fido leverages a Boltzmann distribution of probability based on reward to select actions, allowing it to continuously explore its state-action space. A kinematically accurate robot was simulated with a differential drive system, a sensor array, and other outputs for testing and evaluation purposes. The robot was trained on a number of common robotic tasks and successfully converged on these tasks in very few reward iterations while maintaining impressively low latency. Next a hardware implementation was built using a differential drive system, capable of utilizing real world sensor inputs to learn tasks such as driving to a point and staying still. In the future, we hope to further improve Fido's software and consider further applications in the present.

References

Baird, L. C., & Klopf, A. H. (1993). Reinforcement learning with high-dimensional, continuous actions. *Wright Laboratory, Wright-Patterson Air Force Base, Tech. Rep. WL-TR-93-1147*.

Biggs, G., & MacDonald, B. (2003). A survey of robot programming systems. , 1-3.

Dini, S., & Serrano, M. (2012). Combining q-learning with artificial neural networks in an adaptive light seeking robot.

Dudek, G., & Jenkin, M. (2000). *Computational principles of mobile robotics.* New York, NY, USA: Cambridge University Press.

Gaskett, C., Wettergreen, D., & Zelinsky, A. (1999). Q-learning in continuous state and action spaces. , 417-428.

Kim, D. S., & Papagelis, A. J. (n.d.). *Multi-layer perceptron: Artificial neural networks.* `http://www.cse.unsw.edu.au/ cs9417ml/MLP2/`.

MacLeod, C. (2010). *An introduction to practical neural networks and genetic algorithms for engineers and scientists* (Tech. Rep.). Robert Gordon University.

Rummery, G. A. (1995). *Problem solving with reinforcement learning* (Unpublished doctoral dissertation). University of Cambridge Ph. D. dissertation.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards* (Unpublished doctoral dissertation). University of Cambridge England.

Werbos, P. (1974). Beyond regression: New tools for prediction and analysis in the behavioral sciences.