Formatting instructions for NIPS 2017

David S. Hippocampus*
Department of Computer Science
Cranberry-Lemon University
Pittsburgh, PA 15213
hippo@cs.cranberry-lemon.edu

Abstract

This project explores the integration of optimization problems into deep learning networks, a concept inspired by the works of OptNet and CombOptNet. The objective is to enhance the performance and capabilities of neural networks by incorporating quadratic and integer programming as layers within the network architecture. The project involves a theoretical exploration of this integration, followed by practical implementation and experimentation. Results from replicating the experiments in the original papers, as well as novel experiments designed to further investigate this theme, are presented.

1 Introduction

Deep learning, a subset of machine learning, has been instrumental in the advancement of numerous fields, including computer vision, natural language processing, and artificial intelligence. Despite its success, a key challenge in deep learning is the inability of traditional network layers to capture complex dependencies and constraints between hidden states. This limitation can potentially hinder the performance and capabilities of the networks.

Two recent works, OptNet and CombOptNet, have proposed innovative solutions to this challenge by integrating optimization problems into the network architecture. OptNet introduces quadratic programming layers into the network, while CombOptNet incorporates integer programming layers. These layers are designed to encode constraints and complex dependencies that traditional layers often fail to capture, thereby enhancing the network's performance and capabilities.

The integration of optimization problems into deep learning networks is a promising area of research due to its potential to significantly improve network performance. However, this integration is a complex process that requires a deep understanding of both optimization problems and deep learning networks.

In this project, we aim to explore this integration from both a theoretical and practical perspective. The project begins with a comprehensive understanding of the OptNet and CombOptNet papers, followed by a restatement of their content in layman's terms. This step is crucial for ensuring a solid theoretical foundation for the project.

Following the theoretical understanding, we move into the practical phase of the project. This involves replicating the experiments from the OptNet and CombOptNet papers to validate their theories and results. Additionally, we design and implement new experiments to further explore the potential of optimization integration in deep learning networks. These experiments provide practical insights into the process and benefits of optimization integration.

*Use footnote for providing further information about author (webpage, alternative address)—*not* for acknowledging funding agencies.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

Through this project, we aim to contribute to the understanding and application of optimization integration in deep learning networks, thereby pushing the boundaries of what these networks can achieve.

2 OptnetDifferentiable Optimization as a Layer in Neural Networks

2.1 Introduction and Background

Optimization plays a crucial role in modeling complex phenomena and providing concrete decision-making processes in sophisticated environments. In the machine learning setting, a wide array of applications consider optimization as a means to perform inference in learning. The general idea of solving restricted classes of optimization problem using neural networks goes back many decades, but has seen a number of advances in recent years.

In this paper, the authors present OptNet, a novel architecture that incorporates optimization problems as layers in a neural network. This approach is a step towards learning optimization problems behind real-world processes from data that can be learned end-to-end rather than requiring human specification and intervention.

The authors propose to insert optimization problems anywhere in the network. Specifically, they study OptNet layers defined by a quadratic program. The optimal solution of this optimization problem becomes the output of the layer, and any of the problem data can depend on the value of the previous layer. The forward pass in the OptNet architecture involves setting up and finding the solution to this optimization problem.

Training deep architectures, however, requires not just a forward pass in the network but also a backward pass. This requires computing the derivative of the solution to the quadratic program with respect to its input parameters. To obtain these derivatives, the authors differentiate the Karush-Kuhn-Tucker (KKT) conditions of the quadratic program at a solution to the problem using techniques from matrix differential calculus.

The authors have developed a solver that can solve batches of quadratic programs over 100 times faster than existing highly tuned quadratic programming solvers such as Gurobi and CPLEX. One crucial algorithmic insight in the solver is that by using a specific factorization of the primal-dual interior point update, they can obtain a backward pass over the optimization layer virtually "for free" (i.e., requiring no additional factorization once the optimization problem itself has been solved). Together, these innovations enable parameterized optimization problems to be inserted within the architecture of existing deep networks.

2.2 OptNet: solving optimization within a neural network

Although in the most general form, an OptNet layer can be any optimization problem, in this paper we will study OptNet layers defined by a quadratic program. The quadratic programming problem can be formally defined as follows:

$$\begin{array}{ll} \underset{z}{\text{minimize}} & \frac{1}{2} z^T Q z + q^T z \\ \text{subject to} & Az = b, Gz \leq h \end{array} \tag{1}$$

where z is the optimization variable, Q is a positive semidefinite matrix, q, A, b, G, and h are problem data. The optimal solution of this optimization problem becomes the output of the layer, denoted as z_{i+1}, and any of the problem data Q, q, A, b, G, h can depend on the value of the previous layer z_i.

The forward pass in the OptNet architecture involves setting up and finding the solution to this optimization problem. Training deep architectures, however, requires not just a forward pass in the network but also a backward pass. This requires computing the derivative of the solution to the quadratic program with respect to its input parameters.

To obtain these derivatives, the authors differentiate the Karush-Kuhn-Tucker (KKT) conditions of the quadratic program at a solution to the problem using techniques from matrix differential calculus.

The KKT conditions are a set of inequalities and equalities that are necessary and sufficient for a solution to be optimal.

The Lagrangian of the quadratic program is given by:

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

where ν are the dual variables on the equality constraints and $\lambda \geq 0$ are the dual variables on the inequality constraints. The KKT conditions for stationarity, primal feasibility, and complementary slackness are:

$$Qz^* + q + A^T \nu^* + G^T \lambda^* = 0 \tag{3}$$
$$Az^* - b = 0 \tag{4}$$
$$D(\lambda^*)(Gz^* - h) = 0 \tag{5}$$

where $D(\cdot)$ creates a diagonal matrix from a vector and z^*, ν^*, λ^* are the optimal primal and dual variables.

The authors have developed methods to make this approach practical and reasonably scalable within the context of deep architectures. They have also developed a solver that can solve batches of quadratic programs over 100 times faster than existing highly tuned quadratic programming solvers such as Gurobi and CPLEX.

2.2.1 QP Layers

In the OptNet architecture, the authors propose to insert optimization problems anywhere in the network. Specifically, they study OptNet layers defined by a quadratic program. The quadratic programming problem can be formally defined as follows:

$$\begin{array}{ll} \underset{z}{\text{minimize}} & \frac{1}{2} z^T Q z + q^T z \\ \text{subject to} & Az = b, Gz \leq h \end{array} \tag{6}$$

where z is the optimization variable, Q is a positive semidefinite matrix, q, A, b, G, and h are problem data. The optimal solution of this optimization problem becomes the output of the layer, denoted as z_{i+1}, and any of the problem data Q, q, A, b, G, h can depend on the value of the previous layer z_i.

The forward pass in the OptNet architecture involves setting up and finding the solution to this optimization problem. Training deep architectures, however, requires not just a forward pass in the network but also a backward pass. This requires computing the derivative of the solution to the quadratic program with respect to its input parameters.

To obtain these derivatives, the authors differentiate the Karush-Kuhn-Tucker (KKT) conditions of the quadratic program at a solution to the problem using techniques from matrix differential calculus. The KKT conditions are a set of inequalities and equalities that are necessary and sufficient for a solution to be optimal.

The Lagrangian of the quadratic program is given by:

$$L(z, \nu, \lambda) = \frac{1}{2} z^T Q z + q^T z + \nu^T (Az - b) + \lambda^T (Gz - h) \tag{7}$$

where ν are the dual variables on the equality constraints and $\lambda \geq 0$ are the dual variables on the inequality constraints. The KKT conditions for stationarity, primal feasibility, and complementary slackness are:

$$Qz^* + q + A^T\nu^* + G^T\lambda^* = 0 \tag{8}$$
$$Az^* - b = 0 \tag{9}$$
$$D(\lambda^*)(Gz^* - h) = 0 \tag{10}$$

where $D(\cdot)$ creates a diagonal matrix from a vector and z^*, ν^*, λ^* are the optimal primal and dual variables.

The authors have developed methods to make this approach practical and reasonably scalable within the context of deep architectures. They have also developed a solver that can solve batches of quadratic programs over 100 times faster than existing highly tuned quadratic programming solvers such as Gurobi and CPLEX.

2.2.2 Differentiating through the KKT conditions

To compute the derivative of the solution to the quadratic program with respect to its input parameters, the authors differentiate the Karush-Kuhn-Tucker (KKT) conditions of the quadratic program at a solution to the problem using techniques from matrix differential calculus. The KKT conditions are a set of inequalities and equalities that are necessary and sufficient for a solution to be optimal.

The Lagrangian of the quadratic program is given by:

$$L(z, \nu, \lambda) = \frac{1}{2}z^T Qz + q^T z + \nu^T(Az - b) + \lambda^T(Gz - h) \tag{11}$$

where ν are the dual variables on the equality constraints and $\lambda \geq 0$ are the dual variables on the inequality constraints. The KKT conditions for stationarity, primal feasibility, and complementary slackness are:

$$Qz^* + q + A^T\nu^* + G^T\lambda^* = 0 \tag{12}$$
$$Az^* - b = 0 \tag{13}$$
$$D(\lambda^*)(Gz^* - h) = 0 \tag{14}$$

where $D(\cdot)$ creates a diagonal matrix from a vector and z^*, ν^*, λ^* are the optimal primal and dual variables.

A notable difference from other work within machine learning that the authors are aware of, is that they analytically differentiate through inequality as well as just equality constraints by differentiating the complementarity conditions. This differs from other works where they instead approximately convert the problem to an unconstrained one via a barrier method.

The authors have developed methods to make this approach practical and reasonably scalable within the context of deep architectures.

3 CombOptNet: Fit the Right NP-Hard Problem by Learning Integer Programming Constraints

3.1 Introduction and problem restatement

In this work, the authors propose a method called CombOptNet, which is a module that can be integrated into deep architectures. This module is capable of learning both the cost terms and the constraints, allowing the architecture to extract features from raw data and solve a suitable combinatorial problem.

A key example used in this paper to illustrate the method is the deep keypoint matching problem. Given a source and a target image, each labeled with a set of annotated keypoints, the task is to find the correct matching between the sets of keypoints from visual information. Not every keypoint has to be visible in both images, so some keypoints can also remain unmatched.

To solve this problem, the authors modify the BB-GM architecture by replacing the blackbox backpropagation module that employs a dedicated solver with CombOptNet. This replacement comes with a few important considerations. The method relies on a fixed dimensionality n of the problem for learning a static (i.e., not input-dependent) constraint set. However, in the BB-GM architecture, the dimensionality of the predicted costs depends on the number of nodes (i.e., keypoints) p and the number of edges.

In this work, the authors propose a method called CombOptNet, which is a module that can be integrated into deep architectures. This module is capable of learning both the cost terms and the constraints, allowing the architecture to extract features from raw data and solve a suitable combinatorial problem.

A key example used in this paper to illustrate the method is the deep keypoint matching problem. Given a source and a target image, each labeled with a set of annotated keypoints, the task is to find the correct matching between the sets of keypoints from visual information. Not every keypoint has to be visible in both images, so some keypoints can also remain unmatched.

To solve this problem, the authors modify the BB-GM architecture by replacing the blackbox backpropagation module that employs a dedicated solver with CombOptNet. This replacement comes with a few important considerations. The method relies on a fixed dimensionality n of the problem for learning a static (i.e., not input-dependent) constraint set. However, in the BB-GM architecture, the dimensionality of the predicted costs depends on the number of nodes (i.e., keypoints) p and the number of edges.

3.2 Method

The authors propose a method called CombOptNet, which is a module that can be integrated into deep architectures. This module is capable of learning both the cost terms and the constraints, allowing the architecture to extract features from raw data and solve a suitable combinatorial problem.

3.2.1 CombOptNet as a Module in a Deep Architecture

The method is based on the idea of integrating integer programming solvers into neural network architectures as layers capable of learning both the cost terms and the constraints. The resulting end-to-end trainable architectures jointly extract features from raw data and solve a suitable (learned) combinatorial problem with state-of-the-art integer programming solvers.

3.2.2 Deep Keypoint Matching Problem

A key example used in this paper to illustrate the method is the deep keypoint matching problem. Given a source and a target image, each labeled with a set of annotated keypoints, the task is to find the correct matching between the sets of keypoints from visual information. Not every keypoint has to be visible in both images, so some keypoints can also remain unmatched.

To solve this problem, the authors modify the BB-GM architecture by replacing the blackbox backpropagation module that employs a dedicated solver with CombOptNet. This replacement comes with a few important considerations. The method relies on a fixed dimensionality n of the problem for learning a static (i.e., not input-dependent) constraint set. However, in the BB-GM architecture, the dimensionality of the predicted costs depends on the number of nodes (i.e., keypoints) p and the number of edges.

3.2.3 Learning Constraints

The authors propose a method for learning constraints in the form of integer programming problems. The constraints are learned by training a neural network to predict the coefficients of the constraints based on the input data. The learned constraints are then used to solve the integer programming problem using a state-of-the-art solver.

3.2.4 Performance Analysis

The authors provide an extensive performance analysis of their method, demonstrating its effectiveness on synthetic data and a competitive computer vision keypoint matching benchmark. The

method is shown to outperform traditional methods that rely on hand-crafted constraints or heuristics.

4 Optnet Experimental Project Replication and Innovation

In this project, we aim to replicate and innovate upon the experiments conducted in the OptNet paper **?**. OptNet is a novel approach that integrates optimization problems (specifically, Quadratic Programs) as layers in a neural network. We focus on replicating three experiments available in the OptNet GitHub repository **?**.

4.1 Sudoku Experiment

The Sudoku experiment demonstrates the use of OptNet for solving Sudoku puzzles. The authors model the Sudoku problem as an Integer Quadratic Program (IQP) and relax it to a Quadratic Program (QP) that can be solved using OptNet. The Sudoku problem is formulated as follows:

$$
\begin{aligned}
\underset{x}{\text{minimize}} \quad & 0 \\
\text{subject to} \quad & x_{ijk} \in \{0,1\},\ i,j,k = 1,\dots,9 \\
& \sum_{k=1}^{9} x_{ijk} = 1,\ i,j = 1,\dots,9 \\
& \sum_{i=1}^{9} x_{ijk} = 1,\ j,k = 1,\dots,9 \\
& \sum_{j=1}^{9} x_{ijk} = 1,\ i,k = 1,\dots,9 \\
& \sum_{i=3p+1}^{3p+3} \sum_{j=3q+1}^{3q+3} x_{ijk} = 1,\ k,p,q = 1,\dots,9 \\
& x_{ijk} = y_{ijk},\ i,j,k = 1,\dots,9,\ y_{ijk} \neq 0
\end{aligned}
\tag{15}
$$

where x_{ijk} is a binary variable that is 1 if the number k is in cell (i,j), and y_{ijk} are the given numbers in the Sudoku puzzle.

The Sudoku dataset provided in the repository is used for this experiment. The model is trained to learn the QP parameters that can solve the Sudoku puzzles. The loss function is defined as the discrepancy between the model's solution and the actual solution.

The results of the experiment are evaluated based on the accuracy of the solved puzzles.

One of the main challenges in replicating this experiment was ensuring that the QP was always feasible and had a solution. This was addressed by carefully initializing the QP parameters and using regularization techniques.

4.2 Denoising

Denoising is a classic problem in the field of image processing and computer vision. The goal is to remove noise from a noisy image, thereby improving its quality. In the context of OptNet, the denoising problem is formulated as a quadratic programming problem and solved using the OptNet architecture.

The mathematical formulation of the denoising problem is as follows:

Given a noisy image y, we want to find an image x that minimizes the following objective function:

$$\min_x \frac{1}{2}||x-y||_2^2 + \lambda \sum_i ||D_i x||_1$$

where D_i are operators that compute finite differences in the i-th direction, and λ is a regularization parameter that controls the trade-off between the data fidelity term and the regularization term. The l_1 norm encourages sparsity in the finite differences, which leads to piecewise constant solutions that are characteristic of denoised images.

The above problem is a Lasso problem, which is a special case of quadratic programming. However, because of the l_1 norm, it is not differentiable everywhere, which makes it challenging to solve using standard optimization methods. In the OptNet paper, the authors propose to use a differentiable approximation of the l_1 norm, which allows them to use the QP solver in the forward pass and backpropagation in the backward pass.

During the replication of this experiment, we encountered challenges related to the high computational cost of the QP solver. Quadratic programming problems are generally computationally expensive to solve, especially for large-scale problems such as image denoising. We resolved this issue by using a more efficient implementation of the QP solver, which uses a primal-dual interior point method. This method is more efficient than standard QP solvers for large-scale problems, and it allows us to solve the denoising problem in a reasonable amount of time.

4.3 Classification (CLS)

Classification is a fundamental problem in machine learning, where the goal is to assign a label to an input data point based on its features. In the context of OptNet, the authors propose to use a quadratic programming (QP) formulation for multi-class classification.

The mathematical formulation of the classification problem in OptNet is as follows:

Given a set of training data (x_i, y_i), where x_i is the feature vector and y_i is the corresponding label, we want to find a weight matrix W and bias vector b that minimize the following objective function:

$$\min_{W,b} \frac{1}{2}||W||_F^2 + C\sum_i \xi_i$$

subject to the constraints:

$$y_i(W^T x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0$$

where $||.||_F$ denotes the Frobenius norm, C is a regularization parameter, and ξ_i are slack variables that allow for misclassification of data points. The above problem is a standard formulation of the Support Vector Machine (SVM) problem, which is a well-known method for classification.

In the OptNet paper, the authors propose to solve the above QP problem using the OptNet architecture. They replace the non-differentiable hinge loss used in standard SVM with a differentiable quadratic penalty, which allows them to use the QP solver in the forward pass and backpropagation in the backward pass.

During the replication of this experiment, we encountered challenges related to the scalability of the QP solver. As the size of the training data increases, the size of the QP problem also increases, which leads to increased computational cost. We resolved this issue by using mini-batch training, which allows us to solve smaller QP problems at each iteration, thereby reducing the computational cost.

4.4 Extension of Optnet:Comparison of Backward Pass Time: SGD vs OptNet

In our extension of the OptNet project, we conducted an experiment to compare the time taken for the backward pass in a simple linear model using Stochastic Gradient Descent (SGD) and the OptNet model. The goal of this experiment was to evaluate the computational efficiency of the OptNet model in comparison to traditional optimization methods.

We implemented a simple linear model using PyTorch, a popular deep learning framework. The model was trained using SGD, a widely used optimization algorithm in machine learning. The backward pass time, which is the time taken to compute the gradients of the loss function with respect to the model parameters, was measured over 1000 training steps.

Next, we implemented the OptNet model as described in the original paper. The OptNet model solves a Quadratic Programming (QP) problem in the forward pass and uses backpropagation in the backward pass. The QP problem is formulated as follows:

$$\begin{gathered}\min_x \frac{1}{2}x^T Qx + p^T x \\ \text{subject to } Gx \leq h \\ Ax = b\end{gathered}$$

where Q is a positive semi-definite matrix, p is a vector, G and h define inequality constraints, and A and b define equality constraints. The backward pass time for the OptNet model was also measured over 1000 training steps.

The results of our experiment showed that the backward pass time for the OptNet model was significantly higher than that for the SGD model. This is expected as the OptNet model involves solving a QP problem, which is computationally more expensive than the simple gradient computations in SGD.

However, it is important to note that the OptNet model provides a more accurate solution to the optimization problem, as it solves the problem exactly in the forward pass. On the other hand, SGD only provides an approximate solution, as it uses gradient descent to iteratively update the model parameters.

In conclusion, while the OptNet model may be computationally more expensive, it provides a more accurate solution to the optimization problem. Future work could explore ways to improve the computational efficiency of the OptNet model, such as by using more efficient QP solvers or by using hardware acceleration.

5 CombOptNet Experimental Project Replication and Innovation

5.1 Knapsack

The Knapsack problem is a classic problem in combinatorial optimization. The problem is defined as follows: given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

The Knapsack problem can be formally defined as follows:

Given:

- A set of n items, each with a weight w_i and a value v_i for $i = 1, \ldots, n$.
- A weight limit W.

Find a vector $x = (x_1, \ldots, x_n)$, where x_i is the number of instances of item i to include in the knapsack, that maximizes

$$\sum_{i=1}^{n} v_i x_i \tag{16}$$

subject to

$$\sum_{i=1}^{n} w_i x_i \leq W \tag{17}$$

and

$$x_i \in 0, 1 \quad \text{for all } i. \tag{18}$$

The Knapsack problem is a combinatorial optimization problem, and it is NP-hard. However, it has a pseudo-polynomial time algorithm using dynamic programming. The basic idea is to construct a table of size $n \times W$ and fill it in a bottom-up manner. The entry in the i-th row and w-th column will store the maximum value that can be obtained using items from 1 to i with a total weight less than or equal to w.

In the experiment, the authors reported the evaluation accuracy (i.e., $y = y^*$ in percentage) for $m = 1, 2, 4, 8$ constraints. The results are as follows:

m	Accuracy (height1
64.7 $\pm$ 2.8 2	63.5 $\pm$ 3.7 4
65.7 $\pm$ 3.1 8	62.6 $\pm$ 4.4 height

The results show that increasing the number of learnable constraints does not significantly improve the performance.

One of the main challenges in replicating the experiment was understanding the implementation of the Knapsack problem in the provided code. This required a deep understanding of both the problem itself and the specific implementation details. To overcome this, we spent time studying the problem and the code, and also consulted external resources to clarify any confusing aspects.

Another challenge was dealing with the computational resources required for the experiment. The Knapsack problem is NP-hard, and the computational complexity can be quite high, especially for large problem instances. To mitigate this, we carefully managed our computational resources and used efficient implementations whenever possible.

5.2 Expand: Static Constraints

The Static Constraints experiment is part of the broader study conducted in the paper, focusing on the impact of the number of learnable constraints on the performance of the model. The experiment is designed to provide additional results about the effect of increasing the number of learnable constraints.

In this experiment, we consider a combinatorial optimization problem where the constraints are static, i.e., not dependent on the input. We study the impact of varying the number of learnable constraints on the performance of the model.

Specifically, the problem we consider can be formalized as follows:

Given:

- A set of n variables, each of which can take the value 0 or 1.
- A set of m static constraints, each of which is a linear inequality involving these variables.

Find an assignment of the variables that satisfies all the constraints and maximizes some objective function.

In the setup of the experiment, we use the following parameters:

- Seed: The seed for the random number generator, used to ensure the reproducibility of the experiment.
- Working Directory: The directory used to store the results of the experiment.
- Dataset Parameters: These include the type of the dataset (dataset_type), the specification of the dataset (dataset_specification), the size of the dataset (train_dataset_size), etc.
- Training Parameters: These include whether to use CUDA (use_cuda), the name of the loss function (loss_name), the name and parameters of the optimizer (optimizer_name and optimizer_params), etc.

In the experiment, we vary the number of learnable constraints and observe its impact on the performance of the model.

The computation process involves training the model with different numbers of learnable constraints and evaluating its performance. The constraints are learned during the training process and are used

in the combinatorial optimization task during the evaluation. The performance is measured in terms of the evaluation accuracy.

The results of the Static Constraints experiment are not explicitly provided in the paper. However, it is mentioned that increasing the number of learnable constraints generally improves performance and reduces variance, but it can also lead to overfitting.

One of the main challenges in replicating the Static Constraints experiment was the lack of explicit problem definition and results in the paper. This required us to make assumptions about the problem and the expected results, which could potentially lead to inaccuracies in the replication. To overcome this, we carefully studied the paper and the provided code to understand the problem and the experiment setup.

Another challenge was dealing with the potential overfitting issue when increasing the number of learnable constraints. To mitigate this, we used techniques such as regularization and early stopping during the training process.