Using Deep Q-Learning to Control Optimization Hyperparameters

Samantha Hansen
IBM T.J. Watson Research Center

Abstract

We present a novel definition of the reinforcement learning state, actions and reward function that allows a deep Q-network (DQN) to learn to control an optimization hyperparameter. Using Q-learning with experience replay, we train two DQNs to accept a state representation of an objective function as input and output the expected discounted return of rewards, or q-values, connected to the actions of either adjusting the learning rate or leaving it unchanged. The two DQNs learn a policy similar to a line search, but differ in the number of allowed actions. The trained DQNs in combination with a gradient-based update routine form the basis of the Q-gradient descent algorithms. To demonstrate the viability of this framework, we show that the DQN's q-values associated with optimal action converge and that the Q-gradient descent algorithms outperform gradient descent with an Armijo or nonomonotone line search. Unlike traditional optimization methods, Q-gradient descent can incorporate any objective statistic and by varying the actions we gain insight into the type of learning rate adjustment strategies that are successful for neural network optimization.

1 Introduction

This paper demonstrates how to train a deep Q-network (DQN) to control an optimization hyperparameter. Our goal is to minimize an objective function through gradient-based updates of the form

$$x_{t+1} = x_t - \alpha_t g_t \tag{1}$$

where α_t is the learning rate. At each iterate x_t, we extract information about the objective derived from Taylor's theorem and line search methods to form a state feature vector. The state feature vector is the input to a DQN and the output is the expected discounted return of rewards, or q-value, connected to the action of increasing, decreasing, or preserving the learning rate. We present a novel definition of the reinforcement learning problem that allows us to train two DQNs using Q-learning with experience replay [10, 21] to successfully control the learning rate and learn the q-values associated with the optimal actions.

The motivation for this work is founded on the observation that gradient-based algorithms are effective for neural network optimization, but are highly sensitive to the choice of learning rate [9]. Using a DQN in combination with a gradient-based optimization routine to iteratively adjust the learning rate eliminates the need for a line search or hyperparameter tuning, and is the concept for the Q-gradient descent algorithm. Although we restrict this paper to deterministic optimization, this framework can extend to the stochastic regime where only gradient estimates are available.

We train two DQNs to minimize a feedforward neural network that performs phone classification in two separate environments. The first environment conforms to an Armijo line search procedure [1, 14], either the learning rate is decreased by a constant factor or an iterate is accepted and the learning rate is reset to an initial value. The second environment differs in that the learning rate can also increase and is never reset. The trained DQNs are the input to the Q-gradient descent

(Q-GD) versions 1 & 2, and we test them against gradient descent with an Armijo or nonmonotone [5] line search to show that these new algorithms are able to find better solutions on the original neural network, as well as on a neural network that is doubled in size and with three times the amount of data. We also compare how each algorithm adjusts the learning rate during the course of the optimization procedure in order to extract characteristics that explain Q-GD's superior performance.

The paper is organized as follows: in Section 2 we review reinforcement learning (RL) theory and in Section 3 we define the RL actions, state, and reward function for the purpose of optimization. Section 4 describes the Q-learning with experience replay procedure used to train the DQNs. In Section 5, we test the Q-GD algorithms against gradient descent with an Armijo or nonomontone line search on two neural networks that perform phone classification. Section 6 reviews relevant literature and finally, in Section 7 we provide concluding remarks and discuss future areas of research.

Notation: We use brackets indexed by either location or description to denote accessing an element from a vector. For example, $[s]_i$ denotes the i^{th} element and $[s]_{\text{encoding}}$ denotes the element corresponding to description 'encoding' for vector s.

2 Review of Reinforcement Learning

Reinforcement learning is the presiding methodology for training an *agent* to perform a *task* within an *environment*. These tasks are characterized by a clear underlying goal and require the agent to sequentially select an *action* based on the *state* of the environment and the current *policy*. The agent learns by receiving feedback from the environment in the form of a *reward*.

At each time step t, the agent receives a representation of the environment's state $s_t \in \mathcal{S}$ and based on the policy $\pi : \mathcal{S} \to \mathcal{A}$ chooses an action $a_t \in \mathcal{A}$. The agent receives a reward r_{t+1} for taking action a_t and arriving in state s_{t+1}. We assume that the environment is a Markov Decision Process (MDP), i.e. given the current state s_t and action a_t, the probability of arriving in next state s_{t+1} and receiving reward r_{t+1} does not depend on any of the previous states or actions.

A successful policy must balance the immediate reward with the agent's overall goal. RL achieves this via the action-value function $Q^\pi(s,a) : (\mathcal{S}, \mathcal{A}) \to \mathbb{R}$, which is the discounted expected return of rewards given the state, action, and policy,

$$Q^\pi(s,a) = \mathbb{E}_\pi \left[R_{t+1} \;\middle|\; s_t = s,\ a_t = a \right] \tag{2}$$

where

$$R_{t+1} = r_{t+1} + \sum_{k=1}^{T-t-1} \gamma^k r_{t+1+k},\ 0 < \gamma \leq 1, \tag{3}$$

T is the maximum number of time steps and the expectation is taken given that the agent is following policy π. The optimal action-value function, $Q^*(s,a) = \max_\pi Q^\pi(s,a)$ satisfies the Bellman equation,

$$Q^*(s,a) = \mathbb{E}_{\pi^*} \left[r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q^*(s_{t+1}, a') \;\middle|\; s_t = s,\ a_t = a \right] \tag{4}$$

which provides a natural update rule for learning. At each time step the effective estimate $\hat{y}_t$ and target y_t are given by

$$\hat{y}_t = Q_t(s_t, a_t), \quad y_t = r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q_t(s_{t+1}, a') \tag{5}$$

and the update is based on their difference; this method is referred to as Q-learning. Notice that the estimate/target come from LHS/RHS of (4) and will both continue to change until Q converges.

For finite number of states and actions Q is a look-up table. When the number of states is too large or even infinite, the table is approximated by a function. In particular, when the action-value function is a neural network it is referred to as a deep Q-network (DQN). A practical choice is to choose a network architecture such that the inputs are the states and the outputs are the expected discounted return of rewards, or q-value, for each action. We only consider the case of using a DQN and henceforth use the notation

$$Q(s;\theta) : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{A}|} \tag{6}$$

to denote that the DQN is parameterized by weights θ.

The weights are updated by minimizing the ℓ_2 norm between the estimate and target, $\|\hat{y}_t - y_t\|_2^2$, yielding iterations of the form

$$\theta \leftarrow \theta - \beta(\hat{y}_t - y_t)\nabla_\theta Q(s_t;\theta) \tag{7}$$

where β is the learning rate and

$$\hat{y}_t = Q(s_t;\theta), \qquad [y_t]_a = \begin{cases} r_{t+1} + \mathbb{1}_{t\neq T-1}\left(\gamma \max_{a'\in\mathcal{A}}[Q(s_{t+1};\theta)]_{a'}\right) & a = a_t \\ [Q(s_t;\theta)]_a & a \neq a_t. \end{cases} \tag{8}$$

For the last action, only the reward is present in the target definition and for the non-chosen actions, the targets are set to force the error to be zero.

The action at each time step is chosen based on the principle of exploration versus exploitation. Exploitation takes advantage of the information already garnered by the DQN while exploration encourages random actions to be taken in prospect of finding a better policy. We employ an ϵ-greedy policy which chooses the optimal action w.r.t the DQN's q-values with probability $1-\epsilon$ and randomly otherwise:

$$a_t = \begin{cases} \arg\max_a [Q(s_t;\theta)]_a & r \geq \epsilon \\ \text{randomly chosen action} & r < \epsilon \end{cases} \tag{9}$$

where $r \sim U[0,1]$.

Equation (9) is the effective policy since it maps states to actions. Q-learning is an off-policy procedure because it follows a non-optimal policy (with probability ϵ a random action is taken) yet makes updates to the optimal policy, as illustrated by the max term in (8). For a comprehensive introduction to RL, see [18].

3 Reinforcement Learning for Optimization

In this section, we outline the environment, state, actions, and reward function that define the reinforcement learning problem for the purpose of optimization.

3.1 Actions

We present two procedures for adjusting the learning rate and show how they are implemented in practice. The first strategy mimics an Armijo line search [1, 14] in that the learning rate is reset to an initial value after accepting an iterate and can only henceforth be decreased. The second strategy permits the learning rate to increase or decrease and is never reset. The two methods are outlined in Algorithm 1 and are referred to as Q-gradient descent (Q-GD) versions 1 & 2, respectively.

Q-GD is a gradient descent optimization procedure that uses a trained DQN to determine the learning rate. The Q-GD inputs are an initial iterate and learning rate x_1 and α_c, trained DQN $Q(s;\theta)$, and maximum number of time steps T. We use the notation x_t to denote the candidate iterate, which changes at every time step, and $\bar{x}$ to represent an accepted iterate with associated decent direction $d(\bar{x})$. In steps 3 and 4, a state feature vector representative of the objective (discussed in the next section) is formed and passed through the DQN to determine the action. After the action is taken, the candidate iterate is updated in step 12.

When a good initial learning rate is known then the first version is preferable, e.g. $d(\bar{x})$ is the Newton direction and $\alpha_c = 1$ for convex f. For non scale-invariant search directions, such as the gradient direction, the second version is advantageous.

Algorithm 1 Q-gradient descent versions 1 & 2

Input: initial iterate x_1, initial learning rate α_c, trained DQN $Q(s;\theta)$, number of time steps T

1: Set $\bar{x} = x_1$, $d(\bar{x}) = -\nabla f(x_1)$, $\alpha_1 = \alpha_c$
2: **for** $t = 1, \ldots, T$ **do**
3: Compute state feature vector s_t
4: $a_t = \arg\max_a [Q(s_t;\theta)]_a$
5: **if** $a_t = a_{\text{half}}$ **then**
6: $\alpha_{t+1} = \frac{1}{2}\alpha_t$
7: **else if** $a_t = a_{\text{double}}$ **then** ▷ Only for version 2
8: $\alpha_{t+1} = 2\alpha_t$
9: **else if** $a_t = a_{\text{accept}}$ **then**
10: $\bar{x} = x_t,\ d(\bar{x}) = -\nabla f(\bar{x}),\ \alpha_{t+1} = \begin{cases} \alpha_c & \text{version 1} \\ \alpha_t & \text{version 2} \end{cases}$ ▷ Update accepted iterate
11: **end if**
12: $x_{t+1} = \bar{x} + \alpha_{t+1} d(\bar{x})$ ▷ Update candidate iterate
13: **end for**
14: **return** $x^* = x_T$

3.2 Environment and State

The environment is a combination of the objective function $f : \mathbb{R}^n \to \mathbb{R}$ and set of allowed actions and needs to be formulated as a MDP in order for the Q-learning algorithm to operate. The Markov condition could be satisfied by including the initial iterate, and the current, as well as all proceeding learning rates and descent directions into the state definition. However, for objective functions with large number of variables such an approach is computationally prohibitive and would severely limit the trained DQN's ability to generalize to a broader family of functions. We seek to define the state such that it characterizes the objective function at a given iterate, contains some history, and is universal to all functions. We use a nonmontone line search as a starting point since it provides an effective criteria for determining the learning rate that is independent of function variable size or type.

A nonmonotone line search chooses the learning rate such that the new iterate is sufficiently less than the maximum objective value of the past M iterates,

$$f(x_t + \alpha_t d_t) \leq \max_{i=t,\ldots,t-M+1} f(x_i) + c\alpha_t d_t^T \nabla f(x_t), \quad c > 0. \tag{10}$$

This suggests that the state features needed in order to determine the learning rate are the current learning rate, candidate iterate objective value, max objective from the past M steps, and the dot

product between the descent direction d_t and gradient $\nabla f(x_t)$. Although this feature set would neither satisfy the Markov property nor completely capture the objective, updates based on (10) work well in practice and we use these statistics as motivation for the state features.

We employ an encoding that indicates whether the candidate iterate is higher/lower than the M lowest achieved objective values. Let F_M^{t-1} be a list of the M lowest objective values obtained up to time $t-1$, the state encoding is given by

$$[s_t]_{\text{encoding}} = \begin{cases} 1 & f(x_t) \leq \min(F_M^{t-1}) \\ 0 & \min(F_M^{t-1}) < f(x_t) \leq \max(F_M^{t-1}) \\ -1 & \text{otherwise.} \end{cases} \tag{11}$$

The number of function evaluations must also be a state feature since the states wouldn't otherwise be stationary and the maximum number of time steps T designates an absorbing state. Based on RPROP [16], the final state feature is a measure of alignment between successive descent directions

$$[s_t]_{\text{alignment}} = \frac{1}{n} \sum_{i=1}^{n} \text{sign}([d_t]_i [d_{t-1}]_i). \tag{12}$$

In summary there are six features: current learning rate, objective value, dot product between the search direction and gradient, min/max encoding (11), number of function evaluations, and alignment measure (12).

For the purpose of making the state features independent of the specific objective function, all of the features are transformed to be in the interval $[-1, 1]$. For each feature $[s]_i$, a maximum and minimum value is estimated so that

$$[\hat{s}]_i = 1 - 2([s]_i - [s_{\min}]_i)/([s_{\max}]_i - [s_{\min}]_i). \tag{13}$$

Additionally, since the objective values and gradient norms both converge towards a lower bound c_i, these features are transformed twice. First via $[s]_i \leftarrow 1/([s]_i - c_i)$ and they by (13), where c_i is set to 0 for the gradient norm and an objective lower bound f_{lb} for the function values. In general, f_{lb} can be set to zero for objectives that are a sum of loss functions.

3.3 Reward Function

The reward function is crucial in ensuring that the DQN learns a policy consistent with the goal of finding the lowest objective value in the fewest number of steps, and we define it as the inverse distance from the objective lower bound,

$$r_{id}(f, x_t) = \frac{c}{f(x_t) - f_{lb}}, \quad c > 0, \quad f_{lb} < f(x) \ \ \forall x. \tag{14}$$

The reward function (14) is strictly positive and asymptotes as f approaches the lower bound.

We tested reward functions based on a sufficient decrease condition or change in objective value between successive iterates,

$$r_{sd}(f, x_t) = 1_{f(x_{t-1}) \geq 1.001 f(x_t)}, \qquad r_{\text{oc}}(f, x_t) = f(x_{t-1}) - f(x_t) \tag{15}$$

and found that they did not adequately capture the optimization goal. To compare the different reward functions we plotted $f(x_T)$ against $R_{\max} = \max_t R_t$; for each training episode of DQN v1 we recorded the sequence of objective values ($f(x_T)$ being the objective value at the last times step) and used this information to calculate $R_{\max}$ for each reward function. Figure 1 shows that reward functions based on sufficient decrease or objective change yield high $R_{\max}$ values for suboptimal final solutions. The main difference is that (14) is based on degree of difficulty in decreasing the objective and will generate the highest rewards during the final time steps.

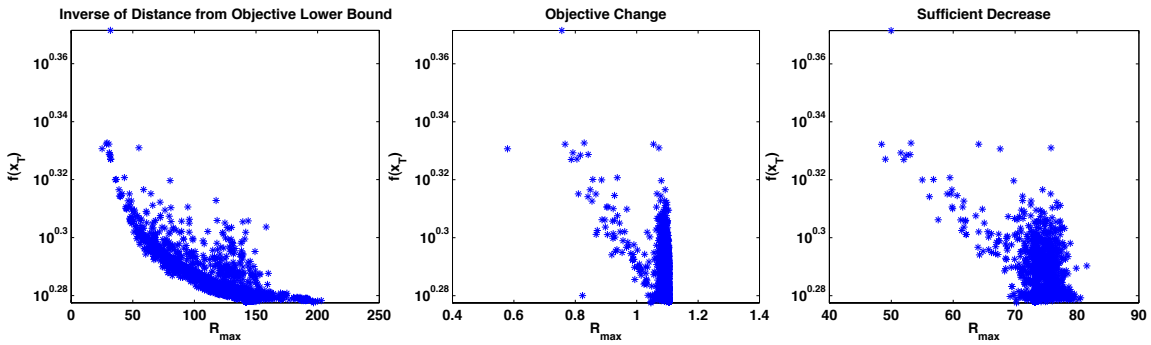


Figure 1: Comparison of reward functions. The images plot $R_{\max} = \max_t R_t$ versus $f(x_T)$ for reward functions defined by r_{id} (inverse distance from objective lower bound), r_{oc} (objective change) and r_{sd} (sufficient decrease) given by equations 14 and 15. Only r_{id}, shown in the leftmost graph, has the highest $R_{\max}$ values concentrated towards lowest final objective values.

4 Training

This section outlines the Q-learning with experience replay method used to train DQN versions 1 & 2 [21, 10]. Algorithm 2 exhibits the overall procedure, but omits some of the specific details, which are discussed in the subsections for the sake of clarity. Note that updates w.r.t. $f(x)$ are explicitly shown and are indexed by the time step t while the DQN update in step 25 is referenced via equation (16) and is implicitly indexed by the time step and episode.

The DQN learns how to minimize the function $f(x)$ through repeated attempts, called learning episodes. For each learning episode, the x iterate is set to an initial value and the DQN then has T time steps to find the lowest objective value. An alternative approach for limiting the number of time steps is to end the episode once the objective has decreased past a certain threshold. Both approaches force the DQN to learn a trade off between finding a good learning rate and exploring the space. Restricting the number of time steps reflects real world applications where there are computational and time constraints and also does not require a-priori knowledge of the objective function.

4.1 Experience Replay

An experience consists of a $(s_i, a_i, r_{i+1}, s_{i+1})^j$ tuple for some episode $j \in [1, e]$ at time step $i \in [M-1, T]$, where M and e are in Algorithm 2 steps 2 and 3. These tuples are stored in a memory of experiences $\mathcal{E}$. Instead of updating the DQN with only the most recent experience, a subset $\mathcal{S} \subset \mathcal{E}$ of experiences are drawn from memory and used as a mini-batch to update the DQN:

$$\theta \leftarrow \theta - \frac{\beta}{|\mathcal{S}|} \sum_{(s_i, a_i, r_{i+1}, s_{i+1})^j \in \mathcal{S}} (\hat{y}_i - y_i) \nabla_\theta Q(s_i; \theta) \tag{16}$$

where the estimate $\hat{y}_i$ and target y_i are given via (8).

The A most recent episodes along with the top B best games (in terms of $R_{\max}$ value) are stored in memory. At each DQN update (step 25) the subsample $\mathcal{S}$ is formed by randomly drawing experiences from $\mathcal{E}$ and an experience from each of the top B best games. Adding randomly drawn experiences to the mini-batch helps prevent the DQN from over learning during a particular time and episode.

4.2 Training Specifications

The Q-learning input parameters in Algorithm 2 for both DQN versions 1 & 2 were fixed as follows: the discount factor was set to $\gamma = .99$ and the exploration probability ϵ was initially set to 1 then uniformly decayed to .1 over the first 100 episodes. For experience replay, $A = 45$, $B = 5$, and the mini-batch size was set to $|\mathcal{S}| = 32$. Additionally, for the first 50 episodes the top B best games were not used in the mini-batch sample. The constants c_1 and c_2 used to calculate the reward (see steps 20 and 22) were fixed as .1 and .12, respectively. The total number of episodes E are 150K for version 1 and 400K for version 2.

The objective input parameters in Algorithm 2 consist of the objective function $f(x)$ with lower bound f_{lb}, initial weights x_1, initial learning rate α_c, encoding memory M, and the total number of time steps T. The objective function has the form

$$\frac{1}{N}\sum_{i=1}^{N}\ell(h(z_i;x),t_i) \tag{17}$$

where z_i is an acoustic feature vector with phonetic label t_i, $\ell(\cdot)$ is a cross entropy loss, and $h(z;x)$ is a feedforward neural network parameterized by x with sigmoid activations and a softmax function at the output layer. We set the input objective function to f_{train}, which has a neural network architecture $65 \times 16 \times 8 \times 42$ and $N = 5000$ data points. The number of time steps is $T = 1000$ and $M = 3$.

At the start of each episode, the x iterate is reset to x_1 and is updated for the first M time steps using the initial learning rate (step 4) in order to form the first state feature vector. In steps 8 and 9, the six state features form the input to the DQN and the resulting action is determined by an ϵ-greedy policy. Based on the action, the learning rate is either modified, step 10 or 12, or the current iterate is accepted and a new gradient direction is calculated, step 15. The iterate x_t is updated in step 17 and this causes the environment to change to the next state (step 18). The reward for arriving to state s_{t+1} is calculated using either the objective value at the new iterate (step 20) or the previous iterate (step 22) for when the action is to accept. As an aside, we found it beneficial to calculate the reward for each action at the last time step since the targets associated with absorbing states do not change during training and thus play a vital role for propagating back information. The tuple $(s_t, a_t, r_{t+1}, s_{t+1})^e$ form an experience and is added to memory $\mathcal{E}$ (step 24). In additional to the current experience, a random subset of experiences are drawn and used to form a mini-batch update for the DQN (step 25).

Special modifications were needed for training DQN v2 since one of its actions permits the learning rate to increase. Too large of a learning rate resulted in updates that caused the objective function to diverge and consequently produce state vectors with infinite features. To prevent this from happening, we used a maximum and minimum learning rate as part of the training procedure. If DQN v2 attempted to increase/decrease the learning rate above/below these values then it would receive a reward of -1 and the episode would terminate early. In addition, we employed an rmsprop update procedure for training DQN v2 [20].

DQN versions 1 & 2 have an architecture of $6 \times 32 \times 16 \times |\mathcal{A}|$ with sigmoid activations for the hidden layers and an identify activation for the last layer. The initial learning rate was set to $\alpha_c = 4$ for version 1 and $\alpha_c = 2$ for version 2. Additionally, for version 2 only learning rates in the range $[.01, 8]$ were allowed.

Algorithm 2 Q-Learning with Experience Replay

Objective Parameters: f, f_{lb}, x_1, α_c, M, T

Q-Learning Parameters: E, θ_0, γ, ϵ, c_1, c_2, β

$\theta \leftarrow \theta_0$

for $e = 1, \ldots, E$ **do** ▷ For each learning each episode

  for $t = 1 \ldots, M-1$ **do**

    $x_{t+1} = x_t - \alpha_c \nabla f(x_t)$

  end for

  set $\bar{x} = x_M$, $d(\bar{x}) = -\nabla f(x_M)$, $\alpha_M = \alpha_c$

  for $t = M, \ldots, T$ **do**

    Generate state feature vector s_t

    Choose action a_t according to ϵ-greedy policy (9)

    if $a_t = a_{\text{half}}$ **then**

      $\alpha_{t+1} = \frac{1}{2}\alpha_t$

    else if $a_t = a_{\text{double}}$ **then** ▷ Only for version 2

      $\alpha_{t+1} = 2\alpha_t$

    else if $a_t = a_{\text{accept}}$ **then**

      $\bar{x} = x_t,\ d(\bar{x}) = -\nabla f(\bar{x}),\ \alpha_{t+1} = \begin{cases} \alpha_c & \text{version 1} \\ \alpha_t & \text{version 2} \end{cases}$

    end if

    $x_{t+1} = \bar{x} + \alpha_{t+1} d(\bar{x})$

    Generate state feature vector s_{t+1}

    if $a_t \neq a_{\text{accept}}$ **then**

      $r_{t+1} = c_1/(f(x_{t+1}) - f_{lb})$

    else if $a = a_{\text{accept}}$ **then**

      $r_{t+1} = c_2/(f(\bar{x}) - f_{lb})$

    end if

    Add experience $(s_t, a_t, r_{t+1}, s_{t+1})^e$ to memory $\mathcal{E}$

    Sample $\mathcal{S} \in \mathcal{E}$ and update θ via (16)

  end for

end for

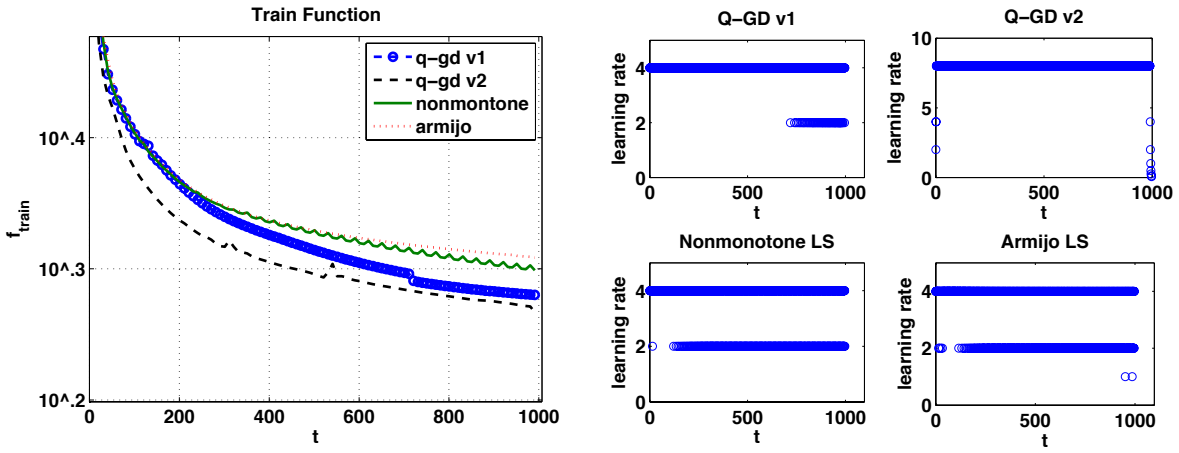
return θ

5 Experiments

The trained DQNs along with the initial learning rates α_c are the input to the Q-gradient descent algorithms versions 1 & 2 outlined in Algorithm 1. Since there are no theoretical guarantees that the DQNs would find a good policy or converge, we demonstrate that Q-GD versions 1 & 2 are effective algorithms by comparing them against gradient descent with an Armijo or nonmonotone line search and show that the DQN q-values associated with the optimal actions converge to the discounted return of rewards at each time step.

The line search algorithms operate under the same rules as Q-GD v1, but an iterate is accepted only if (10) is satisfied. We set $c = 10^{-4}$ and $M = 3$ for nonmonotone and, by definition, $M = 1$ for Armijo.

5.1 Results on Train Function



(a) Objective Value versus Time Step

(b) Learning Rate versus Time Step

Figure 2: Comparison of Q-GD versions 1 & 2 and gradient descent with a nonmonotone or Armijo line search on train function.

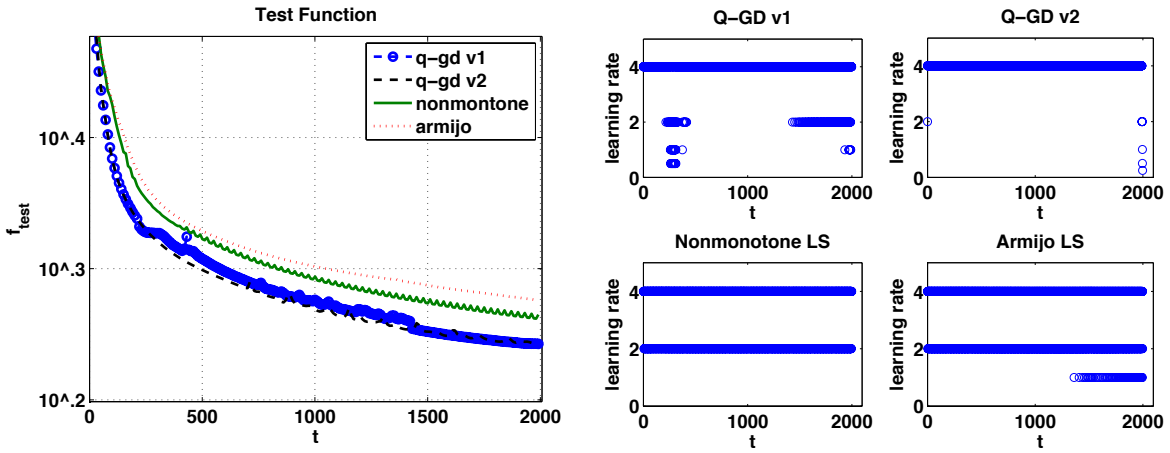
We first compare Q-GD versions 1 & 2 and gradient descent with an Armijo or nonmonotone line search on the function used to train DQN versions 1 & 2; f_{train} has the form (17) with $N = 5000$ and feedforward neural network architecture $65 \times 16 \times 8 \times 42$. Figure 2a demonstrates their performance in minimizing f_{train} and figure 2b plots the learning rate at each time step. After 1000 time steps, the final objective values are 1.86, 1.91, 1.98, and 2.04 for Q-GD v2, Q-GD v1, nomonotone, and Armijo, respectively.

The plots of the learning rates illuminate why the Q-GD algorithms are superior. Q-GD v2 has the advantage that it can increase the learning rate and its policy for minimizing the train function was very simple: it increased the learning rate from 2 to 8 during the first initial time steps and then left the learning rate unchanged until decreasing it at each of the last seven time steps. Q-GD v1 offers a fairer comparison to the Armijo and nonomontone line searches since they all follow the same structure: every time an iterate is accepted the learning rate is reset to 4 and can only then be decreased by a factor of two. The notable difference between Q-GD v1 and the line search algorithms is the frequency in which the learning rate is decreased. Q-GD v1 decreased the learning rate 5.1% of the time while the Armijo and nonmonotone line searches decreased the

learning rate 36.4% and 27.3% of the time. Q-GD v1 also only decreased the learning rate during the final quarter of the optimization procedure.

The learned policies illustrate that a good initial learning rate is more important than a line search procedure for fast initial objective decrease and it is beneficial to decrease the learning rate more aggressively during the final time steps. Unlike the line searches, the Q-GD algorithms have knowledge of when the optimization procedure is going to end and can act accordingly.

5.2 Generalization Ability



(a) Objective Value versus Time Step

(b) Learning Rate versus Time Step

Figure 3: Comparison of Q-GD versions 1 & 2 and gradient descent with a nonmonotone or Armijo line search on test function.

We next test to determine if the strategy learned by DQN versions 1 & 2 on the train function also works for a new, but related function. The test function has the same form as the train function, but with three times the amount of data and double the number of variables ((17) with $N = 15000$ and architecture $65 \times 32 \times 16 \times 42$). The purpose of this configuration is to show that we can train the DQN using a small problem and later implement it on larger problems in terms of both variable size and data. We also increased the number of time steps from 1000 to 2000.

Figure 3 exhibits how Q-GD versions 1 & 2 and the nonmonotone and Armijo line search algorithms measure on the test function. In figure 3a, we observe that the algorithms retain their relative ordering regarding objective decrease in a fixed number time steps; the final values are 1.73, 1.74, 1.84 and 1.89 for Q-GD v2, Q-GD v1, nonomonotone and Armijo, respectively. The gap in performance between Q-GD versions 1 & 2 reduced, showing that Q-GD v1 was more adapt at generalizing to a new function. As with the train function, both Q-GD versions 1 & 2 decreased the learning rate less frequently than either the nonmonotone or Armijo line searches. However, both Q-GD versions were more cautious using a higher learning rate at the start of the of the optimization procedure. Q-GD versions 1 & 2 maintained their underlying strategies, except version 1 chose to decrease the learning rate during the first quarter and version 2 only initially increased the learning rate to 4 (as opposed to 8).

Overall, these results show that Q-GD versions 1 & 2 were robust when given a new, larger function and used over a longer number of time steps.

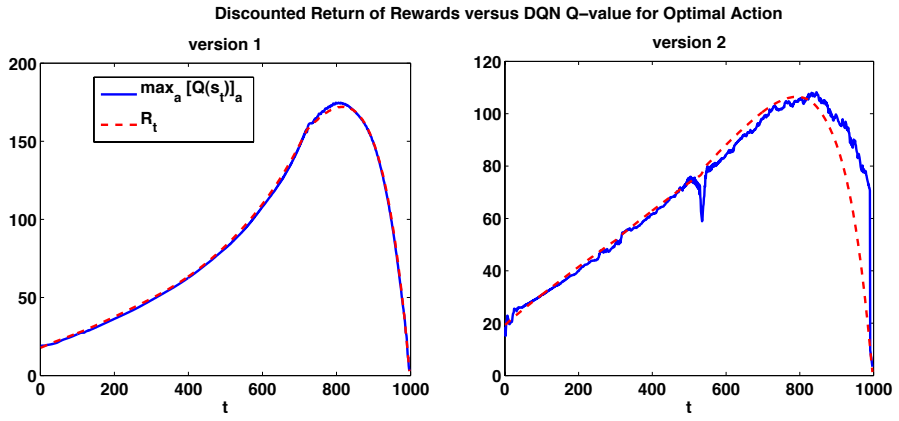


Figure 4: Plot of DQN versions 1 & 2 predicted q-value for optimal action versus the discounted return of rewards (3) at each time step t on train function.

5.3 Convergence of DQN Q-values

The purpose of this section is to show that the six state features detailed in Section 3.2 are rich enough for the DQN to discriminate states in order to learn the q-values associated with the optimal actions. We also demonstrate the effect of individually zeroing out the state features for Q-GD version 1 on the train function.

For the final episode, we recorded the q-value associated with the selected action (no longer using an ϵ-greedy procedure) and resulting reward at each time step in order to compare the DQN predicted q-values against the discounted return of rewards, defined by (3). Figure 4 shows that DQN v1's q-values converged to the discounted return of rewards while DQN v2 found the overall shape of the distribution. Even though DQN v2 was trained with more episodes (400K versus 150K), the addition of one extra action exponentially increases the search space, creating a much more difficult problem.

To investigate how the state features influence the Q-GD algorithms, we ran Q-GD v1 with either the objective value, gradient norm, or alignment measure set to zero; since the features are transformed to lie in the interval $[-1, 1]$ this corresponds to fixing a given feature at its median value. We left the learning rate, objective encoding, and number of time steps unchanged as they are arguably the bare minimum inputs needed to satisfy the Markov property.

Table 1 reports the final objective value and the ratio of halving the learning rate or accepting an iterate obtained for setting a given state feature to zero during a run of Q-GD v1 on the train function. The baseline (none of the features are set to zero) is a final objective of 1.91 and 51/946 half/accept ratio. As a result of zeroing out a state feature, DQN v1 chooses to half the learning rate more frequently and ends up with a worse solution. This experiment shows that DQN v1 depends on each feature to determine the appropriate action.

Table 1: Effect of setting a state feature to zero. Baseline (none of the feature are set to zero) is a final objective value of 1.91 and a 51/946 half/accept ratio.

Feature	Objective	Half/Accept
objective value	1.96	320/677
gradient norm	1.98	273/724
alignment measure	2.04	364/633

6 Related Work

Neural network models yield state of the art performance in speech recognition, natural language processing, and computer vision [6, 8, 3]. Riedmiller popularized neural networks as an approximation to the action-value function with the advent of the Neural Fitted Q Iteration [15] and applications of his work appear in settings ranging from playing games to robotics [12, 19, 10].

Using reinforcement learning to replace an optimization heuristic or be embedded within the optimization algorithm has been explored in a variety of domains [2, 4, 11, 13, 17]. However, none of the previous approaches use deep Q-learning or our proposed RL formulation. Our work is most similar to [17]; the authors use RL to replace a Levenberg-Marquardt heuristic for controlling a damping parameter used in a Gauss-Newton update routine. Unlike our work, they approximate the action-value function by a linear combination of basis functions, which they train using Least Square Policy Iteration. To our knowledge, our work is the first to successfully apply deep Q-learning to controlling an optimization hyperparameter.

7 Conclusions

This paper lays the foundation for using deep Q-learning to control an optimization hyperparameter. We defined the state, reward function, and actions such that a DQN could learn how to control the learning rate used in a gradient-based optimization routine, resulting in two Q-gradient descent algorithms. Given that there are no theoretical guarantees that the DQN would find the optimal policy or that its q-values would converge, we presented numerical evidence that the Q-GD algorithms performed better than either gradient descent with an Armijo or nonmonotone line search and that the DQNs' q-values for the optimal action converged to the discounted return of rewards at each time step. Additionally, we demonstrated that the Q-GD algorithms were able to generalize when the train function was replaced with a larger test function.

A main advantage of the Q-gradient descent method is that it can easily incorporate any objective statistic by adding it to the state feature vector. Future areas of work involve using this framework to explore additional state features that can facilitate optimization decisions. We trained the DQNs in a simple environment in order to demonstrate feasibility. To make this method practical for large scale optimization it is necessary to extend Q-GD to the stochastic regime, that is create Q-stochastic gradient descent. A final area of work involves expanding the actions to include controlling additional hyperparameters, such as a momentum term. Overall, the presented framework allows us to develop new optimization algorithms and gain intuition to the type of strategies that are successful for minimizing neural networks.

References

[1] Larry Armijo. Minimization of functions having lipschitz continuous first partial derivatives. *Pacific Journal of mathematics*, 16(1):1–3, 1966.

[2] Justin A Boyan and Andrew W Moore. Learning evaluation functions for global optimization and boolean satisfiability. In *AAAI/IAAI*, pages 3–10, 1998.

[3] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[4] Marco Dorigo and LM Gambardella. Ant-q: A reinforcement learning approach to the traveling salesman problem. In *International Conference on Machine Learning*, pages 252–260, 1995.

[5] Luigi Grippo, Francesco Lampariello, and Stephano Lucidi. A nonmonotone line search technique for newton's method. *SIAM Journal on Numerical Analysis*, 23(4):707–716, 1986.

[6] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, and Tara N Sainath. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 29(6):82–97, 2012.

[7] Nitish Shirish Keskar and George Saon. A nonmonotone learning rate strategy for sgd training of deep neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 4974–4978. IEEE, 2015.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[9] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[10] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, DTIC Document, 1993.

[11] Victor V Miagkikh and William F Punch III. Global search in combinatorial optimization using reinforcement learning algorithms. In *Proceedings of the Congress on Evolutionary Computation*, volume 1, pages 189–196. IEEE, 1999.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[13] Robert Moll, Theodore J Perkins, and Andrew G Barto. Machine learning for subproblem selection. In *ICML 00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 615–622, 2000.

[14] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[15] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *Machine Learning: ECML 2005*, pages 317–328. Springer, 2005.

[16] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE, 1993.

[17] Paul L Ruvolo, Ian Fasel, and Javier R Movellan. Optimization on a budget: A reinforcement learning approach. In *Advances in Neural Information Processing Systems*, pages 1385–1392, 2009.

[18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[19] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[20] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop. *COURSERA: Neural networks for machine learning*, 2012.

[21] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.