Learning to learn by gradient descent by reinforcement learning

Ashish Bora

Abstract

Learning rate is a free parameter in many optimization algorithms including Stochastic Gradient Descent (SGD). Choosing a good value of learning rate is non-trivial for important non-convex problems such as training of Deep Neural Networks. In this work, we formulate the optimization process as a Partially Observable Markov Decision Process and pose the the choice of learning rate per time step as a reinforcement learning problem. On a simple quadratic function family, our agents using Deep Q Networks are able to outperform two simple baselines. We also implement a strong baseline given by 'Graduate Student Descent' and show that DQN agents approach its performance. Finally, we present several visualizations that may be helpful to understand the DQN training process.

1. Introduction

Gradient based optimization is one of the most important classes of optimization techniques. It has been extensively used in a wide variety of settings, including the recent success of Deep Neural Networks achieving state of the art performance on many tasks [10, 4, 12].

Stochastic Gradient Descent (SGD) is probably the simplest, and one of the very widely used gradient based optimization methods. Given a differentiable function $f(\theta)$, to find the minimizer θ^* of this function, SGD starts at a randomly chosen initial point θ_0, and then attempts to iteratively refine it to lower the function value. The refinement is done by taking a step along the negative gradient of the function with respect to parameters θ. Thus, if θ_t, is the sequence of parameter values produced by SGD, we have:

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta_t} f(\theta_t), \tag{1}$$

where $\nabla_{\theta_t} f(\theta_t)$ represents the gradient of the $f(\theta)$ at θ_t, and η is the learning rate (also known as step size). We shall call $f(\theta)$ interchangeably as loss.

It is quite important to select the learning rate η carefully. If we choose a very large learning rate, it may lead to wild oscillations or divergence. On the other hand, choosing a very small learning rate, leads to slow optimization and increases the chance of getting stuck in a local minimum.

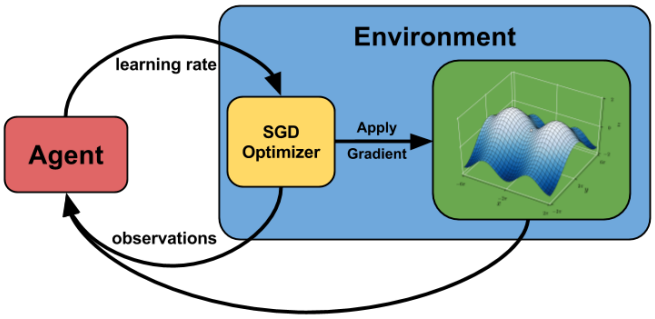


Figure 1: Our approach. Function image is from https://en.wikipedia.org/wiki/Saddle_point

While there is a good theoretical understanding, and practical algorithms to choose learning rates for convex functions, the same is not true for many non-convex function families. Particularly for important non-convex optimization problems, such as training of Deep Neural Networks, proper choice of learning rate can give significant improvements. In practice, methods like grid search or random search [3] are often used to select the step size. Trial and error methods like these can be prohibitively expensive.

On the other hand, the loss trajectory as we apply SGD gives important cues about a good value of learning rate. Several rules of thumb about how the learning rate should be changed, given a particular observation of this trajectory are widely known. For example, if the loss is increasing, showing large oscillations, or has saturated, it is advised that the learning rate be reduced. Conversely, if the rate of decrease of loss is too little at the beginning of learning, it is recommended to try a larger learning rate.

With this motivation, we hypothesize that it should be possible to design an algorithm that can learn to set a good value of learning rate by monitoring the optimization process. In this work, we formulate this process as a Partially Observable Markov Decision Process (POMDP) and present an attempt towards using reinforcement learning to choose the learning rate at every optimization step. On a quadratic function family, our algorithm is able to outper-

form two simple baselines and approaches the performance of a version of 'Graduate Student Descent' algorithm. We also show several visualizations that throw some light on the DQN training process.

2. Basic setup

In this section we describe our basic setup at a high level. Further details are given in relevant sections. A schematic diagram of our basic setup is shown in Fig. 1. The function to be optimized ($f(\theta)$) and the SGD optimizer are part of the environment. At every time step, the agent is asked to pick a learning rate for the SGD update. The SGD optimizer then updates the parameters using the learning rate according to Equation (1). At every step, the agent gets some observations about the current state of the environment, i.e. from the SGD optimizer and the function, to help aid its decision. We treat this as an episodic task with fixed number of time steps. The goal of the agent is to minimize the function. Accordingly, the total episodic reward is a function of the final loss value (see section 3).

Similar to the setting in [2], we note that generalization in our framework is with respect to a distribution over functions that we would like to optimize. i.e. the agent is said to have learned well, if after training on random functions from a distribution over functions, it is able to optimize new functions from the same distribution.

Additionally, considering all functions makes learning impossible due to *No Free Lunch* theorems [15]. Thus, we restrict our agent to operate on a fixed class of parametrized and differentiable functions that we would like to optimize with SGD. We denote this class by $\mathcal{F}$ and the functions in this class by $f(\theta)$ where θ are the parameters. Multiple parametrized families can be part of $\mathcal{F}$, which allows our framework to handle a wide range of function classes, such as neural networks with different architectures. We assume a given fixed distribution (unknown to the agent) $\mathcal{D}$ over $\mathcal{F}$, with respect to which we would like to generalize. Additionally, to handle several functions from $\mathcal{F}$, each with a different number of parameters, we constrain the observations that the agent gets to be of a fixed dimension irrespective of the function to be optimized.

At the start of every episode, a fresh random function $f(\theta)$ is sampled such that $f(\theta) \sim \mathcal{D}$. Our agent then tries on minimize this function by sending actions to the SGD optimizer and getting observations from the environment.

3. POMDP formulation

The state of the environment at time t is given by $S_t = (f, \theta_0, t, \theta_t)$, where f is the function we are trying to minimize, θ_0 is the initial value of the parameters, and θ_t is the current value of the parameters.

To simplify the problem of choosing a learning rate, we use a discrete set of possible learning rates. In our experiments we use the set $\{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}\}$. As described in the previous section, when one of these actions is chosen, SGD update with the learning rate chosen by the action is applied. We add one more action called RESTART. If this action is chosen, the parameters are reset to the initial point, i.e. θ_0. This is why we need θ_0 to be part of the state.

In addition to transitions by SGD updates or RESTART, a third type of transition is encountered at the last step of an episode. If $t = T$, then irrespective of the chosen action, we transition to the terminal state marking the end of the episode. This is the reason t is part of the state.

At each step the agent gets the following observations from the state:

1. $1 - t/T$, i.e. the fraction of time left.
2. $\log f(\theta_t)$
3. $\log \|\theta_t\|_2$
4. $\log \|\nabla_{\theta_t} f(\theta_t)\|_2$

We take logarithm to avoid very large or very small values.

Since the functions from $\mathcal{F}$ can have a very different range of values, we focus on the relative decrease in the function value. Thus the total episodic reward is chosen to be the natural logarithm of the relative change in the function value, i.e. $\log\left(\frac{f(\theta_0)}{f(\theta_T)}\right)$. Although giving this reward at the final step is sufficient, it results in very sparse rewards. To avoid this, we distribute the rewards across time steps using reward shaping [13]. Accordingly, the reward given at time t is the relative decrease in function value, i.e. $\log\left(\frac{f(\theta_t)}{f(\theta_{t+1})}\right)$. We use an undiscounted formulation so that the total sum of per step rewards is exactly the same as the desired total episodic reward. We assume f to have non-negative values and for stability, we also add a small constant (10^{-10} in our experiments) to both the numerator and the denominator before taking the logarithm.

4. Deep Q-Network

We use Q-learning with function approximation to predict the Q-values for all actions. Deep Q-networks were first introduced in [12] and have showed impressive performance

on playing ATARI games. Inspired by their success, we also use neural networks for approximating Q-values.

We note that since SGD is a stochastic method, the observations may fluctuate. Thus, the immediate set of observations may not be sufficiently informative to determine a good learning rate. However, hidden beneath these variations, there usually are clear trends. Thus we also want the state of the model to account for the history of the observed values, not just the current ones. Additionally, we are in a partially observable setting with very few observations at each time step. Thus, similar to [12], we use stacking over time to give more context to the DQN.

At the beginning of every episode, we do not have enough observations to stack. So as a workaround, at the beginning of every episode, we also give the agent some burn in. This is done by taking several RESTART actions and stacking the observations as a result. We do not increment t in this process. Note that the reward for these steps is exactly zero and the parameter values remain at their initial value, i.e. θ_0. Number of burn-in actions is set to equal the stacking horizon so that the stack is completely filled up.

Other than stacking, the DQN architectures vary across experiments and are described in the relevant sections.

5. Related Work

Choosing learning rates for optimization algorithms is a widely studied problem. Several fixed learning rate schedules, such as a exponential or polynomial decrease, are commonly used. By far, the simplest schedule is to fix a constant learning rate. A popular choice is to use decreasing learning rates that satisfies the stochastic approximation conditions. These can be guaranteed to converge to at least a local minima with probability one [11]. Another such method is the Search Then Converge (STC) algorithm [6]. However, these methods are agnostic to the problem at hand.

Adaptive methods such as backtracking line search are also widely used. Other algorithms like RMSProp [14], Adagrad [7], and Adam [9] try to change the optimization procedure so as to approximate higher order moments using first order information. These algorithms typically introduce more hyperparameters in the learning process which need to be tuned, but often give superior performance as compared to a pure first order method like SGD. While these methods do try to adapt to the problem at hand, they do not explicitly optimize for the fact that we would like to stop the optimization at a finite time.

The work in [2] is probably the closest to our work. This paper introduces a framework where the optimizer is completely replaced by a LSTM. Each parameter value and the gradient of loss with respect to that parameter is input into an LSTM and the output is a single value denoting the update to be applied to the model. The parameters of the LSTM itself are then optimized using stochastic gradient descent. Their method is also capable of learning approximate second or even higher order methods since the LSTM cells have memory. Our approach in contrast is constrained to first order SGD method since we are only predicting the learning rate. Even so, our approach enjoys other advantages over this work. First, our approach is highly scalable since our agent gets a small set of observations and performs limited computation on it, i.e. application of DQN. In contrast their method applies an LSTM per parameter. Second, our formulation is generic and can work with any (possibly non-differentiable) metric used as a reward function. For example, our reward function can be the accuracy of a classification model while the model is being trained using SGD with cross-entropy loss. Thirdly, The RESTART action allows the agent to be more adventurous and let it try higher learning rates.

Our work is also related to [5]. The limitation of our work is that we consider only a finite set of possible learning rates as opposed to continuous values used in this work. Our discrete formulation however allows us to use include a RESTART action.

6. Quadratic Environment

For our experiments, we use the following family of quadratic optimization problems.

$$\mathcal{F} = \{\|aW\theta - y\|^2 \mid a \in \mathbb{R}; W \in \mathbb{R}^{10\times 10}; y, \theta \in \mathbb{R}^{10}\}$$

We construct a distribution over these functions by taking a to be an exponentially distributed random variable with mean = 1, and each entry of W and y to be IID Gaussian random variables with zero mean and unit variance. The initial value of parameters, θ_0, also has IID Gaussian entries. Once sampled, these values are kept fixed for the duration of the episode and resampled at the beginning of each episode to give a new function with a new starting point.

These functions are quite easy to optimize since they are convex. In fact, we can determine the optimal step size quite easily by finding the maximum eigenvalue of a^2W^TW. Despite the simplicity, we chose this environment as a test to demonstrate the abilities of DQNs on this task.

7. Metrics

To monitor the training process and evaluate our agent, we report the total reward per episode averaged over several

episodes, i.e.

$$\frac{1}{N}\sum_{i=1}^{N}\log\left(\frac{f(\theta_0^i)}{f(\theta_T^i)}\right) = \log\left(\prod_{i=1}^{N}\frac{f(\theta_0^i)}{f(\theta_T^i)}\right)^{1/N},$$

where i indexes over episodes and N is the total number of episodes used for averaging. We can interpret this metric as the logarithm of the geometric mean of ratio of initial function value to final function value.

We track this metric while training as well as evaluation. While training, there is a finite probability of taking a random action (see Section 9). This probability is zero while evaluation. Accordingly, we see that the training metric is lower than the evaluation metric.

8. Baselines

To compare the performance of our algorithm, we implemented several baselines. These are described in this section.

Our first baseline is the random agent. This agent just takes random actions at each time step irrespective of the observations. Our second baseline is a fixed agent. It takes a particular fixed action at each time step, again ignoring any observations it may see. Among the many possible fixed agents, one for each action, we only report results for the one that maximizes our evaluation metric. These two baselines are very simple and ignore the structure of the problem completely. Nonetheless, they are useful to understand the problem space and set a minimum for the performance of our DQN agents.

Our third baseline is named after 'Graduate Student Descent' [1], a pun on gradient descent. It is meant to stand for informal search performed by graduate students do to make their methods work. In our setting, we implement it as a simple algorithm that looks at the history of optimization for two time steps and makes decision accordingly. The algorithm is given in Algorithm 1

Algorithm 1 Graduate Student Descent

```
Take the action with highest learning rate
while not done do
    if the loss is decreasing then
        take the same action
    else
        take RESTART
        take action with a smaller learning rate
```

To implement this algorithm, the agent needs access to past observations for at least two time steps. Thus, we give the agent access to two past values of loss and the actions that were taken. At time t, to make its decision the agent has access to action taken and loss value at time steps $t-1$ and $t-2$ and it has to select action at time t. Similar to DQN agent, we also give burn in to this agent with two RESTART actions at the beginning of each episode.

Results for baselines are shown in Fig. 2. Note that for these agents, there is no gap between the training and evaluation performance. All rewards are scaled up by a factor of 10. We see that random agent gets an average reward of about 2.6, which translates to $\sim$ 1.3 fold decrease in function value on average (geometric mean). The agent with fixed learning rate 10^{-3} achieves the best performance among fixed agents. It gets an average reward of about 17.8($\sim$ 6 fold decrease). Graduate Student agent achieves the best performance with an average reward of about 33.2 ($\sim$ 27.6 fold decrease).

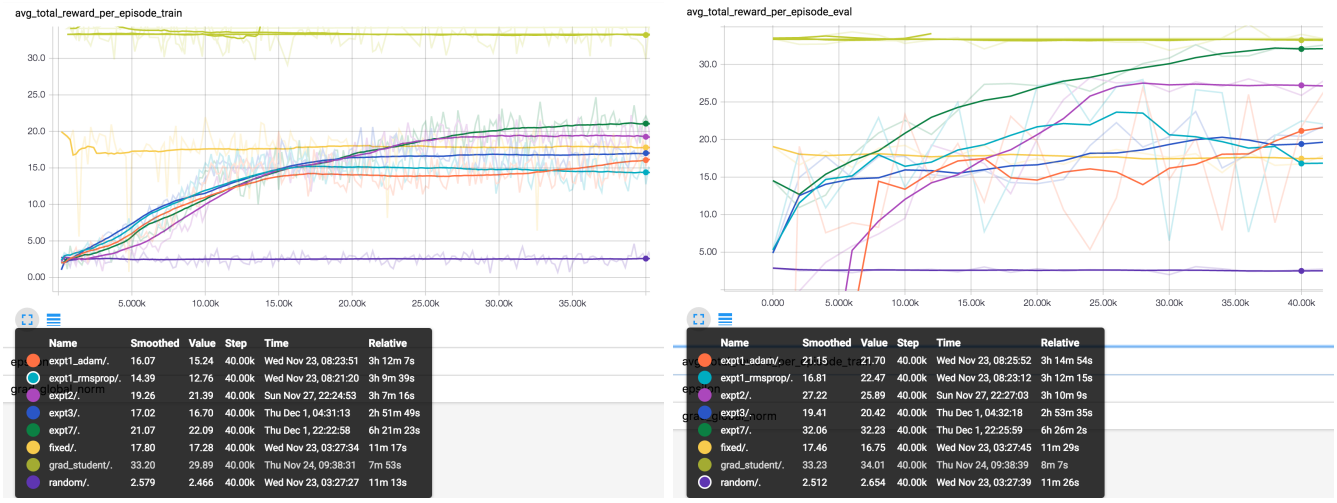
9. Experiments

For training of DQNs, we follow Algorithm 1 in [12]. We use a large experience replay memory to reduce the correlation between updates to the Q network. We decrease the value of ϵ linearly from 1 to 0.1 over the first one million updates after which it is held constant. This makes the DQN learning method an instance of off-policy learning. We also use a target Q network to further decorrelate Q network updates, and gradient clipping to stabilize DQN learning.

In all our experiments we set T, the number of steps per episode to be 100. All rewards are scaled up by a factor of 10 to bring them in a better range. The training batch size is kept at 32 and gradient clipping at 10. Training metric is averaged every 200 episodes. The evaluation is performed once every 2000 training episodes and is averaged over 1000 episodes. For all experiments, we do a grid search over optimization algorithms and learning rates. We tried Adam [9] and RMSProp [14] as our optimization algorithms and all learning rates from the set $\{10^{-1}, 10^{-2}, \ldots, 10^{-6}\}$.

9.1. Stacked observations with depth-2-DQN

In this experiment we stack the observations over time 8 time steps. Since the agent gets 4 observations from the environment at each time step (Secion 3), the input to the Q network is of size $8 \times 4 = 32$. We use a two layer neural network architecture. First linear layer maps the inputs to 12 hidden units. This is followed by a ReLU non-linearity and another linear layer maps them to 6 outputs corresponding to 6 Q-values. We update the target Q network every 200 training episodes.



(a) Training (b) Evaluation

Figure 2: Average reward per episode while training (left) and evaluation (right), for baselines and various DQN agents

Results are shown in Fig. 2. From the grid search, we show only the best results for Adam as well as RMSProp. Both algorithms achieve the best results at learning rate 10^{-4}, while the one with Adam is slightly better. We see that the best DQN agents perform about as well as the fixed agent on the evaluation metric. These agents are outperformed by a large margin by the Graduate Student agent.

9.2. Adding actions to DQN state

The Graduate Student agent has access to the history of actions that were taken in the past. This information is not available to the DQN agent if we only stack the observations. Thus, in this experiment, we tried to add the history of actions taken to the DQN input. Since ordering of actions is arbitrary, we use a one hot encoding on actions yielding a vector with size equal to the number of actions (6 in our case) per time step. After stacking for 8 timesteps, the input to DQN is thus of dimension 8 * (4 + 6) = 80 inputs. Since the number of inputs is larger, we also increase the size of hidden layer of the DQN to 32. Rest of the architecture remains the same.

Results are shown in Fig. 2. From the grid search, we observe that the best results are obtained with Adam and a learning rate of 10^{-6}. The performance improves over not using actions in the state. On the evaluation metric, the best DQN agent achieves 27.22. This is still smaller than 33.2 achieved by the graduate student.

9.3. Deeper architecture

It is quite surprising that the DQN agent did not achieve performance comparable to the Graduate Student agent. Both agents have access to exactly the same information and hence the DQN agent had the opportunity to learn the same rules as the Graduate Student agent. One possible explanation for this is that the DQN is not expressive enough to perform the kind of computation the Gradute Student Agent does. To test if this was true, we tried DQN architecture with a larger depth and more hidden units.

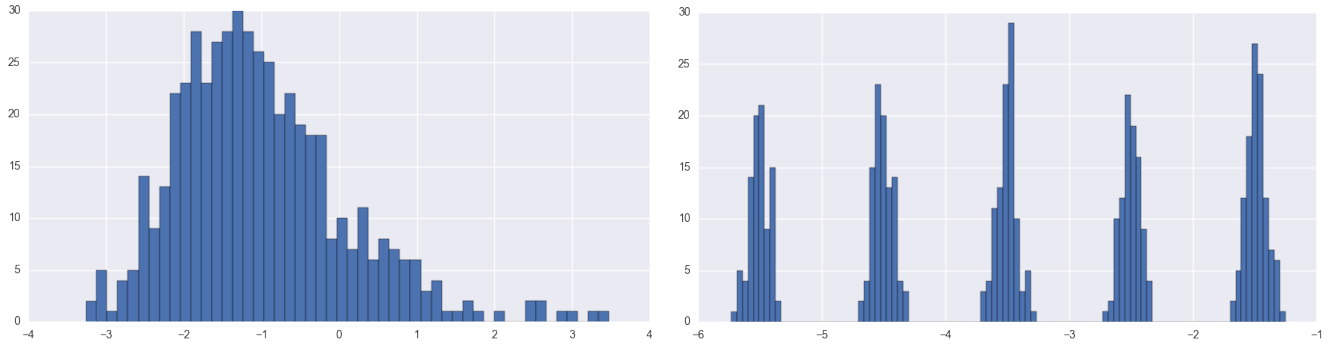
Inputs to the DQN are same as the previous experiment, i.e. stacked observations and stacked actions. These are then linearly mapped to 128 units followed by a ReLU nonlinearity. They are then linearly mapped to 64 units, again followed by a ReLU. Finally we again linearly map from 64 to 6, to get 6 Q-values, one for each action.

The results are shown in Fig. 2. We see that more depth seems to give no benefit. Theoretically, model with more depth can learn to mimic the smaller one and hence should have been at least as good. But it might be the case that the models are not getting enough samples for some of the actions and hence it is hard to learn well. We shall explore this in the next experiment.

9.4. Eigenvalue distributions

In this experiment we aim to understand our environment a bit better, aiming to explain why the DQN agents do not perform as well as the Graduate Student agent.

Recall that we are working with functions from a quadratic



(a) Exponential

(b) Multimodal

Figure 3: Optimal fixed step size distribution with exponential (left) and multimodal (right) distribution for a.

family (section 6). For functions from this family, we can compute the optimal fixed step size as the inverse of the maximum eigenvalue of $a^2 W^T W$. Using any learning rate smaller than twice the optimal one leads to convergence (from any starting point), but the rate of convergence becomes slower as we deviate from the optimal value. Using a learning rate larger than twice the optimal value may lead to divergence (depending on the initial point).

Fig. 3a shows the distribution of the logarithm (base 10) of the optimal fixed step size. We see that for most functions 10^{-2} is the best fixed learning rate among the ones available to our agent and for a considerable fraction 10^{-3} is a good choice. For most functions very small learning rates are not needed. This also explains why 10^{-3} is the best fixed learning rate: it achieves good performance on most problems while making mistakes on a relatively small fraction of the problems. Such distribution however is potentially problematic for the DQN agents because the action distribution is skewed. In effect the agents rarely see any examples where actions 10^{-6} or 10^{-5} are good. Most examples give high rewards for 10^{-2} and 10^{-3}. This skew may be one explanation for poor DQN performance.

9.5. Multimodal Environment

It is plausible that having a more equalized distribution of best actions would help in learning. Thus we propose a new distribution over the quadratic function family, which we call Quadratic-Multimodal. Here, instead of sampling a from the exponential distribution, we sample a uniformly at random from the set $\{10^0, 10^{0.5}, 10^1, 10^{1.5}, 10^2\}$. Fig. 3b shows the distribution of the logarithm (base 10) of the optimal fixed step size for this environment. This is much more balanced than the previous distribution.

We run the same baselines on this environment. The results are shown in Fig. 4. Unlike previous environment, random agent now get large negative rewards. For this environment, the best fixed agent is the one that takes the action with the smallest learning rate, 10^{-6}. This is to be expected because there is a significant portion of functions for which higher learning rates lead to divergence.

Out of our previous DQN experiments, depth two gave the best results. So we use the same for DQN models on this environment. We tried stacking only observations, and stacking observations and actions. For each experiment, we again do grid search over optimization algorithms and learning rates. Best results are achieved by using learning rate 10^{-5} and Adam optimizer for both models.

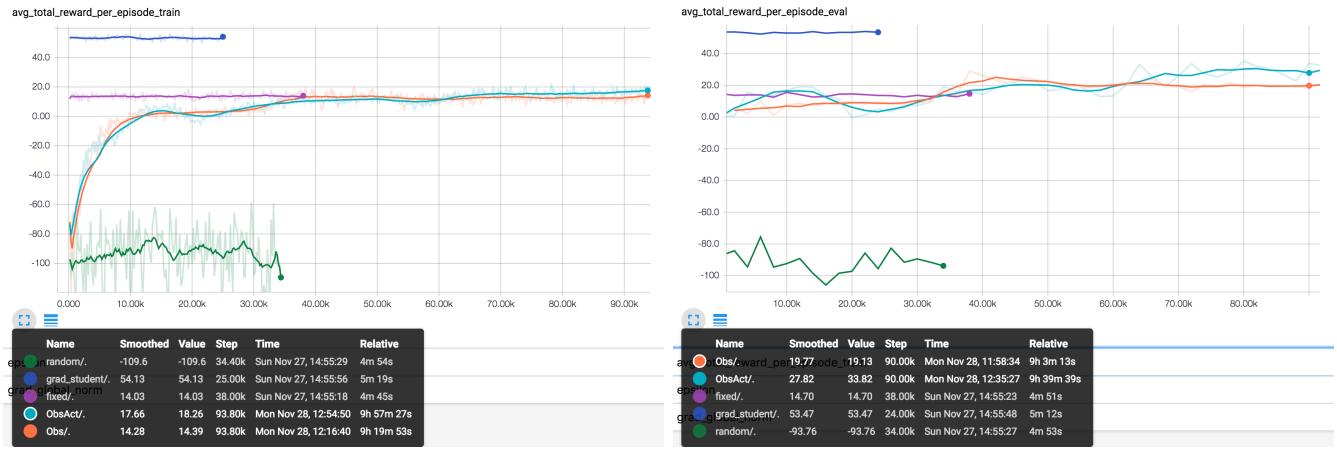
Broadly, the results are similar to the previous environment. The DQN models are slightly better than the best fixed agent. The DQN with observations and actions performs better than the one without access to actions. Both of them are still outperformed by the Graduate Student agent. This means that unbalanced action distribution may not be the sole or the primary reason for poor performance of DQNs.

9.6. DQN training visualizations

To further dig into the DQN learning process and understand the problems, in this experiment, we logged several statistics as the network trains and present three visualizations based on this data.

TD loss is shown in Fig. 5a (bottom right). For most models we see that the loss is going down as we train. This is the desired behavior. For the model in second experiment, the loss goes down but then starts to increase.

Histogram of actions taken by various agents is shown in Fig. 5b. We see that the DQN agents are taking a variety of actions, while still performing only marginally better than the agent with fixed action. Taking a variety of actions while



(a) Training (b) Evaluation

Figure 4: Average reward per episode while training (left) and evaluation (right) on the multimodal environment

training is a good sign of exploration.

Histogram of Q values is shown in Fig. 5a. These show a consistent increase in the Q values. Although Q learning is known to overestimate the sum of future rewards, the plots show that the estimated values are very large as compared to the average reward per episode. Thus our models are grossly overestimating the Q values. This may be one of the explanations for why the DQN agents are not performing as well as the Graduate Student agent.

9.7. Slower target Q update

One of the reasons for large overestimation of Q values can be too frequent update of target Q network. This results in a feedback loop through which Q values can keep increasing indefinitely. Thus in this experiment we try to decrease the frequency of target network update.

In our previous experiments, we updated the target Q network every 200 episodes, i.e. after every 20000 updates. We try two larger values: updating after every 600 episodes and after every 2000 episodes. For each setting, we again do the grid search and find that using Adam optimizer with learning rate 10^{-6} and update target network every 600 episodes gave the best results.

This experiment was done using the original distribution (a is an exponential random variable). The results are shown in Fig. 2. We observe that with this modification, the performance improves by a large margin. The best DQN agent now achieves 32.06 on the evaluation metric which is quite close to the performance of the graduate student agent (33.23).

10. Discussion and Future work

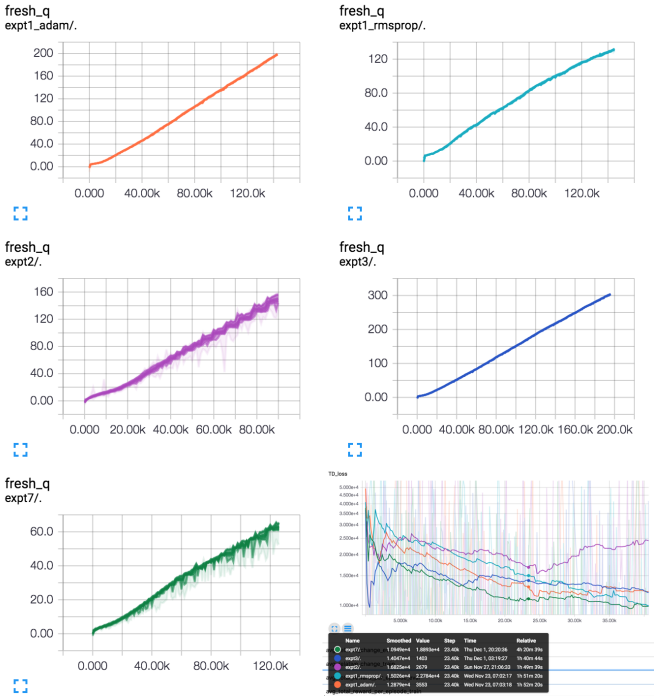
In terms of formulation, we can also try to optimize for number of steps in the training. i.e. we can add an extra action which results in termination of the SGD procedure. To encourage fewer optimization steps, we can give a small negative reward at each time step.

Actions such as restart or terminate are discrete and hence in this work, we used a simpler formulation of discrete set of values for learning rates. It might be useful to consider mixed action types so that we certain discrete actions, but at the same time model the learning rate choice as a continuous action space.

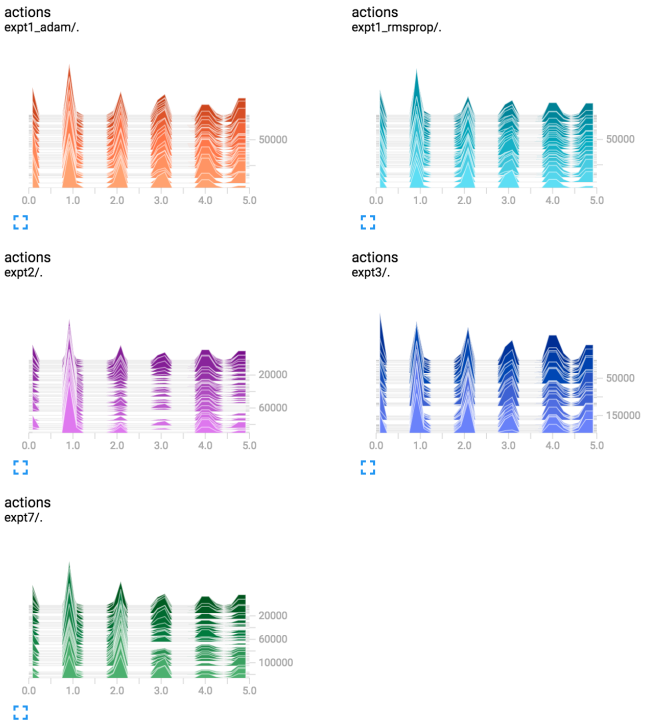
As seen on the experiments, the DQN agents are still unable to outperform the Graduate Student agent. Thus exploring other state representations, other DQN architectures, trying other optimization algorithms or better hyperparameter settings are some of the things that can be helpful. It may also be useful to try simpler function approximators like linear functions with tile coding.

Ideally the DQN should use the full history of observations, not just a stack of the previous few time steps. Thus, we can use a Recurrent Neural Network (RNN) to consume the set of observations gathered as a part of the interaction with the environment at each time step. We can use another neural network which maps the hidden state of the RNN to Q-values for each of the possible actions. This architecture is similar to Deep Recurrent Q learning [8].

In this work, we used a simple quadratic function family. For this work to be useful, it is necessary to try similar ideas for training of more complex and non-convex models like neural networks.



(a) Q values and TD Loss plot. X axis is training episodes



(b) Action Distribution. X axis is action IDs, Y axis is training episodes, Z axis is histogram strength

11. Conclusion

In this work, we studied the problem of choosing the learning rate for gradient based optimization problems. We formulated it as a POMDP and deployed Deep Q Networks to learn controllers for this task. On a simple quadratic function family, we have shown that DQN agents outperform simple baselines and get very close to the performance of a strong baseline. We have shown visualizations of the training that throw some light on the inner workings of the DQN training process and helped us fix some crucial hyperparameters. Finally, we briefly discussed several interesting directions of future research which can build on this work.

Acknowledgement

The author would like to thank Matthew Hausknecht for several useful discussions and tips about the problem setup and DQN training.

References

[1] https://sciencedryad.wordpress.com/2014/01/25/grad-student-descent/.

[2] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, and N. de Freitas. Learning to learn by gradient descent by gradient descent. *arXiv preprint arXiv:1606.04474*, 2016.

[3] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[4] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[5] C. Daniel, J. Taylor, and S. Nowozin. Learning step size controllers for robust neural network training. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[6] C. Darken and J. E. Moody. Note on learning rate schedules for stochastic optimization. In *NIPS*, pages 832–838, 1990.

[7] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[8] M. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable mdps. *arXiv preprint arXiv:1507.06527*, 2015.

[9] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In

Advances in neural information processing systems, pages 1097–1105, 2012.

[11] H. Kushner and G. G. Yin. *Stochastic approximation and recursive algorithms and applications*, volume 35. Springer Science & Business Media, 2003.

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[13] A. Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287, 1999.

[14] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2), 2012.

[15] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.