ENEE662 Project Report
Machine Learning with Differentiable optimization layer

Yuxin Miao
ymiao@terpmail.umd.edu

Yanjun Fu
yanjunfu@umd.edu

Ruiqi Xian
rxian@umd.edu

Abstract

In machine learning area, convolutional neural networks are widely used especially in computer vision. Through forward and backward propagations, images could be recognized and classified after training. Recent work has shown how to embed differentiable layers in deep learning architecture.These layers encode constraints and complex dependencies between the hidden states that traditional convolutional and fully-connected layers often cannot capture. We add differentiable layer on the basis of original convolutional layers and take MNIST as training as well as testing dataset. Based on Lenet [2], we explore multiple optimization methods, such as adaptive gradient descent[5], Adadelta[15] and adaptive moment estimation[8]. Furthermore, application scenarios has been extended to Reinforcement Learning. We also implement a famous Reinforcement Learning model DQN [11] and check whether optimization layers can enhance its representation ability.

1 Introduction

Current, machine learning (ML) is a power tool for a wide range of problems, such as image classification [9, 6], playing the game of Go [12], and playing the Atari [11]. In general, ML is a optimization problem. Assume we have some experience E and a performance measure P with respect to a task T. ML learns from experience E and maximize the performance P on this task T.

Taking advantage of the development of GPUs and very large scale datasets, such as ImageNet [4], the deep learning (DL) models becomes the new paradigm of machine learning. Normally, the DL model is to minimize a loss function, which in most cases in either convex or concave, such as the cross entropy loss for image classification. We can view a deep learning model as a function approximator which approximate the relationship between the inputs, such as images, and the target outputs, such as the labels for each image. Although the loss function is generally convex or concave, the deep neural networks (DNN) itself have very high dimensions and is neither convex nor concave [7]. Therefore, in this project, we do not try to discuss the convexity of deep learning models. Instead, we try to discuss how to integrate convex optimization into the current DNNs.

Normally, current DNN uses simple although non-linear function as activation of the output of the last layer, which will be feed forward to the next layer. To introduce more complicated and more powerful depiction of the dependence between two layers, we utilize the differentiable optimization layers, which is explored by several existing literature [1, 2]. Besides utilizing optimization layers in supervised learning, such as image classification, we also explore the application of optimization layers in Reinforcement Learning settings, which is a sequence decision learning process.

Moreover, the optimization methods (algorithms) is crucial for solving a optimization problem, including deep learning. The stochastic gradient descent (SGD) is the mostly used optimization methods in deep learning, especially image classification. We also explore how different optimization methods including Adam [8], Adamgrad [5], Adamdelta [15] can influence the speed of convergence

and the performance of the DNN. The implementation details and the corresponding experimental results are shown in section 5 and section 6.

2 Preliminaries

2.1 Deep Learning

Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction. These methods have dramatically improved the state-of-the-art in speech recognition, visual object recognition, object detection and many other domains such as drug discovery and genomics. Deep learning discovers intricate structure in large data sets by using the backpropagation algorithm to indicate how a machine should change its internal parameters that are used to compute the representation in each layer from the representation in the previous layer. Deep convolutional nets have brought about breakthroughs in processing images, video, speech and audio, whereas recurrent nets have shone light on sequential data such as text and speech. Deep learning is part of a broader family of machine learning methods based on artificial neural networks. Recent works have shown deep learning structures like ImageNet[4], AlexNet [9] and ResNet[6]. In our project, LeNet is adopted. The convolutional neural network we built includes two convolutional layers, two pooling layers, two fully connected layers and one differentiable layer.

The backpropagation procedure to compute the gradient of an objective function with respect to the weights of a multi-layer stack of modules is nothing more than a practical application of the chain rule for derivatives. The key insight is that the derivative (or gradient) of the objective with respect to the input of a module can be computed by working backwards from the gradient with respect to the output of that module (or the input of the subsequent module) (Fig. 1). The backpropagation equation can be applied repeatedly to propagate gradients through all modules, starting from the output at the top (where the network produces its prediction) all the way to the bottom (where the external input is fed). Once these gradients have been computed, it is straightforward to compute the gradients with respect to the weights of each module.

2.2 Reinforcement Learning

Reinforcement learning (RL) [13] is a kind of machine learning that learns how to choose actions based on current situation to maximize a reward signal. The learner and decision maker is called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment. At each time-step t, the agent selects an action and the environment responds and presents new situations to the agent based on its choice. The interaction process is shown in Figure 1.

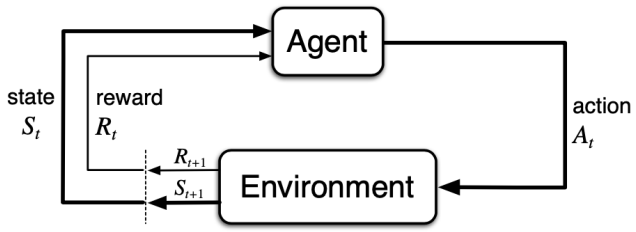

Figure 1: The agent–environment interaction [13].

Markov Decision Process Normally, the environment is stated in Markov decision process (MDP). Consider a MDP that is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, any state S_t in this MDP is Markov, *i.e.*,

$$P\left[S_{t+1} \mid S_t\right] = P\left[S_{t+1} \mid S_1, \cdots, S_t\right].$$

The transition probability between two states after taking certain action a is defined by

$$\mathcal{P}_{ss'}^{a} = P\left[S_{t+1} = s' \mid S_t = s, A_t = a\right].$$

The reward function of taking certain action a at the state s is defined by

$$\mathcal{R}_{s}^{a} = \mathbb{E}\left[R_{t+1} \mid S_t = s, A_t = a\right].$$

The return G_t is the total discount reward from time-step t,

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

where $\gamma \in [0, 1]$ is the discount factor. In this decision process, a policy π is a distribution over actions given states,

$$\pi(a \mid s) = P\left[A_t = a \mid S_t = s\right],$$

which fully defined the behaviour of an agent. The action-value function $q_\pi(s, a)$ is the expected return starting from state s, taking action a, and then following policy π,

$$q_\pi(s, a) = \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right].$$

The goal of RL is to find one of the optimal policies π_* that achieve the optimal action value function

$$q_{\pi_*}(s, a) = q_*(s, a) = \max_\pi q_\pi(s, a).$$

An optimal policy can be defined as

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a \in \mathcal{A}} q_*(s, a), \\ 0 & \text{otherwise.} \end{cases}$$

Deep Q-learning The traditional Q-learning [14] estimated the action-value function by iteratively updating Q according to the bellman equation:

$$Q_{i+1}(s, a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q_i\left(s', a'\right) \mid s, a\right], \tag{1}$$

where $\mathcal{E}$ represents the environment. Ideally speaking, Q_i converges to Q_* when $i \to \infty$. However, this updating algorithm is not practical while confronting more complicated environments, such as playing computer games. The first drawback is that the action-value is estimated separately for each sequence, without any generalization. The second drawback is that maintain a table containing Q is impractical while the state-action space is very large even continuous. Therefore, we use a function-approximator, such as a deep neural network, to estimate the action-value function, and $Q(s, a; \theta) \approx Q^*(s, a)$ [10, 11]. The function approximator with trainable parameters θ is referred as a Q-network. The loss function we would like to minimize is

$$L_i\left(\theta_i\right) = \mathbb{E}_{s, a \sim P(\cdot)}\left[\left(y_i - Q\left(s, a; \theta_i\right)\right)^2\right], \tag{2}$$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q\left(s', a'; \theta_{i-1}\right) \mid s, a\right]$ is the target for iteration i and $P(s, a)$ is a probability distribution over sequences s and actions a that is called the behaviour distribution. Then we can have the following gradient derived from the loss function 2,

$$\nabla_{\theta_i} L_i\left(\theta_i\right) = \mathbb{E}_{s, a \sim P(\cdot); s' \sim \mathcal{E}}\left[\left(r + \gamma \max_{\rho'} Q\left(s', a'; \theta_{i-1}\right) - Q\left(s, a; \theta_i\right)\right) \nabla_{\theta_i} Q\left(s, a; \theta_i\right)\right]. \tag{3}$$

The procedure of Deep Q-learning [10] is shown in algorithm 1.

3 Differentiable Convex Optimization Layers

Current deep neural networks (DNN) can be view as some function approximators that can map the input to the target output. The typical layers in DNN, such as fully-connected layers and convolutional layers, conduct some linear matrix operations. To make the DNN a more powerful function approximator, non-linear activations are used between two different layers. The wide-used non-linear activation functions includes the ReLU:

$$\mathrm{ReLU}(x) = \max\{x, 0\}, \tag{4}$$

the sigmoid or logistic function:

$$\mathrm{sigmoid}(x) = \left(1 + e^{-x}\right)^{-1}, \tag{5}$$

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory D to capacity N
Initialize Q-network Q with random parameters
for episode $= 1, M$ **do**
  Initialize sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q_* \left(\phi\left(s_t\right), a; \theta\right)$
    Execute action a_t in emulator and observe reward rt and image x_{t+1}
    Set $s_{t+1} = s_t$, a_t, x_{t+1} and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q\left(\phi_{j+1}, a'; \theta\right) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient decent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ according to equation 3
  end for
end for

and the softmax function:

$$\operatorname{softmax}(x)_j = e^{x_j} / \sum_i e^{x_i}. \tag{6}$$

In the traditional feedforward networks, the output of each layer is the simple non-linear function of the previous layer. To enhance the representation ability of DNNs, besides classical non-linear activation functions, we utilize the differentiable optimization layer [2, 1] to model the more complicated relationship between two adjacent DNN layers. The optimization layers cast more constraints between two layers, and the output of the $i+1$th layer in a network is the solution to a constrained optimization problem based upon previous layers.

3.1 Quadratic Optimization Layer

We consider a rather simple optimization layer that is to solve a quadratic program (QP),

$$\begin{array}{rl} z_{i+1} = \underset{z}{\arg\min} & \frac{1}{2} z^T Q\left(z_i\right) z + p\left(z_i\right)^T z \\ \text{subject to} & A\left(z_i\right) z = b\left(z_i\right) \\ & G\left(z_i\right) z \preceq h\left(z_i\right), \end{array} \tag{7}$$

where z is the optimization variable, $Q(z_i) \succeq 0$, $p(z_i)$, $A(z_i)$, $b(z_i)$, $G(z_i)$, $h(z_i)$ are parameters of the optimization problem, and z_i is the output of the previous layer. The parameters Q, p, A, G and h depend in any differentiable way on the previous layer z_i [2]. Therefore, these parameters can be optimized like other parameters in the network. Training a DNN requires back propagation, which further requires us to compute derivative of the solution of the QP with respect to the parameters that depend on the last layer. For parameters that depend on the output of the last layer z_i, we can further compute the derivative of z with respect to z_i and the parameters of the previous layers through the chain rule. The Lagragian of QP 7 is given by

$$L(z, \nu, \lambda) = \frac{1}{2} z^T Q z + p^T z + \lambda^T (Gz - h) + \nu^T (Az - b), \tag{8}$$

where λ is the dual variable for inequality constraints and ν is the dual variable for equality constraints. The KKT conditions for the QP implies that

$$\begin{aligned} Qz^\star + p + A^T \nu^\star + G^T \lambda^\star &= 0 \\ Az^\star - b &= 0 \\ \mathbf{diag}\left(\lambda^\star\right)\left(Gz^\star - h\right) &= 0, \end{aligned} \tag{9}$$

where $z^\star$ is the optimal solution for the primal problem and $\lambda^\star$ and $\nu^\star$ is the optimal solutions for the dual problem. Taking the differentials of these conditions, and we have

$$\begin{aligned} \mathrm{d}Qz^\star + Q\,\mathrm{d}z + \mathrm{d}p + \mathrm{d}A^T \nu^\star + A^T\,\mathrm{d}\nu + \mathrm{d}G^T \lambda^\star + G^T\,\mathrm{d}\lambda &= 0 \\ \mathrm{d}Az^\star + A\,\mathrm{d}z - \mathrm{d}b &= 0 \\ \mathbf{diag}\left(Gz^\star - h\right)\mathrm{d}\lambda + D\left(\lambda^\star\right)\left(\mathrm{d}Gz^\star + G\,\mathrm{d}z - \mathrm{d}.h\right) &= 0 \end{aligned} \tag{10}$$

In the backpropagation, for a backward pass vector $\frac{\partial \ell}{\partial z^\star} \in \mathbb{R}^n$, we have

$$\begin{bmatrix} d_z \\ d_\lambda \\ d_\nu \end{bmatrix} = - \begin{bmatrix} Q & G^T \operatorname{\mathbf{diag}}(\lambda^\star) & A^T \\ G & \operatorname{\mathbf{diag}}(Gz^\star - h) & 0 \\ A & 0 & 0 \end{bmatrix}^{-1} \begin{bmatrix} \left(\frac{\partial \ell}{\partial z^\star}\right)^T \\ 0 \\ 0 \end{bmatrix}. \tag{11}$$

Then the relevant gradients with respect to the parameters of QP is

$$\begin{aligned} \nabla_Q \ell &= \frac{1}{2}\left(d_z z^{\star T} + z^\star d_z^T\right) & \nabla_p \ell &= d_z \\ \nabla_A \ell &= d_\nu z^{\star T} + \nu^\star d_z^T & \nabla_b \ell &= -d_\nu \\ \nabla_G \ell &= \operatorname{\mathbf{diag}}(\lambda^\star)\, d_\lambda z^{\star T} + \lambda^\star d_z^T & \nabla_h \ell &= -\operatorname{\mathbf{diag}}(\lambda^\star)\, d_\lambda. \end{aligned} \tag{12}$$

Since the parameters are optimized iteratively by the optimizer of the DNN, sometimes the QP 7 is not feasible. This require us to cleverly initialize the parameters so that we can make sure the QP is feasible during the training process. For instance, we initialize the parameter Q as

$$Q = LL^T + \epsilon I, \tag{13}$$

where L is a lower-triangular matrix and we optimize L instead of Q. In this case, we can always make sure Q is semi-positive definite during the training process. We form h as

$$h = Gz_0 + s_0$$

for some learnable z_0 and s_0.

Besides the QP, we can use different kinds of convex optimization problem to form the optimization layer such as the Linear Program (LP). Existing work [1] suggests that the optimization layer can be formed as any convex cone program that is

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax \preceq_{\mathcal{K}} b, \end{array} \tag{14}$$

where $\mathcal{K}$ is a proper cone.

3.2 Non-linear Activation Functions and Convex Optimization

Why we use differentiable optimization layer to enhance the representation ability of the DNN? An insight is that the classical non-linear activation functions including ReLU, sigmoid and softmax can be regarded as the special cases of convex optimization problems [1]. By utilizing new constraints and convex optimization problems, we actually create new non-linear "activation functions". The ReLU that is shown in 4 can be reformulated as s projecting a point $x \in \mathbf{R}^n$ onto the non-negative orthant as

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\|x - y\|_2^2 \\ \text{subject to} & y \geq 0. \end{array} \tag{15}$$

The sigmoid function can be reformulated as projecting a point $x \in \mathbf{R}^n$ onto the interior of the unit hypercube as

$$\begin{array}{ll} \text{minimize} & -x^\top y - H_b(y) \\ \text{subject to} & 0 < y < 1 \\ & 1^\top y = 1, \end{array} \tag{16}$$

where $H_b(y) = -\left(\sum_i y_i \log y_i + (1 - y_i)\log(1 - y_i)\right)$. The softmax function can be reformulated as projecting a point $x \in \mathbf{R}^n$ onto the interior of the $(n-1)$-simplex $\Delta_{n-1} = \left\{p \in \mathbf{R}^n \mid 1^\top p = 1 \text{ and } p \geq 0\right\}$ as

$$\begin{array}{ll} \text{minimize} & -x^\top y - H(y) \\ \text{subject to} & 0 < y < 1 \\ & 1^\top y = 1, \end{array} \tag{17}$$

where $H(y) = -\sum_i y_i \log y_i$

4 Optimization Methods

Since we are doing a multi-class classification task, MSE may significantly lower down the performance of the whole network. So we use cross-entropy as our loss function, that is, the loss function is as follows:

$$J(\theta) = -\sum_{i=1}^{N}\sum_{j=1}^{K} y_{ij}\log(h_{\theta(x_i)_j}) + (1-y_{ij})\log(1-h_\theta(x_i)_j)$$

where the prediction $h_\theta(x_i)$ will be given by $\mathrm{softmax}(Wx_i + b)$ which will have the value between 0 and 1, representing the probability of x_i belonging to specific class, and $y_{ij} \in 0, 1$ represents the true class label of i-th example.
The main problem for the multi-class classification task is to minimize the training loss and testing loss, so that more correct labeled images there will be.

4.1 Adaptive Gradient Descent

Adaptive gradient descent method (Adagrad)[5] is an algorithm for gradient-based optimization. Unlike stochastic gradient descent (SGD), which has fixed learning rate for all the parameters, Adagrad adapts the learning rate for different parameters respectively. It performs smaller updates for parameters that correspond to frequently occurred features, and larger updates for parameters that associated with infrequent features. It eliminates the need to manually tune the learning rate and improve the robustness of normal SGD methods.
In Adagrad, the equation for the parameter update is as follows:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\epsilon I + \mathbf{diag}(G_t)}}.g_t$$

where θ is the vector of parameters, η is the initial learning rate, ϵ is the small singular used to avoid the division of zero, I is the identity matrix, g_t is the gradient estimate in time step t that can obtained using the following equation.

$$g_t = \frac{1}{n}\sum_{i=1}^{n} \bigtriangledown_\theta J(x_i, y_i, \theta_t)$$

J is the loss function derived before.
The key of Adagrad is the matrix G_t, which is the sum of the outer product of the gradients until time step t:

$$G_t = \sum_{\tau=1}^{t} g_\tau g_\tau^T$$

The reason to use $\mathbf{diag}(G_t)$ instead of G_t in the update equation is that computing the square root of the full matrix is computationally expensive, especially in high dimension.
The main weakness of Adagrad is that the learning rate could be very small because of the square root of the gradients in the denominator. After hundreds of iterations, the algorithms will not be able to update the results.

4.2 ADADELTA

Adadelta[15] is an extension of Adagrad, which avoids the drawback of Adagrad. Instead of accumulate and store all the past squared gradients, the sum of gradients is defined as a decaying average of all past squared gradients.

$$\mathbb{E}[g^2] = \gamma\mathbb{E}[g^2]_{t-1} + (1-\gamma)g_t^2$$

γ is the parameter similar to momentum in SGD, which defines how the current state related to the past. Additionally, adadelta also defines another exponentially decaying average regards to the parameter updates:

$$\mathbb{E}[\triangle\theta_t^2] = \gamma\mathbb{E}[\triangle\theta^2]_{t-1} + (1-\gamma) \triangle \theta_t^2$$

The parameter update then becomes:

$$\triangle\theta_t = -\frac{\sqrt{\mathbb{E}[\triangle\theta^2]_{t-1} + \epsilon_1}}{\sqrt{\mathbb{E}[\triangle g^2]_{t-1} + \epsilon_2}}.g_t$$

And it is just leading to:

$$\triangle\theta_t = -\frac{\mathrm{RMS}[\triangle\theta]_{t-1}}{\mathrm{RMS}[g]_t}.g_t$$
$$\theta_{t+1} = \theta_t + \triangle\theta_t$$

4.3 Adaptive Moment Estimation

Adaptive moment estimation (Adam)[8] is another gradient based optimization method with adaptive learning rate for each parameter. In addition to use an exponentially decaying average of past squared gradient v_t, adam also stores an exponentially decaying average of past gradient m_t

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2$$

m_t and v_t are estimates of the mean and uncentered variance of the gradients. Since vectors of 0's are normally used for initial parameters, but m_t and v_t are biased towards zero, so additional step to correct the bias is needed:

$$\hat{m_t} = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v_t} = \frac{v_T}{1-\beta_2^t}$$

And then it yields the adam update equation:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v_t}} + \epsilon}\hat{m_t}$$

5 Implementation

5.1 Optimization Layers

In our implementation, we further simplify the QP 7. We choose the QP with the inequality constraints and eliminate the equality constraints. We make Q, G and h random initialized to be optimized later, and p depend on the output of the last layer z_i. We directly assign z_i to p. Therefore, the simplified QP is given by

$$\begin{aligned} z_{i+1} = \arg\min_{z} \quad & \tfrac{1}{2} z^T Q z + {z_i}^T z \\ \text{subject to} \quad & Gz \preceq h, \end{aligned} \tag{18}$$

where z_{i+1} is the output of the QP optimization layer.

We also implement a LP optimization layer for the DQN. We do not add LP optimization layer for deep learning because the training of a CNN for image classification is more time consuming. We formulate the LP optimization layer as

$$\begin{aligned} z_{i+1} = \arg\min_{z} \quad & {z_i}^T z \\ \text{subject to} \quad & Gz \preceq h. \end{aligned} \tag{19}$$

To see whether ReLU that is reformulated as a QP can work well for the DQN, we also implement the ReLU optimization layer based on equation 15. The implementations for different optimization layers can be integrated into the DNN, and they are compatible with the GPU acceleration framework and can be run in a batch fashion. Therefore, the speed of the optimization problem solvers is much faster than traditional solvers.

5.2 Deep Learning with Optimization Layers

MNIST Dataset In computer vision field, there are many open source datasets available, such as MNIST, CIFAR, KITTI, IMAGENET. Considering our limited hardware resources, MNIST was employed as our training and testing dataset. MNIST is a large dataset of hand-written digits, see Figure 2. It contains 60,000 training images and 10,000 testing images. Each image has 28x28 pixels, with grayscale levels.

Figure 2: MNIST dataset.

LeNet with QP optimization layer The convolutional neural network we built in this project is a basic LeNet, which has 2 convolutional layers, 2 pooling layers and 2 fully connected layers. See Figure3, we set 16 and 32 5x5 filters in both convolutional layers, which applied at stride 1, no padding. The max-pooling layers have 2x2 kernel applied at stride 2. Since the numbers of hidden parameters in the two fully connected layers are 512 and 50, which is not large compare to other CNN, we did not apply methods like L1/L2 regularization or dropout to avoid overfitting problem.

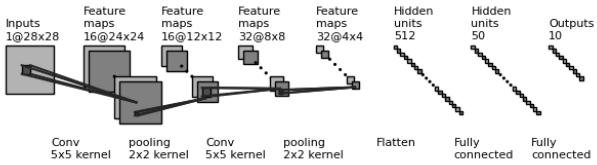

Figure 3: LeNet structure.

Instead of using activation functions like softmax, ReLU, and sigmoid, we used a QP optimization layer after 2 fully connected layers to get our outputs.

5.3 Deep Q-learning with Optimization Layers

OpenAI Gym Environment [3] OpenAI gym [3] is a toolkit for RL and provides a collection of environments including classic control and toy text which has small scale tasks. Due to the limited time and the computational resources, we choose the classical control task provided by OpenAI gym called Pole-Balancing [13], which is shown in figure 4. In this task, we apply forces to a cart moving along a track and try to keep the pole hinged to the cart from falling over. Every time the pole falls down or the cart is outside the track, the task fails and we reset the environment so that the cart is in the middle of the track and the pole is vertical. The reward here is $+1$ for each time step if the failure does not happen within the time-step. The goal of DQN is to maximize the reward, *i.e.*, the duration

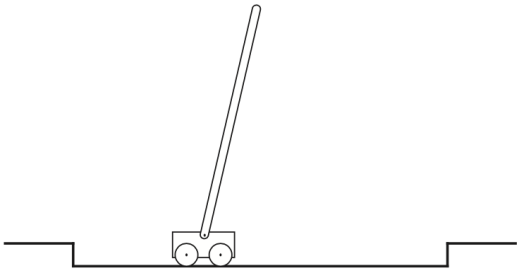

Figure 4: The Pole-Balancing task [13].

that the task does not fail. Instead of using the states provide by OpenAI gym, we extract a frame (an image) of time-step t as the state s_t. The action a_t is whether apply forces to the pole at time-step t.

Deep Q-learning with Convolutional Neural Network Since we uses a frame of time-step t as the state, we utilize a CNN as the function proximator, *i.e.*, the Q-network. In the Q-network, we implement three convolutional layers whose kernel sizes and strides are 5 and 2, respectively. Each convolutional layer is followed by a batch normalization. At the end, we implement a fully-connected layer. The activation function we use is ReLU. The structure of the Q-network is conv-bn-ReLU-conv-bn-ReLU-conv-bn-ReLU-fc, where conv represents a covolutional layer, bn represents a batch normalization and fc represents a fully-connected layer. The optimization method is RMSProp.

We also add QP optimization layer or LP optimization layer following the fully-connected layer to see the performance of QP optimization layer or LP optimization layer in DQN. In order to check whether the ReLU that is reformaulated as a QP can work well, we also replace the ReLU functions in Q-network with ReLU optimization layers.

6 Experiments

6.1 Deep learning with optimization layer

Optimization Methods For computer vision tasks, stochastic gradient descent (SGD) method is the mostly used optimization method. However, when we used LeNet structure with optimization layer as output layer, SGD was not giving an converged result. So, instead of using SGD, we implemented three different optimization methods: $Adagrad$,$Adadelta$,$Adam$ to see if the optimization layer would work and compare the performance of these three methods. Because of our limited computational resources, we only implement 40 epochs for each method. The result is shown in Figure5 and Figure6.

From the results, we can see that all the three methods could provide a converged result after approximately 30 epochs, which means the optimization layer could be adopted as the output layer in the CNN functioning as similar as $Sigmoid$,$ReLu$,$Softmax$.

From the two figures, we could also tell the performance of these three methods. The result of $Adagrad$ is not as good as other two methods. The loss for both training sets and testing sets are very high at first 10 epochs and still higher than other two results after convergence. As mentioned before, this is because of the square root of the gradients in the denominator in the update equation. This in turn causes the learning rate to shrink and eventually becomes very small, so the algorithms would not be able to get additional information from the inputs.

The loss for $Adam$ method is higher than $Adadelta$ at first 10 epochs. It is because the difference between parameters before and after updates are pretty high at the beginning, as we usually set initial parameters as vectors of 0's. So the past information could be a negative factor for the updates. After 30 epochs, both results converges to a similar point.

Optimization Layer To compare with the results using optimization layer, we also used $Adam$ trained a LeNet with Relu as the output layer, see Table1.All the results are obtained with a GTX 1070 6GB. The accuracy for two layers are very similar. The training loss and testing loss for ReLu is much lower than optimization layer. Because of complexity of the optimization layer, the time

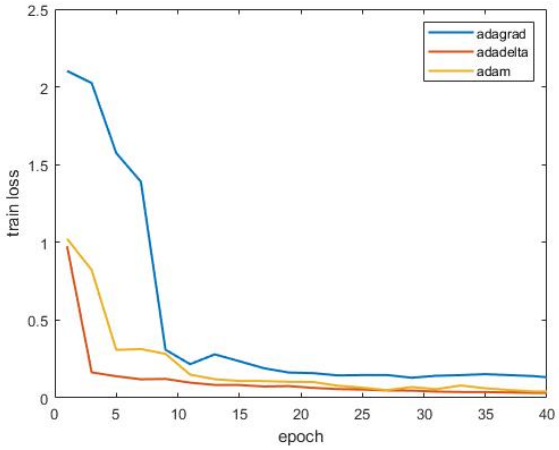

Figure 5: Training Loss.

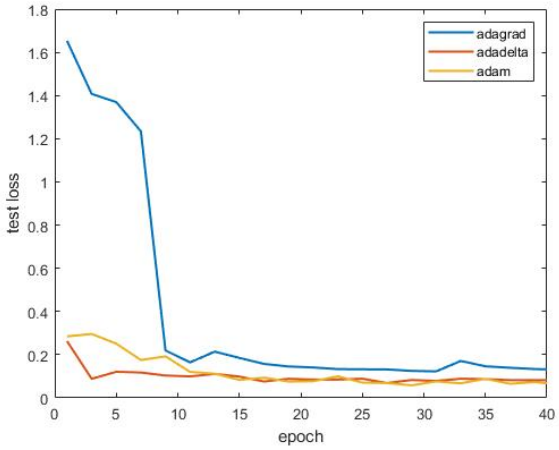

Figure 6: Testing Loss.

	Adagrad	Adadelta	Adam	Adam with ReLu
Train Loss	0.1326	0.0316	0.0410	0.0291
Test Loss	0.1320	0.0820	0.0674	0.0621
Accuracy	0.9721	0.9911	0.9871	0.9935
Time/s	5772	4800	4910	4410

Table 1: LeNet results table

consuming is also much higher than ReLu. However, the advantage of putting an optimization layer in the neural network structure is that we could add extra constraints to the neural network without losing the performance.

6.2 Deep Q-learning with Optimization Layers

QP and LP Optimization Layers We run the DQN, the DQN with a QP optimization layer, and the DQN with a LP optimization layer for 100 episodes, respectively, then we record the change of rewards for each framework and compare them. The result is shown in figure 7.

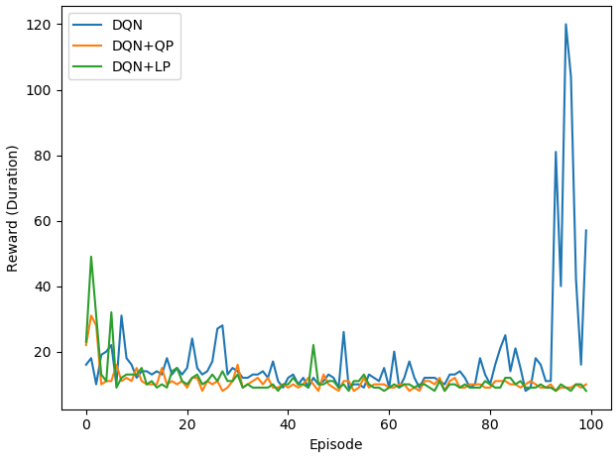

Figure 7: Rewards for the DQN, the DQN with with a QP optimization layer, and the DQN with a LP optimization layer.

We observe that the performance of DQN with QP optimization layers or LP optimization layers does not exceed the performance of DQN itself, and the performance of the DQN with a LP optimization layers is slightly better than the performance of the DQN with a QP optimization layer. The first possible reason of this phenomenon is that DQN, which is a kind of off-policy learning, often suffers from high variance and slow divergence during the training process [13], and the QP optimizaton layers and the LP optimization layers make the DQN even harder to train. The second possible reason is that we need further fine-tune the hyperparameters of the network and the optimization layers, which we did not finish due to the limited computational resources (GPUs).

ReLU optimization Layers We replaced the ReLU function used in DQN with the ReLU optimization layers we mentioned in equation 15 and compare the rewards of different implementations of ReLU. The result is shown in figure 8.

We observe that the two implementations of ReLU have comparable performance, though the peak of rewards of DQN is higher than it of the DQN with replaced ReLU. The extremely high peak is caused by the instability, *i.e.*, high variance of the training of the off-policy learning.

7 Discussion and Conclusion

For convolutional neural network with optimization layer, we could obtain very promising results using different optimization methods. However, compared to traditional activation functions like $ReLu$,$Softmax$, the optimization layer will cost more time because of its complexity. But with the ability to add more constraints, optimization layer could definitely be of great use for some problems. The next step could be simplifying the optimization layer or developing specific solver for this kind of problem, so that it could be competative with $ReLu$,$Sigmoid$ in terms of computational cost.

As for the RL scenarios, such as DQN, applying the optimization layers such as QP optimization layer and LP optimization layer does show performance improvement. This maybe

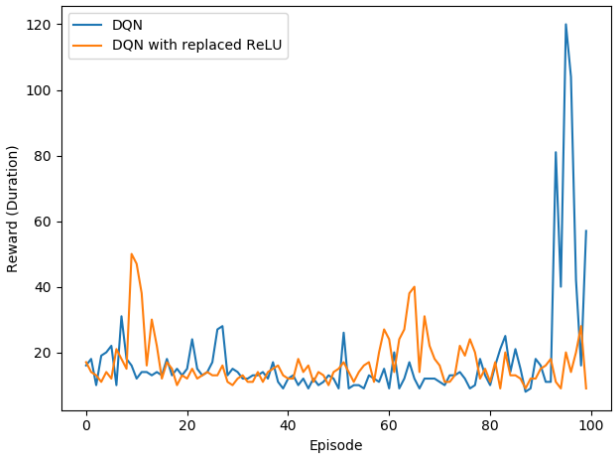

Figure 8: Rewards for the DQN and the DQN with with a ReLU optimization layer.

caused by the nature of off-policy learning such as DQN, which includes high variance and slow convergence. The future work includes fine-tuning the hyperparameters to see whether optimization layers can boost the performance of DQN and deriving better optimization problems from the theoretical perspective.

References

[1] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and Z. Kolter. Differentiable convex optimization layers. *arXiv preprint arXiv:1910.12430*, 2019.

[2] B. Amos and J. Z. Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning*, pages 136–145. PMLR, 2017.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[5] J. C. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, 2011.

[6] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[7] P. Jain and P. Kar. Non-convex optimization for machine learning. *Foundations and Trends® in Machine Learning*, 10(3-4):142–363, 2017.

[8] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[11] 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.

[12] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan. 2016.

[13] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[14] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[15] M. D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.