A Universal Robotic Control System using Reinforcement Learning with Limited Feedback

Joshua Gruenstein Michael Truell

November 16, 2015

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 the specific hardware inputs and outputs and does not need to be modified for the task at hand. In addition, Fido was built to learn with limited feedback, allowing humans to train Fido in a reasonable time frame. This was achieved through the training of artificial neural networks with a wire-fitted moving least squares interpolator following the Q-learning reinforcement algorithm and an action selection policy that utilizes a Boltzmann distribution of probability. A kinematically accurate robot was simulated with a differential drive system, a sensor array, and other outputs. The robot was successfully trained to do an array of tasks with limited feedback, such as line following, driving to a set location, and blinking a multicolor light-emitting diode.

1 Introduction

The most prevalent control system used in mobile robotics is a procedurally programmed expert system (Biggs & MacDonald, 2003). Such systems use 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 Neural Network 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

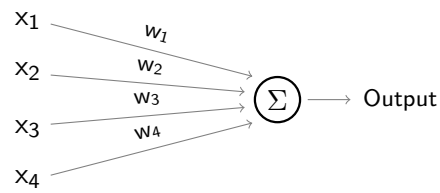

Figure 1: Single Neuron Diagram

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.

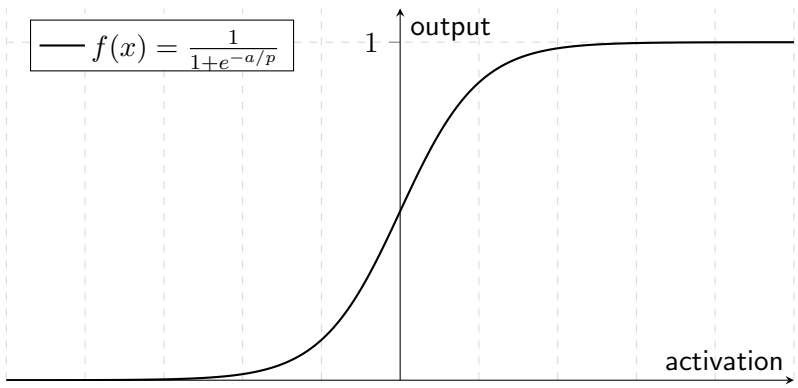

Figure 2: Sigmoid Function Graph

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.

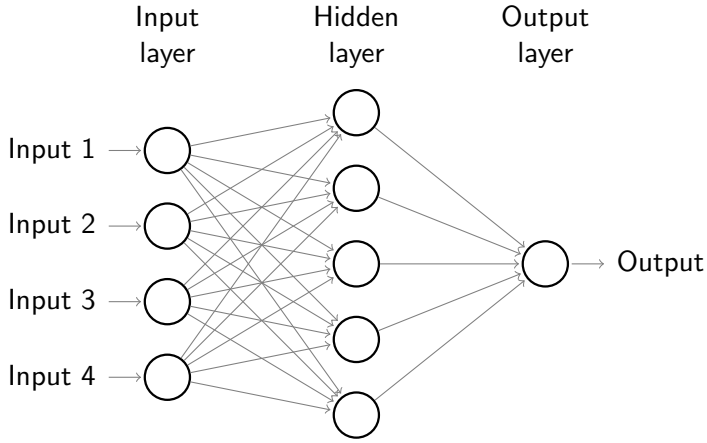

Figure 3: Single Output Feed-forward Network

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 Back Propagation

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).

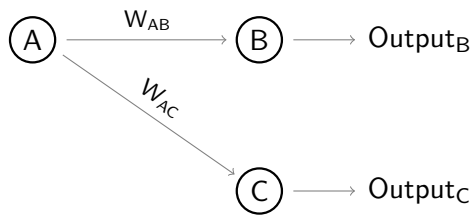

Figure 4: Single Branch in a Back Propagation Neural Network

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.

3 Learning

3.1 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.

The Q-Learning algorithm had to be immediately modified in two ways to make it suitable for Fido. Its scalability had to be improved, and it had to be able to work in continuous state-action spaces.

Q-learning in its simplest form uses a table to model the Q function, storing past state-action pairs and each pair's respective Q-value. However, this strategy lacks scalability. Since Fido will have to continue to learn throughout its whole lifetime, requiring it to store a large number of Q-values, this strategy is rendered impractical. To allow Fido to incorporate a large amount of data into its model of Q while keeping latency low, it is necessary to use a function approximator to estimate Q. Feed-forward neural networks were chosen for this task for their ability to model non-linear functions, lightweight computational footprint, and high trainablility.

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.

3.2 Wire-Fitted Q-Learning

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.

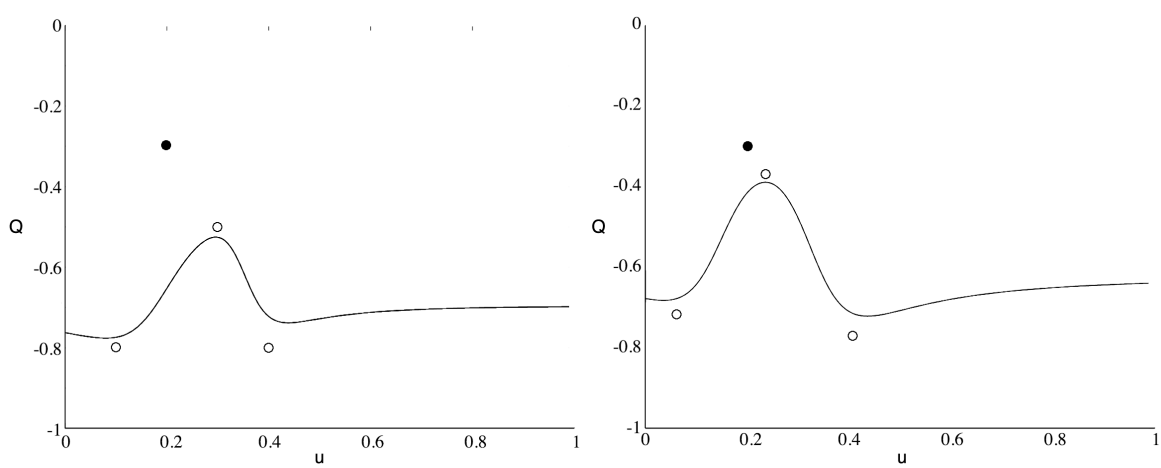

Figure 5: Moving Least Squares Interpolator (adapted from Gaskett, Wettergreen, & Zelinsky, 1999)

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.

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 using back propagation to output these new wires when given the previous state.

3.3 Boltzmann Probability Distribution Selection Policy

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.

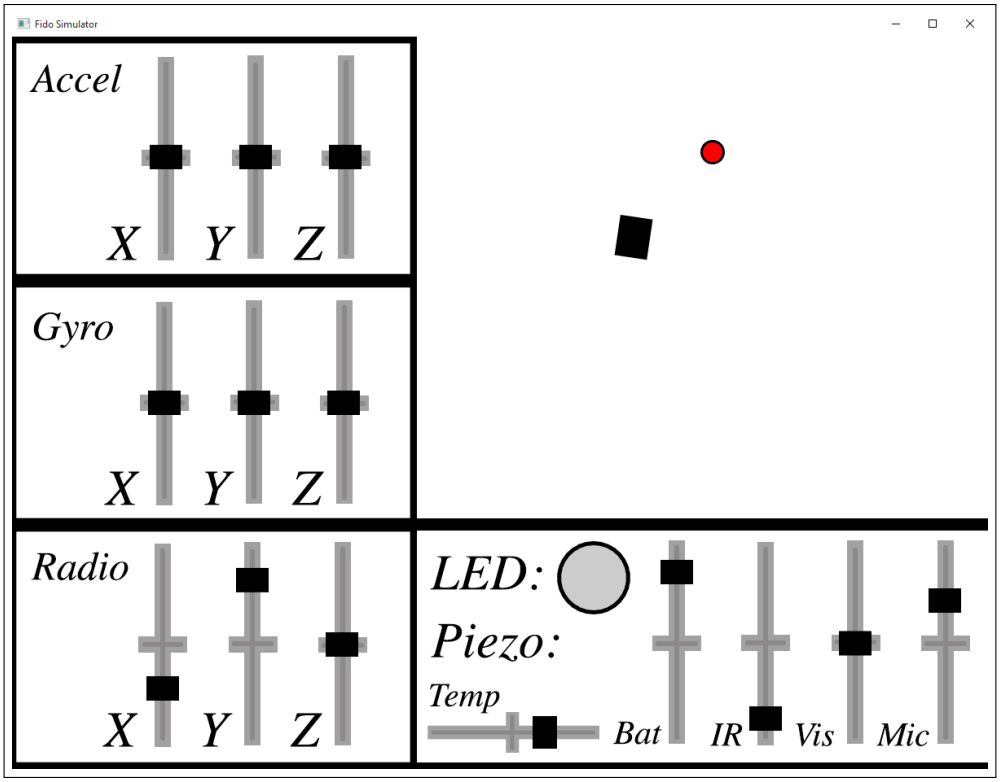

Figure 6: Screenshot of the Fido Simulator Graphical User Interface

4 Implementation

Fido was implemented in the C++ programming language with minimal use of libraries such as the Standard Library and Boost. This was to enable cross-platform functionality, lightweight and speedy performance, and the ability to run on microcontrollers as firmware in the future. 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 use 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 test Fido's performance. 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 diagrams and formula 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.

5 Results

5.1 Experiments

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

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.

5.2 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 Tasks

The data collected demonstrates the prowess of the Fido control system. 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 to Fido, "Follow," took a median of 21 reward iterations, well within the patience of a human.

6 Discussion

6.1 Future Software Research

Currently, the number of wires that our neural network outputs is constant. However, the complexity of the functions that the wire-fitting function has to model varies from task to task. To add to the universality of Fido, we plan on implementing a method of dynamically changing the number of wires that the neural network outputs. Such a method would gauge the variance and bias that the interpolator is experiencing. Variance is the deviation of a function from its mean. Bias is the error that results from under fitting. The wire-fitting function's variance could be measured in a range of actions. Bias could be measured as the moving average of the wire-fitted interpolator function's error at predicting Q-values, or the moving average of the distance that the wire-fitting function's wires have to move during gradient descent. Our proposed method would look to reduce $bias^2 + variance$ by increasing the number of wires if there is high bias squared, and decreasing the number of wires if there is high variance. Each time a wire is removed or added to create a new set of wires, this set of wires would be changed to best model the wire-fitting function formed by the past set of wires.

The topology of our feed-forward neural networks is static throughout Fido's lifetime; the way neurons are arranged in the feed-forward network don't change to fit the task at hand. To increase the generality of Fido, we would like to research ways to evolve the topology of Fido's neural network as it performs actions and receives reward. This may mean measuring variance and bias values and determining the direction of $bias^2 + variance$ as outlined above, but may also take the form of an existing variation of back propagation, of which there are many.

6.2 Hardware Implementation

A schematic and printed circuit board have been designed for Fido's hardware implementation, and is the next step in testing and improving the Fido control system. The PCB was modeled after the simulator reference design, and as such contains dual motor control, a photo-resistor for visible light, a photo-transistor for infrared light, etc. The board (designed in CadSoft Eagle) operates off of the Teensy 3.1 microcontroller development platform, which combines a 96Mhz ARM Cortex M4 microcontroller and a USB bootloader for easy debugging. The software base is in the process of being ported to the avr-gcc compiler toolchain for microcontrollers, which is only possible through a small code footprint and low reliance on C++ Standard Library functionality.

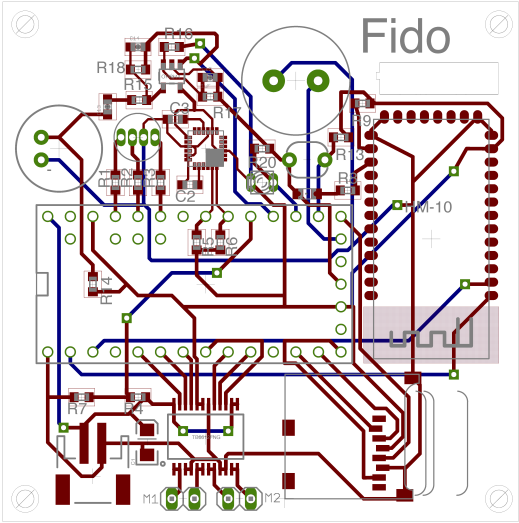

Figure 7: Fido PCB Layout

7 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. In the future, we hope to further improve Fido's software and are working toward the completion of Fido's hardware implementation.

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.