Hyper-parameters Optimization of Neural Network Using Deep Q-Learning

Zhengnyu Hu(zh2317), Zixiao Zhang(zz2500), Haocong Shi(hs2991)

Abstract

We used a reinforcement learning method to optimize the hyper-parameters of a neural network. By demonstrating a effective state, actions and reward function, we use Q-Learning with experience replay to train a deep Q-network (DQN), getting its efficient weight parameters. Combining the trained DQN with gradient-based update routine, we use this new algorithm to control the hyper-parameters of a neural network, making the neural network converges more quickly. Also, because of the adjustment of the learning rate during the progress, it can avoid the result being stuck in a local optimal.

1. Introduction

In the traditional neural network model, we usually set the initial parameter value based on experience and use stochastic gradient descent (SGD) optimization to update the hyper-parameters. However, it is required lots of previous experience and costs lots of time and memory space to get the adjust the hyper-parameters. Therefore, we want to use an algorithm to get the optimal hyper-parameter without prior experience and with less time.

The core idea of our work is to train a deep Q-network (DQN) to control two hyper-parameters of neural network, learning rate and regularization parameter, to meet the optimal result. We tried to minimize the objective function via gradient-based updates.

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

where α is the learning rate and g_t is the gradient of x_t. We extract the feature vector from each state and used them as the input of the DQN. The output of it is the discounted return, or q-value, actions, or learning rate and regularization parameter. With the definition of the reinforcement learning method we used, we can used the DQN with experience replay of Q-learning to control the hyper-parameters we wanted to optimize.

We know that gradient-based algorithms provides a good result in neural network optimization, however, it is sensitive to learning rate. Also, regularization parameter is a significant hyper-parameter for the performance of a neural network. So if a system can learn to control these two hyper-parameter, rather than choosing the values of them through human experience, the efficiency of a neural network will be improved. That's the motivation of our work.

This paper is organized as follows: Section 2 is the related work to review what have been done in the hyper-parameter optimization area and what inspire us to do the work in this paper. In section 3, we introduce the reinforcement learning model we used and the algorithm we developed. Section 4 is about how we train the DQN and we show the analysis of our experiment result as well as making a comparison with other method. Finally in section 5, we draw a conclusion of our work.

2. Related Work

Neural network were widely used and proved to be effective in solving many problem in natural language processing, speech recognition and computer vision. Neural network was used to approximate the value function (**?**). And then it was extended to the approximation of action-value function using the Neural Fitted Q Iteration (**?**). In current time, hyper-parameters optimization is an important research topic in machine learning, and is widely used in practice (**?**)(**?**)(**?**)(**?**). (**?**) came up with an automatic learning model which learning from a neural network to optimize parameters of other neural networks. (**?**) used a recurrent network to generate the model descriptions of neural networks and train this RNN with reinforcement learning to maximize the expected accuracy of the gen-

Proceedings of the 30th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28.

erated architectures on a validation set. DeepMind came up with a CNN model trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. The model is used to play Atari and has a good performance. Applications of using neural networks in RL appear in settings ranging from playing games to robotics (**?**)(**?**). (**?**) presented 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 hyper-parameter. However, they only controlled one hyper-parameter, which is not that efficient to optimize the whole performance of the neural network.

3. Reinforcement Learning Model

In our project, we use a deep Q network (DQN) to dynamically adjust the hyper-parameters in a fully-connected network (objective net), which is applied to conduct the task of image classification. And we expect, with the help of DQN, we can minimize the loss of objective net as small as possible in limited steps.

By using DQN, we should first define other typical components of any RL problem: states, actions and rewards. We accept the suggestions given by (**?**), and define them as follows:

3.1. Environment

The objective function of our work are

$$f(z_t) = -\frac{1}{n}\sum_{z_t}[ylna + (1-y)ln(1-a)] \quad (2)$$

$$g(z_t) = -\frac{1}{n}\sum_{z_t}[ylna + (1-y)ln(1-a) + \lambda z_t^T z_t] \quad (3)$$

where (z_t, y) stands for one training example.

We planed to use f(x) to adjust the learning rate, and used g(x) to adjust regularization parameter. The environment combines the objective function and a set of actions and other needed elements, formulated as a MDP model. We figure out that if we include the initial and current DQN weight, the proceeding learning rates and the descent directions in the states, the Markov condition would be satisfied.

3.2. State

We used a feature vector with six state features to represent the state. The state features are the hyper-parameter we intend to adjust, candidate iterate objective value, max objective from the past M steps and the dot product between the descent direction d_t and gradient. The first four features are straight forward so we just discuss the last two features below.

Algorithm 1 Q-gradient Descent for Learning Rate

Input: initial iterate x_1, initial learning rate α_c, trained DQN $Q(s;\theta)$, number of time steps T
Set $\bar{x} = x_1, d(\bar{x}) = -\nabla f(x_1), \alpha_1 = \alpha_c$,
for $t = 1, ..., T$ **do**
  Compute state feature vector s_t
  $a_t = argmax_a[Q(s_t;\theta)]_a$
  if $a_t = a_{half}$ **then**
    $\alpha_{t+1} = \frac{1}{2}\alpha_t$
  else if $a_t = a_{accept}$ **then**
    $\bar{x} = x_t, d(\bar{x}) = -\nabla f(\bar{x}), \alpha_{t+1} = \alpha_c$
  end if
  $x_{t+1} = \bar{x} + \alpha_{t+1} d(\bar{x})$
end for
return $x^* = x_T$

Let F^{t-1} be a list of the M lowest objective values obtained up to time t-1, the state encoding is given by

$$[s_t] = \begin{cases} 1 & f(x_t) \le min(F_M^{t-1}) \\ 0 & min(F_M^{t-1}) < f(x_t) \le max(F_M^{t-1}) \\ -1 & otherwise \end{cases} \quad (4)$$

Let d_t be the descent direction,

$$d(\bar{x}) = -\nabla f(\bar{x}) \quad (5)$$

Then the state alignment is given by

$$[s_t]_{alignment} = \frac{1}{n}\sum_{i=1}^{n} sign([d_t]_i [dt-1]_i) \quad (6)$$

Besides, in order to make the state features independent of the specific object function, all of the features are transformed to be in the interval [-1, 1].

3.3. Action

For a given state, the actions are the combinations of how we want to change the learning rate as well as regularization parameter. Generally, learning rate and regularization parameter are very small. Thus, we used the strategy in which the learning rate or regularization parameter is reset to an initial value after accepting an iterate.

Therefore, when we control the learning rate, there are two actions: maintain the learning rate or half the

Algorithm 2 Q-gradient Descent for Regularization Coefficient

Input: initial iterate x_1, initial $\lambda = \lambda_c$, trained DQN $Q(s;\theta)$, number of time steps T, initial learning rate α
Set $\bar{x} = x_1, d(\bar{x}) = -\nabla g_\lambda(x_1), \lambda_1 = \lambda_c$,
for $t = 1, ..., T$ **do**
  Compute state feature vector s_t
  $a_t = argmax_a[Q(s_t;\theta)]_a$
  if $a_t = a_{half}$ **then**
    $\lambda_{t+1} = \frac{1}{2}\lambda_t$
  else if $a_t = a_{1.25}$ **then**
    $\lambda_{t+1} = 1.25\lambda_t$
  else if $a_t = a_{accept}$ **then**
    $\bar{x} = x_t, d(\bar{x}) = -\nabla g_{\lambda_t}(\bar{x}), \lambda_{t+1} = \lambda_c$
  end if
  $x_{t+1} = \bar{x} + \alpha d(\bar{x})$
end for
return $x^* = x_T$

learning rate. As for adjusting regularization coefficient, besides the two choices for learning rate, we also permit it to increase by quarter.

3.4. Reward

For adjusting learning rate, the reward is defined as the inverse distance from objective net training loss to a lower bound.

$$r_{id}(f, x_t) = \frac{c}{f(x_t) - f_{lb}}, c > 0, f_{lb} < f(x) \quad \forall x \tag{7}$$

However, as regularization is introduced to prevent model from overfitting, which means it is validation loss that should be the metrics to affect the adjustment of regularization coefficient, so we use the same form of reward function as adjusting learning rate with only replacement of training loss into error rate.

$$r_{id}(h, x_t) = \frac{c}{h(x_t) - h_{lb}}, c > 0, h_{lb} < h(x) \quad \forall x \tag{8}$$

where h(x) is

$$h(x) = \frac{1}{n}\sum_{z_v}(I_{pred=y})) \tag{9}$$

where *pred* is the prediction of objective net on validation set (z_v, y).

4. DQN Training

In order to make things less complicated, we try to train two separate DQNs to adjust learning rate and regularization coefficient respectively. And for both tasks, the dataset for our objective nets is CIFAR-10 dataset, which contains 50000 training examples as well as 10000 test examples.

4.1. Experience Replay

In the training part, we used the trick of experience replay.

An experience consists of a $(s_i, a_i, r_{i+1}, s_{i+1},$ label$)^j$ tuple for some episode $j \in [1, e]$ at time step i $\in$ [M-1, T]. Here we add a label into the tuple to make it easier to use.

These tuples are stored in a memory of experiences E. Instead of updating the DQN with only the most recent experience, a subset S $\in$ E are drawn from memory and used as a mini-batch to update the DQN:

$$\theta = \theta - \frac{\beta}{|S|}\sum_{(s_i,a_i,r_{i+1},s_{i+1},label)^j \in S}(\hat{y} - y_i)\nabla_\theta Q(s_i;\theta) \tag{10}$$

The subset sample is formed by randomly drawn experience from the A most recent episodes and top B best episodes. Adding both recent and best episodes to the mini-batch helps prevent the DQN from over learning during a particular time and episode.

4.2. Training Parameters Initialization

We follow (**?**) to implement Q-Learning with experience replay to train our DQN. At the beginning, our DQN has two hidden layers plus input layer and output later ($6 \times 32 \times 16 \times 2$), and the architecture of our objective net is $3072 \times 100 \times 50 \times 10$. All activation function in both nets are sigmoid function except for the output layer, where we use identify activation for DQN and softmax function for objective net. On the other hand, loss functions for our two nets are different, L2 loss is used for DQN whereas cross entropy loss for objective net.

Based on above architectures, we then started to train our model, and soon we found it would take too much time to train. It takes about 20 minutes per episode (according to the (**?**) every episode includes 1000 steps). As training processes, the time needed by one episode continues increasing. This situation undoubtedly makes it impossible for us to tune the parameters for our model, such as the learning rate for DQN.

Algorithm 3 Q-Learning with Experience Replay for α

Objective Parameters: f, f_{lb}, x_1, α_c, M, T
Q-Learning Parameters: E, θ_0, γ, ϵ, c_1, c_2, β
$\theta = \theta_0$
for $e = 1, ..., E$ **do**
  for $t = 1, ..., M-1$ **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, ..., T do **do**
    Generate state feature vector s_t
    Choose action a_t according to ϵ-greedy policy
    if $a_t = a_{half}$ **then**
      $\alpha_{t+1} = \frac{1}{2}\alpha_t$
    else if $a_t = a_{accept}$ **then**
      $\bar{x} = x_t, d(\bar{x}) = -\nabla f(\bar{x}), \alpha_{t+1} = \alpha_c$
    end if
    $x_{t+1} = \bar{x} + \alpha_{t+1} d(\bar{x})$
    Generate state feature vector s_{t+1}
    if $a_t \neq a_{accept}$ **then**
      $r_{t+1} = c_1/(f(x_{t+1}) - f_{lb})$
    else if $a = a_{accept}$ **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 ϵ

    Sample S $\in \epsilon$ and update θ via ()
  end for
end for
return θ

Algorithm 4 Q-Learning with Experience Replay for λ

Objective Parameters: g, g_{lb}, x_1, λ_c, M, T, α
Q-Learning Parameters: E, θ_0, γ, ϵ, c_1, c_2, β
$\theta = \theta_0$
for $e = 1, ..., E$ **do**
  for $t = 1, ..., M-1$ **do**
    $x_{t+1} = x_t - \alpha \nabla g_{\lambda_c}(x_t)$
  end for
  set $\bar{x} = x_M, d(\bar{x}) = -\nabla g_{\lambda_M}(x_M), \lambda_M = \lambda_c$
  for t=M, ..., T do **do**
    Generate state feature vector s_t
    Choose action a_t according to ϵ-greedy policy
    if $a_t = a_{half}$ **then**
      $\lambda_{t+1} = \frac{1}{2}\lambda_t$
    else if $a_t = a_{1.25}$ **then**
      $\lambda_{t+1} = 1.25\lambda_t$
    else if $a_t = a_{accept}$ **then**
      $\bar{x} = x_t, d(\bar{x}) = -\nabla g_{\lambda_t}(\bar{x}), \lambda_{t+1} = \lambda_c$
    end if
    $x_{t+1} = \bar{x} + \alpha d(\bar{x})$
    Generate state feature vector s_{t+1}
    if $a_t \neq a_{accept}$ **then**
      $r_{t+1} = c_1/(g_{\lambda_t}(x_{t+1}) - g_{lb})$
    else if $a = a_{accept}$ **then**
      $r_{t+1} = c_2/(g_{\lambda_t}(\bar{x}) - g_{lb})$
    end if
    Add experience $(s_t, a_t, r_{t+1}, s_{t+1})^e$ to memory ϵ

    Sample S $\in \epsilon$ and update θ via ()
  end for
end for
return θ

In order to reduce the cost of time and make the model trainable, we redesigned the experiments. There are two main alterations. First, we reduce the complexity of both DQN and objective net by setting just one hidden layer between input and output layer. Then we set every episode include just 100 steps. For the first alteration, some upper bound may needed for the ability of our DQN to accurately fit the Q(s, a) function. However, we considered it is acceptable compared with the benefit it brings. For the second alteration, though cutting down the steps would not assure the objective net converge at the end of each episode, it does make sense if we set the baseline to be the accuracy after 100 steps' updates of objective net by original gradient descent method and then compare it with the accuracy produced by Q-gradient descent method under the same constraints. After these changes, 10 to 15 episodes could be trained within the first 20 minutes. Following are the initial values for other vital parameters of the networks.

Learning rate for objective net $\alpha_1 = 1$; learning rate for DQN $\alpha_2 = 0.05$; training episodes for DQN *episodes*= 200; experience replay memory $A = 1000$; training batch size for DQN *batch size* = 64; ϵ for ϵ-greedy policy $\epsilon = 1$ (ϵ will decrease to 0.1 within the first 50 episodes).

Training method for both nets are original gradient descent method.

4.3. Training Result

Under the initial condition, the loss of DQN keeps oscillating all the time and does not show any sign of convergence, as shown in figure 1.

There are two reasons that may lead to the unsteady performance of DQN loss. One is the learning rate of DQN and the other is the ϵ value. Typically, when learning rate is not properly set (often too large) at the beginning, the loss will often oscillate (sometimes

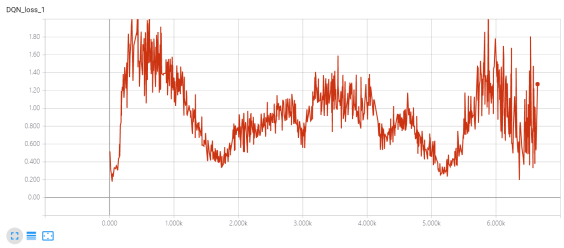

Figure 1. Loss of DQN, learning rate $\alpha = 0.05$, ϵ decreases from 1 to 0.1 within the first 50 episodes

even diverge) because it always overshoots the optimal points thus never converges. In terms of ϵ, if it is not small enough, the agent (DQN) will explore more rather than exploit the acquired information to do updates for each step, which makes the DQN unstable.

Therefore, we first tested different learning rate for DQN, including 0.025, 0.01 and 0.005. At this time, we chose Adam gradient descent method to train DQN. However, unfortunately, none of them seems to converge during training time.

Then we kept learning rate at 0.05 and decrease ϵ to fixed 0.01 for every episode. This time, we observe the convergence of the DQN loss, as shown in Figure 2.

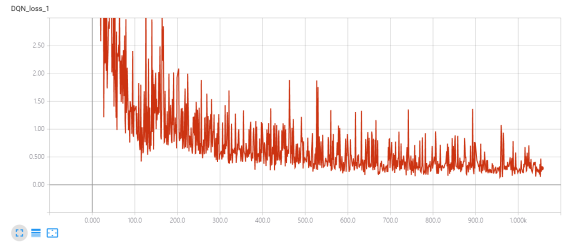

Figure 2. Loss of DQN, learning rate $\alpha = 0.05$, $\epsilon = 0.01$

However, we also observe that after several episodes, the loss and test accuracy of the objective net appear to be the same all the time, which means that at every step the DQN selects the same action to adjust learning rate. This pattern is consistent with agent choosing actions without any exploration, which indicates that we should use a larger ϵ.

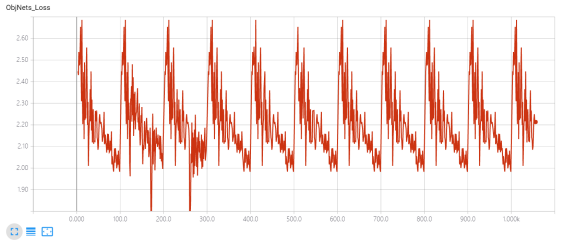

Figure 3. Loss of Objective Net in Multiple Episodes, $\epsilon = 0.01$

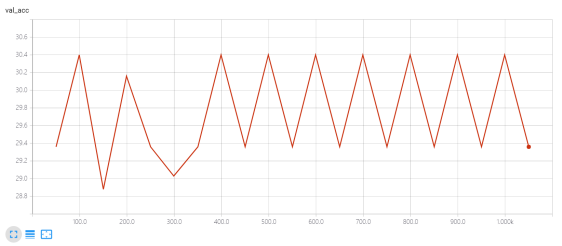

Figure 4. Accuracy of Objective Net, $\epsilon = 0.01$

After realizing epsilon causes the oscillation, we do some explorations of its value, such as fixed 0.05, decreasing from 0.5 to 0.05 within first 10 or 15 or 20 episodes and so on.

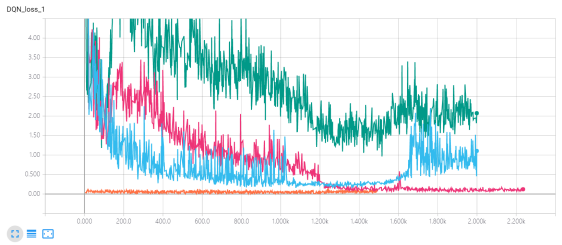

Figure 5. Loss of DQN with Different ϵ Setting

In a short term (around 20 episodes, it is not efficient to wait until the end for switching to different values), some trials do converge, but some may firstly converge and then diverge. And even for converged trials, they do not converge at a steady level.

Given all the trials above, we finally decide ϵ to decrease from 0.2 to 0.05 in the first 10 episodes, and then train DQN for 8 hours (only 100 episodes) considering the time cost. However, even at the very beginning, we found the loss following some convergence

trajectory similar to Figure 2, but finally it explodes to very large numbers.

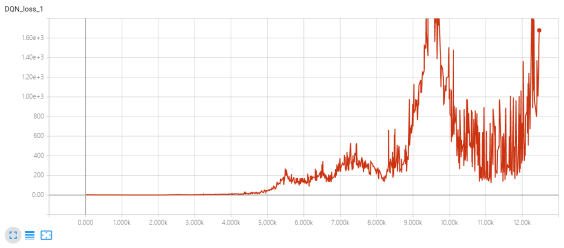

Figure 6. Loss of DQN, learning rate $\alpha = 0.05$, ϵ decreases from 0.2 to 0.05 within the first 10 episodes

After all the experiments, we still could not find a well-converged DQN. However, during the training process, we actually saved some models at likely converged point and some other points. We extracted them and applied them as input to Q-gradient descent method. There are three types of adjusting patterns shown during experiments, shown as Figure 7, Figure 8 and Figure 9. One of them happens to be consistent with the result in the (**?**).

4.4. Q-gradient Descent Result

The first type of pattern adjusts learning rate to be a small value at the beginning, then reset it for the rest of steps.

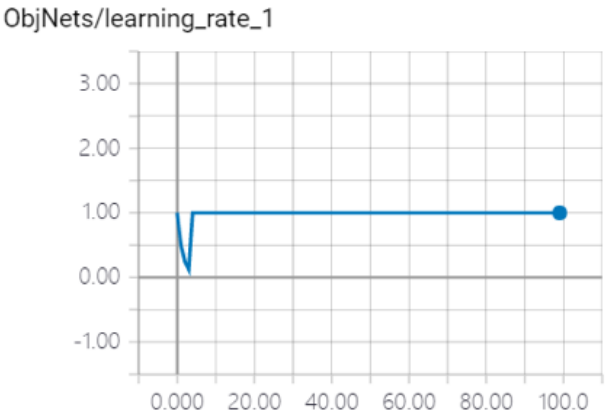

Figure 7. Adjusting Pattern 1

The second type of pattern has a continuous decreasing learning rate, and decreases to almost zero.

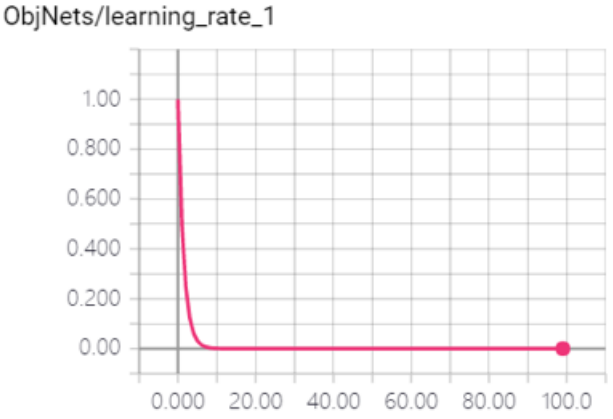

Figure 8. Adjusting Pattern 2

The last type decreases first and then reset the learning rate, and only when it is close to the final steps, it decreases again.

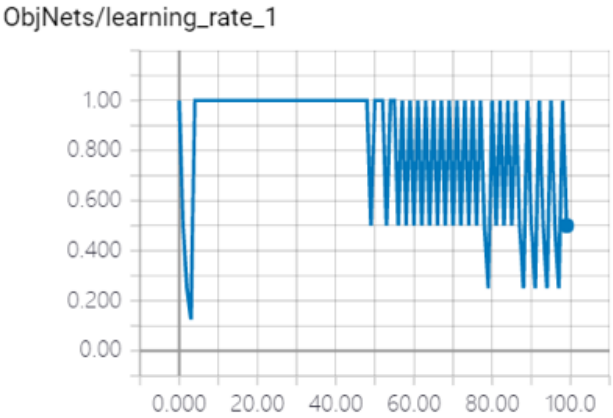

Figure 9. Adjusting Pattern 3

Obviously, the first two types of patterns cannot achieve higher accuracy than original gradient descent. However, as we mentioned above, the third pattern is similar to the result in (**?**). More importantly, it can achieve around 40% accuracy within 100 steps, which is 10% higher than the baseline. The result is shown in Figure 10 and Figure 11.

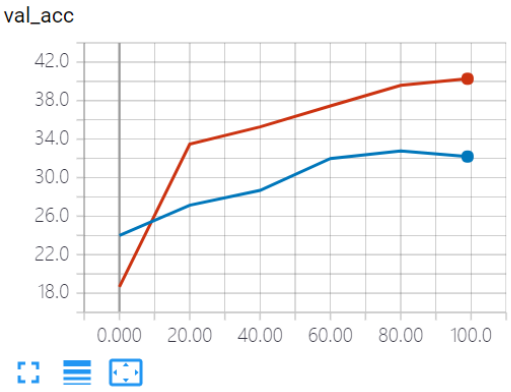

Figure 10. Accuracy of Objective Net (red line: using Q-gradient descent, blue line: using original gradient descent)

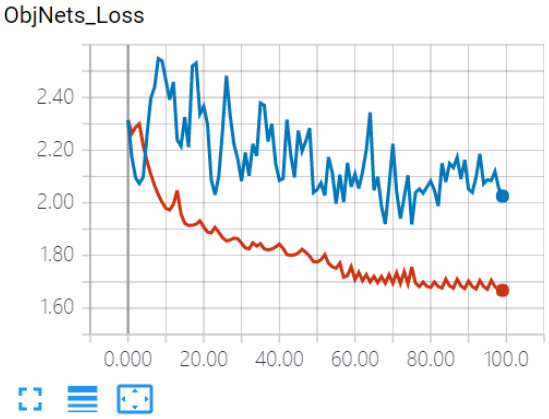

Figure 11. Loss of Objective Net (red line: using Q-gradient descent, blue line: using original gradient descent)

4.5. DQN for Adjusting Regularization Coefficient

We chose L2 norm as the regularization item for objective net. In order to make the objective net be able to overfit in just 100 iterations for the observation of the effect of the regularization, only 1,000 examples out of 50,000 training examples are used to train objective net. As for DQN, we modify the reward function as we mentioned in 3.4, and add one more action at its output layer. As we explore a lot on previous DQN, which is very time consuming, considering adjusting regularization coefficient is similar to adjusting the learning rate, we just trained it to get a glimpse of the result. Unfortunately, it also showed no sign of convergence as before. As shown in Figure 12.

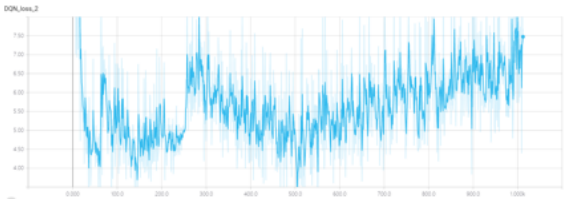

Figure 12. Loss of DQN for Adjusting Regularization Coefficient

5. Conclusion

Unfortunately, our DQN does not converge in limited time. However, we accidentally get a model that behaves as expected, which is just a random result. After analyzing the experiments, we believe there are two main factors that cause the divergence of our model.

The first reason is the training episodes. In (**?**), the author trains their neural net for 150K episodes with 1,000 steps per episode. However, we are not available for more computational power. It is impossible for us to train our net in such a long run. In addition, we actually printed the average reward for each episode during training process, but their values also oscillate or even decrease within our training episodes. Perhaps when the DQN is fed with more data and trained for more episodes, it can converge and find the deterministic relation between defined state and its Q(s, a) value.

The other reason is the value of ϵ. We finally decide to decay ϵ in the first 10 episodes from 0.2 to 0.05. The decay rate is what balances the exploration and exploitation of the model. Because we can only train DQN for limited episodes, we either choose to initialize ϵ as big as 1 and then decrease it steeply, or initialize it as a small number and then decrease it smoothly. Based on the understanding of (**?**), we choose the latter.

In conclusion, in this project, we tried to implement DQN as a controller for the hyper-parameters of a neural networks. However, due to our limited training episodes, we can not get a stable DQN. The main reason may be that we do not train our model for enough episodes. As long as we can get more computation power, we will be able to explore more about it.

References

Andrychowicz M. et al. Learning to learn by gradient descent by gradient descent. In *Proceedings of the Conference on Neural Information Processing Systems*, pp. 3981–3989, Barcelona, Spain, 2016.

Barret et al. Neural architecture search with reinforcement learining. *Under review as a conference paper at ICLR 2017*, 2017.

Bergstra et al. Algorithms for hyper-parameter optimization. *Advances in Neural Information Processing Systems*, (2546-2554), 2011.

Bergstra et al. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(281-305), 2012.

Hansen, Samantha. Using deep q-learning to control optimization hyperparameters. Technical report, IBM T.J. Watson Research Center, 2016.

Long-Ji, Lin. Reinforcement learning for robots using neural networks. Technical report, 1993.

Riedmiller, M. Neural fitted q iteration-first experiences with a data efficient neural reinforcement learning method. In *Proceedings of the Conference on Machine Learning*, pp. 317–328, Berlin, German, 2005.

Snoek et al. Scalable bayesian optimization using deep neural networks. *ICML*, (2171-2180), 2015.

Swersky et al. Multi- task bayesian optimization. *Advances in Neural In- formation Processing Systems*, (2004-2012), 2013.

Tesauro, G. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(215-219), 1994.

Victor V. et al. Global search in combinatorial optimization using reinforcement learning algorithms. In *Proceedings of the Congress on Evolutionary Computation*, volume 1, pp. 189–196, 1999.

Contribution

Each one make $\frac{1}{3}$ contribution of the work, including literature review, coding, training model and report.