Thesis - Progress Report

Mitchell Keegan

s47365357

June 2023

Contents

1 Literature Review **2**

2 The Lagrangian Dual Framework **5**

3 Knapsack Problem **8**

3.1 Introduction 8

3.2 Instance Generation 9

3.3 Model Details & Experimental Setup 9

3.3.1 Output Evaluation 10

3.3.2 Restricted Capacity Instances 11

3.3.3 Full Range Capacity Instances 11

3.4 Results 13

4 Future Works **16**

Bibliography 18

Chapter 1

Literature Review

Constrained optimisation considers problems of the form:

$$\begin{aligned} x^* = \operatorname*{argmin}_x \quad & f(x) \\ \text{subject to} \quad & x \in \mathcal{C} \end{aligned}$$

In the lingo of optimisation, f is call the objective function, x is the set of decision variable, x^* is the optimal solution, and $\mathcal{C}$ defines the set of feasible solutions. Constrained optimisation problems are of great interest owing to their application in computer science, operations research, engineering, finance and other quantitative fields.

The study of constrained optimisation is typically split into subfields according to the structure of the objective function and the constraint set. One important subfield is the study of combinatorial optimisation (CO) problems, in which the constraint set takes on a discrete structure.

Many CO problems of interest fall into the category of $\mathcal{NP}$-hard, which likely precludes the existence of polynomial-time algorithms. Even so, in practice algorithms have been developed which can solve many CO problems exactly (or approximately) with reasonable efficiency. The range of techniques employed in these algorithms is broad, but often involve advanced heuristics, polynomial time approximation schemes, or targeting specific distributions of instances which do not exhibit the worst case complexity.

An important type of CO problem is that of integer programming (IP). IP problems take the form:

$$\begin{aligned} x^* = \operatorname*{argmin}_x \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \\ & x \geq 0 \\ & x \in \mathbb{Z} \end{aligned}$$

A related problem is mixed-integer linear programming, in which some subset of the decision variables are relaxed to be continuous. In this report I will use the terms combinatorial optimisation (CO), integer programming (IP), and mixed-integer linear programming (MILP) interchangeably. While these are not strictly the same, they are closely related and in many cases CO problems can be cast in the form of an IP

problem.

Machine learning, loosely speaking, concerns itself with developing models which can use data to learn to approximate a given function or process. Early attempts at tackling combinatorial optimisation problems using tools from machine learning date back decades. For example [1] attempted to use Hopfield Networks to approximate the objective of the travelling salesman problem (TSP). The last decade has seen explosive advances in machine learning, with progress in deep learning providing practical solutions to problems in the field of computer vision and natural language processing (among many others) which were once considered incredibly difficult or entirely intractable. This has naturally spurred renewed interest in applying ML for CO problems.

The use of machine learning for CO broadly falls into two categories: CO augmented by ML and End-to-End prediction of CO solutions. CO augmented ML refers to the use of machine learning techniques to improve existing CO solvers, often by approximating solutions to some important subproblem. Interesting examples include learning strategies for branching or selecting cutting planes in branch and cut algorithms [2], [3]. Often these techniques are motivated by the fact that solving these subproblems can be computationally expensive, this makes a machine learning model with relatively cheap inference an attractive replacement. In other cases solutions to these subproblems are unsatisfactory and the hope is that machine learning techniques may yield improved results. A more complete survey of these techniques is provided in [4].

This report will focus on End-to-End learning, in which ML models are employed to approximate solutions to CO problems. While there are cases in which CO problems can be solved efficiently for practical purposes, this is far from universally true. In particular traditional solvers may be too slow in situations where solutions are needed in real-time. This presents a compelling use case for ML models, which typically have fast inference times.

The ML models employed in this context can broadly be separated into reinforcement learning and supervised learning. Reinforcement learning is not considered here, [5] surveys recent results from this approach. In supervised learning, models are trained by being shown inputs along with corresponding outputs. An input consists of the parameters for one instance, in the case of an IP problem this would be the A matrix along with the vectors b and c. The output is the decision variables x^* in the optimal solution.

The first challenge for a supervised learning approach is generating training data. The obvious hurdle is that many CO problems of interest are $\mathcal{NP}$-hard, meaning that naively generating target labels may be prohibitively expensive. In some cases many instances of practical interest may already be available due to their use in real-world applications. In other cases we may target more tractable subproblems. Some interesting work on the relationship between instance generation and the qualities of learned models can be found in [6] and [7].

Another challenge is the question of how to enforce constraint satisfaction of the model outputs. There is traditionally no way to directly enforce constraints on the outputs of learned models. One promising approach is the Lagrangian Dual Framework (LDF) introduced in [8], which relaxes the constraints into the objective function. They showed strong results with a high degree of constraint satisfaction on the AC optimal power flow (AC-OPF) problem, a continuous optimisation problem. Interesting the LDF can not only be used to enforce constraints on the predictor output, but on the predictor itself (E.g. constraints between samples). Applications of the LDF to CO problems is the primary focus of this report, and as such will be treated in detail in the next chapter. Another interesting approach, called DC3, is proposed in [9]. DC3 employs differentiable processes to enforce the feasibility of solutions during training. It showed strong results in terms of optimality, feasibility and inference time on AC-OPF. A third approach, proposed in [10], involves interleaving the supervised training of an unconstrained ML model with a "Master step" which enforces the constraints. It has the advantage that in practice any supervised ML could be employed, not only models with gradient based training.

Coverage of a broader range of related topics can be found in survey papers [4], [11] and [12].

Chapter 2

The Lagrangian Dual Framework

The augmented Lagrangian method is a constrained optimisation technique which revolves around relaxing constraints into the objective to form an unconstrained optimisation problem. Consider a general constrained optimisation problem:

$$\begin{aligned} \mathcal{O} = \underset{y}{\operatorname{argmin}} \quad & f(y) \\ \text{subject to} \quad & h_i(y) = 0,\ i = 1, \ldots, m_h \\ & g_j(y) \leq 0,\ j = 1, \ldots, m_g \end{aligned}$$

The new relaxed objective is:

$$f_\lambda(y) = f(y) + \sum_{i=1}^{m_h} \lambda_i h_i(y) + \sum_{j=1}^{m_g} \lambda_j g_j(y)$$

where $\lambda_i \geq 0$ and $\lambda_j \geq 0$ denote the Lagrange multipliers associated with the constraints. Another approach is to consider the violation degree of the constraints:

$$f_\lambda(y) = f(y) + \sum_{i=1}^{m_h} \lambda_i |h_i(y)| + \sum_{j=1}^{m_g} \lambda_j \max(0, g_j(y))$$

Both of these can be generalised by considering functions $\nu_i(h)$ and $\nu_j(g)$ which return either the constraint satisfiability or the degree of constraint violation as required.

$$f_\lambda(y) = f(y) + \sum_{i=1}^{m_h} \lambda_i \nu_i(h_i(y)) + \sum_{j=1}^{m_g} \lambda_j \nu_j(g_j(y))$$

The new unconstrained optimisation problem, now called the Lagrangian Relaxation, is:

$$LR_\lambda = \underset{y}{\operatorname{argmin}}\, f_\lambda(y)$$

for some set of Lagrangian multipliers λ. The solution satisfies forms a lower bound on the original constrained problem, i.e. $f(LR_\lambda) \leq f(\mathcal{O})$.

To find the strongest Lagrangian relaxation of $\mathcal{O}$, we can find the best set of Lagrangian multipliers using the Lagrangian dual.

$$LD = \underset{\lambda \geq 0}{\operatorname{argmax}} f(LR_\lambda)$$

[8] proposes a method, called the Lagrange Dual Framework (LDF), to leverage the augmented Lagrangian to improve constraint satisfaction in neural network models. In particular, consider parametrising the original optimisation problem in terms of the parameters of the objective function and constraints:

$$\begin{aligned} \mathcal{O}(d) = \underset{y}{\operatorname{argmin}} \quad & f(y, d) \\ \text{subject to} \quad & g_i(y, d) \leq 0,\ i = 1, \ldots, m \end{aligned}$$

The equality constraints have been dropped from the problem for clarity, but can easily be added back in as needed. The overall goal is the learn some parametric model $\mathcal{M}_w$, parameterised by weights w, such that $\mathcal{M}_w \approx \mathcal{O}$.

Assume we have a set of training data $D = \{(d_l, y_l = \mathcal{O}(d_l)\}$ for $l = 1, \ldots, n$. The learning problem is now to solve:

$$\begin{aligned} w^* = \underset{w}{\operatorname{argmin}} \quad & \sum_{l=1}^{n} \mathcal{L}(\mathcal{M}_w(d_l), y_l) \\ \text{subject to} \quad & g_i(\mathcal{M}_w(d_l), d_l) \leq 0,\ i = 1, \ldots, m,\ l = 1, \ldots, n \end{aligned}$$

for some loss function $\mathcal{L}$. In essence this states that $\mathcal{M}_w$ should have weights such that it minimises the loss over all training samples, while being such that the output satisfies the constraints for all samples in the training data. To this end, relax the constraints into the loss function to form the Lagrangian loss function:

$$\mathcal{L}_\lambda(\hat{y}_l, y_l, d_l) = \mathcal{L}(\hat{y}_l, y_l) + \sum_{i=1}^{m} \lambda_i \nu(g_i(\hat{y}_l, d_l))$$

where $\hat{y}_l = \mathcal{M}_w(d_l)$. For a given set of Lagrange multipliers λ, the optimisation problem is:

$$w^*(\lambda) = \underset{w}{\operatorname{argmin}} \sum_{l=1}^{n} \mathcal{L}_\lambda(\mathcal{M}_w(d_l), y_l, d_l)$$

This will produce an approximation $\mathcal{M}_{w^*(\lambda)}$ of $\mathcal{O}$. To find a stronger Lagrangian relaxation, use the Lagrangian Dual to compute the optimal multipliers:

$$\lambda^* = \underset{\lambda}{\operatorname{argmin}} \min_{w} \sum_{l=1}^{n} \mathcal{L}_\lambda(\mathcal{M}_w(d_l), y_l, d_l)$$

The LDF algorithm interleaves learning the weights w and multipliers λ. The training process is summarised in Algorithm 1.

Algorithm 1 LDF Algorithm

Input: $D = \{(d_l, y_l = \mathcal{O}(d_l)\}$ for $l = 1, \ldots, n$
for $k = 0, 1, ..., n_{epochs}$ **do**
  for all $(y_l, d_l) \in D$ **do**
    $\hat{y} \leftarrow \mathcal{M}_w(d_l)$
    $w \leftarrow w - \alpha \nabla_w \mathcal{L}_{\lambda^k}(\hat{y}, y_l, d_l)$
  end for
  $\lambda_i^{k+1} \leftarrow \lambda_i^k s_k \sum_{l=n}^{n} \nu_i(g_i(\hat{y}, d_l)), \quad i = 1, ..., m$
end for

In [8], the authors apply this algorithm to the problem of AC optimal power flow, which is a continuous nonlinear optimisation problem. The framework is applied in [13] to two problems constraining the behaviour of the predictor. The first is predicting error in transprecision computer, where dominance relations between samples in the training data are used to guide the predictor output. The second application relates to fairness and privacy in a classification task. There is, to my knowledge, no current literature describing the application of the LDF algorithm to discrete optimisation problems.

Chapter 3

Knapsack Problem

3.1 Introduction

The Knapsack problem (KP) is a classic problem in combinatorial optimisation.

The core statement of the Knapsack Problem is as follows; there are n items, each with weight w_i and value v_i. The goal is to select a set of items which maximise the total value, such that the total weights of the chosen items is not greater than some capacity W. It is formally defined as an integer programming problem below:

$$\begin{aligned} x^* = \underset{x}{\text{argmax}} & \sum_{i=1}^{n} x_i v_i \\ \text{s.t.} & \sum_{i=1}^{n} x_i w_i \leq W \\ & x_i \in \{0,1\},\ i = 1, ..., n \end{aligned}$$

KP has many variants including:

- Bounded knapsack problem - Enforces integral constraints on x_i rather than binary, but bounds the number of available copies of each items.
- Unbounded knapsack problem - Similar to the bounded knapsack problem but does not bound the number of copies of each item.
- Multiple knapsack problem - Introduces multiple knapsacks with independent capacities.

Here I focus only on the 0-1 knapsack problem state above. Algorithms for solving these variants along with many others can be found in [14].

The intention of studying the Knapsack problem here is not for practical applications. There are already effective algorithms including dynamic programming, branch and bound, and a variety of metaheuristic algorithms. The Knapsack problem can be easily stated and understood, while still retaining some base

level of complexity to be of interest. In some sense it is the simplest non-trivial integer programming problem. The appearance of only a single constraint also makes it easier to isolate and understand how hyperparameters relating to constraint satisfaction affect solutions. These characteristics make it ideal for testing the Langrangian Dual Framework.

To my knowledge there are two relevant papers relating to approximating KP solutions using neural networks. In [15], the authors develop GRU, CNN, and dense NN models for approximating KP solutions. Direct comparison to results from this paper is difficult, since the authors use a greedy approach to decode model outputs which always produces feasible solutions. It may be interesting in future to implement the same output decoding scheme as a point of comparison.

In [16] the authors derive theoretical bounds on the depth and width of an RNN designed to mimic a dynamic programming algorithm. They show that an RNN with depth four and width of order $\mathcal{O}\left((p^*)^2\right)$ can exactly mimic a dynamic programming solution, where p^* is an upper bound on the optimal objective. They prove a similar result for a fully polynomial-time approximation scheme (FPTAS), showing that an RNN with depth five and width of order $\mathcal{O}\left(n^4/\epsilon^3\right)$ achieves a worst case approximation ratio of $1-\epsilon$. These results are then generalised to multiple other combinatorial optimisation problems.

3.2 Instance Generation

Some care must be taken to generate instances which represent a wide variety of adequately difficult cases. I will primarily focus on the knapsack capacity as a proxy for instance difficulty. [17],[18],[19] are concerned with the characteristics of instances that are difficult for commonly used KP solvers. In the future it might be interesting to consider how well models trained on uniformly generated instances generalise to some of these harder instances.

For all instances weights and values were uniformly distributed on $[0, 1]$, with the weights and values being uncorrelated.

I have used two methods of generating instances. The first sets the capacity relative to the expected sum of the weights. The capacity is distributed uniformly on $[0.5n\alpha_1, 0.5n\alpha_2]$, $\alpha_1 < \alpha_2 \in [0, 1]$. Choosing α values corresponds to choosing what proportion of items would fit in the knapsack on average. I will refer to these as *restricted capacity instances*.

The second method was proposed in [18]. The capacity of the j^{th} instance is set to $W_j = \frac{j}{S+1}\sum_{i=1}^{n} w_i$, where S is the total number of instances. I will refer to these as *full range capacity instances*. One might expect that if a model can be learned to accurately approximate KP on some restricted range of capacities, that it would be possible to learn a model over the full range of capacities.

3.3 Model Details & Experimental Setup

Target labels were generated using Gurobi 9.5.2. All models were trained using PyTorch v1.12.1 with Numpy v1.23.5. Instance generation, label generations and training were all conducted on a PC running Ubuntu Linux with an Intel i5-11400 processor [1].

[1] All code will be available at https://github.com/mitchellkeegan/Thesis. At the time of writing the repo is still private and the code is functional but not user friendly

3.3.1 Output Evaluation

The output of the neural network is a vector in $\mathbb{R}^n$, where the i^{th} element is a logit corresponding to the i^{th} weight. For the label loss, the binary cross entropy loss function is used.

Using the network outputs to evaluate the knapsack constraint is not so straightforward. At training time there needs to be some way to map the outputs to binary decision variables. The obvious way to do this is to apply a sigmoid function to scale the outputs into the range $[0, 1]$ and then apply a round function. Applying this naively has the problem that the derivative of the round function is zero everywhere, meaning that informative gradients will not be passed back from the constraint evaluation.

To alleviate this problem I implemented a surrogate gradient for the round function as used for spiking neural networks in [20]. For the surrogate gradient I used the gradient of the sigmoid function centred at 0.5. The surrogate gradient takes the form:

$$\frac{dR}{dx} = \frac{ke^{-k(x-0.5)}}{(e^{-k(x-0.5)}+1)^2}$$

The parameter k affects how tightly the function is distributed around 0.5. Figure 3.1 demonstrates how the gradient changes as k is varied. The gradient will be higher when outputs are near 0.5 and tend towards zero near 0 and 1. This reflects the fact that near 0.5 small changes in the input can cause the output of the rounding function to jump between 0 and 1.

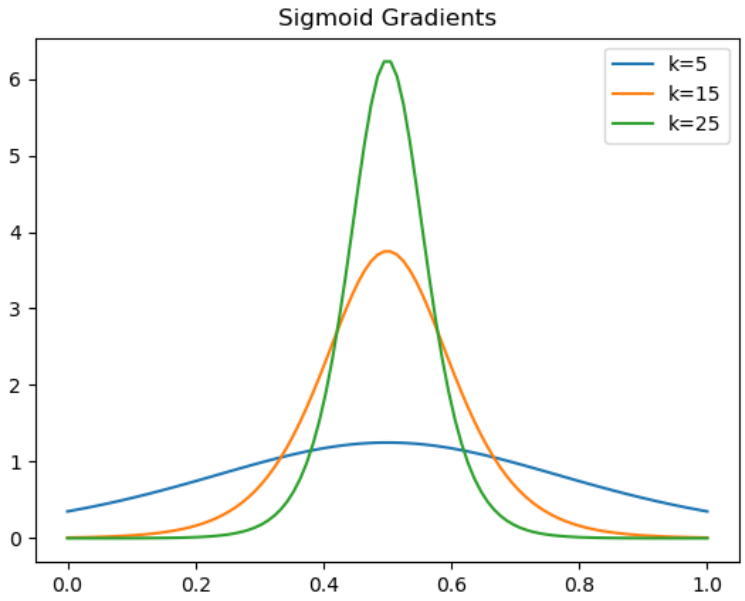

Figure 3.1: Surrogate Gradient

I have also used rounding for inference. [15] uses a greedy approach to decode the output at inference time. This approach may be of interest in the future, but has the disadvantage that it requires some algorithm for post-processing of the network output. I follow [15] in using the approximation ratio (AR) as a metric to evaluate the model. For a set of instances of size S it is defined as below:

$$AR = \frac{1}{S}\sum_{j=1}^{S} max\left(\frac{f^*(x_j)}{f(x_j)}, \frac{f(x_j)}{f^*(x_j)}\right)$$

where $f^*(x_j)$ and $f(x_j)$ are the predicted and true objective values for the j^{th} instance. The approximation ratio is not well defined for instances where either $f^*(x_j)$ or $f(x_j)$ is equal to zero, but not both. I define the approximation ratio to be equal to two whenever this occurs. While arbitrary, the idea is that a strong predictor should have a approximation ratio approaching one. Regardless, instances with $f^*(x_j) = 0$ should be fairly rare, and a strong predictor should only rarely predict $f(x_j)$.

The approximation ratio is used for model validation during training and for the evaluation of the model on the test data.

3.3.2 Restricted Capacity Instances

For the restricted capacity instances, nine different sets of instances were tested for number of weights $n = [50, 200, 500]$ and capacities varying in the range $(\alpha_1, \alpha_2) = [(0.15, 0.25), (0.45, 0.55), (0.75, 0.85)]$. $10,000$ instances were generated, with $8,000$ instances used for training and $1,000$ instances put aside for validation and test sets.

For all instance types I used a simple neural network with two hidden layers and ReLU activation functions. For instances with 50 and 200 weights, the layers have widths of 512 and 256, for the instances with 500 weights the layers have widths of 2048 and 1024.

Initial testing indicated the following:

- Batch normalisation applied after the ReLU activation function is extremely effective, and practically essential for the larger instances. I found dropout to be much less effective than batch normalisation. Interestingly, for both forms of regularisation, I found them most effective when only applied on the first hidden layer, but not the second.
- $k = 25$ appears to work well in practice, varying it does not have a significant effect on the training process.
- A Lagrange step size of 1 is effective.

I implemented a learning rate scheduler which would halve the learning rate after 50 epochs without any improvement in the AR on the validation set. Each model was trained for a maximum of 1,000 epochs, or until the AR had not improved in 100 epochs. The chosen model was the one which minimised the AR.

3.3.3 Full Range Capacity Instances

I generated 30,000 full range capacity instances, as described in Section 3.2. Due to the computational cost I have so far only trained a model on smaller instances with $n = 50$.

Initial testing indicated the following:

- As before, batch normalisation on the first hidden layer is effective.

- A two layer neural networks and layers widths of 512 and 256 was effective. Increasing the width of the layers to 2048 and 1024 did not yield any improvement, so the smaller model was used.
- A Lagrangian step size of 1 could destabilise the training procedure. Much smaller step sizes are required. I experimented with a step size scheduler which would increase the step size at later epochs but it still had a destabilising effect.
- Gradient clipping helped significantly to stabilise the training procedure, although it does not seem strictly necessary and more testing may be needed.

With these in mind I ran a grid search over the learning rate $\in \{10^{-3}, 10^{-4}\}$, Lagrangian step size $\in \{0.1, 0.01, 0.001\}$, maximum gradient norm $\in \{0.1, 1\}$ and k $\in \{5, 15, 25, 35\}$.

Similar to the restricted capacity instances I chose the model which minimised the approximation ratio, with early stopping after 200 epochs without any improvement in the model.

The chosen model had a learning rate of 10^{-4}, lagrangian step size of 0.001, clipped the gradient norms to 1 and set $k = 25$.

It's worth noting that while clipping the gradient norms helped stabilise the training, it is still possible to achieve similar (but very slightly worse) results without any clipping. Varying k had minimal effect, as did reducing the maximum gradient norm to 0.1 or increasing the Lagrangian step size to 0.01.

Initial results for the full capacity were reasonable in terms of the approximation ratio but consistently violated constraints by a large degree (details in Section 3.4). To this end I also sought to train a model which has stronger constraint satisfaction. The model was trained under the same conditions as the previous one, with two differences:

- Experiment with changing the initial value of the Lagrange multiplier within the range $[0, 10]$, and experiment with more aggressive Lagrange step size in the range $[1, 10]$.
- Choose models based on their loss function evaluation instead of the approximation ratio. One problem with this is that the loss function changes between epochs as the Lagrange multipliers are updated, meaning that direct comparison from epoch to epoch may not always be an accurate predictor of model quality. Because of this I also tracked what I called the "constraint normalised" loss, which is the loss function with the Lagrange multiplier fixed at one.

The constraint normalised loss proved somewhat ineffective, and the standard loss tended to produce models with higher constraint satisfaction. That being said I also did not find the loss to be a strong predictor of the quality of the trained models. In the end I manually tested the outputs of a set of models to check how well they balanced constraint satisfaction with accurately approximating the solution. The models I checked were those that minimised the loss during training, it is unlikely that these represented the best models from the training process.

The chosen model was trained with a Lagrange step size of 1 and an initial Lagrange multiplier of 1.

3.4 Results

Tables 3.1, 3.2 and 3.3 report the results on the restricted capacity instances for n=50, 200 & 500.

All figures are reported as a percentage. For constraint violation I report the percentage of instances in which the constraint was violated, and the average violation percentage (taken only over the instances in which the constraint was violated). For the objective statistics I report the percentage of instances in which the predicted objective was under and above the optimal objective, as well as the average undershoot and overshoot calculated only on the instances in which the predicted objective is under/over the optimal objective. Lastly, the approximation ratio is reported. Performance is reported on a test set which was unseen at training time.

For smaller capacity instances there are some extreme outliers in the constraint violation, these have been excluded from the calculation of the mean by only considering instances in which the violation percentage falls within six standard deviations of the mean. These outliers typically occur in instances with extremely small capacities, in these instances one single weight could mean the constraint is violated by a large degree.

	Constraint Violation		*Objective Statistics*				
α	% Violated	Mean Violation	% Under	% Over	Mean Overshoot	Mean Undershoot	AR
0.15-0.25	3.8	10.45	93	3.7	6.2	9.98	1.1107
0.45-0.55	6	7.17	88.4	5.7	3.89	4.59	1.0457
0.75-0.85	8.3	3.68	59.4	7.2	1.1	1.24	1.00828

Table 3.1: $n = 50$

	Constraint Violation		*Objective Statistics*				
α	% Violated	Mean Violation	% Under	% Over	Mean Overshoot	Mean Undershoot	AR
0.15-0.25	4.7	4.22	96.2	3.7	1.85	5.96	1.0629
0.45-0.55	10.8	2.84	90.9	9.1	1.27	2.26	1.0224
0.75 - 0.85	20	2.19	85.2	14.1	0.51	0.6	1.00589

Table 3.2: $n = 200$

	Constraint Violation		*Objective Statistics*				
α	% Violated	Mean Violation	% Under	% Over	Mean Overshoot	Mean Undershoot	AR
0.15-0.25	18.7	3.2	90.9	9.1	1	2.65	1.02604
0.45-0.55	19.1	2.38	90.3	9.7	0.86	1.35	1.01323
0.75-0.85	27.2	2.42	81.1	9.6	0.14	0.34	1.00294

Table 3.3: $n = 500$

The performance on the restricted range capacity instances is generally strong. The model achieves an approximation ratio of under 1.1 in all cases with the exception of low capacity instances with $n = 50$. This demonstrates that the the learned models consistently come close to the optimal solution.

Constraint satisfaction varies across the instance types. For instances with a smaller number of weights the models typically violate the constraints less often, but generally violate the constraint by a larger degree. In instances with a large number of weights, the models violate the constraints more often, but only by a few percentage points on average.

In future work it may be interesting to take a closer look at the distribution of the violation percentages. For example, in instances with violations are the violations typically small with some outliers dominating the mean, or are violations distributed evenly around the mean.

	Constraint Violation		*Objective Statistics*				
α	% Violated	Mean Violation	% Under	% Over	Mean Overshoot	Mean Undershoot	AR
0-0.2	48.75	81.22	54.08	43.93	30.54	16.06	1.273
0.2-0.4	39.52	12.11	62.1	37.1	6.3	8.7	1.0863
0.4-0.6	46.3	8.65	53.73	44.92	4.16	4.55	1.045
0.6-0.8	42.86	5.67	58.12	39.45	2.24	2.28	1.0226
0.8-1	40.3	3.74	56	37.2	0.75	0.85	1.00759
All	43.5	21.7	56.9	40.5	9.3	6.5	1.0876

Table 3.4: Full Range Capacity $n = 50$

Table 3.4 reports results on the full range capacity instance with $n = 50$. The performance on the model trained on a full range of capacities is reasonable in terms of the approximation ratio. Breaking down the test set performance by instance capacity quintiles shows that the primary issue is the poor performance of the model on low capacity instances. Based on the previous set of results on restricted capacity instances this is not particular surprising.

There are a few things to note about these low capacity instances. I suspect that they are fundamentally harder than higher capacity instances in terms of learning a model minimising the approximation ratio. An intuitive way to think about this is that for a low capacity Knapsack there is less room for error in the sense that there are generally very few combinations of items which lead to a near optimal solution.

A possible fix for this is to use more instances, it's likely that 24,000 training instances (corresponding to 4,800 low capacity instances) is not enough to learn a highly performant model. Instead of increasing the size of the entire training set, it may make sense to create an imbalanced training set which is weighted towards lower capacity instances. Some care would need to be taken to ensure that this imbalance does not degrade the performance on higher capacity instances. Generally speaking, this may be a use case for active learning, in which a model can in essence request training data of a specific type [21].

A notable flaw with the model is the poor constraint satisfaction. Across all quintiles the model consistently violated the constraint in at least 40% of instances. For higher capacity instances the violation degree can be relatively small, but this is still undesirable behaviour. The inclusion or exclusion of one item may turn an optimal solution into one which radically violates the Knapsack capacity.

Table 3.5 reports results for a model trained with the intention of improving the constraint satisfaction.

	Constraint Violation		*Objective Statistics*				
α	% Violated	Mean Violation	% Under	% Over	Mean Overshoot	Mean Undershoot	AR
0-0.2	9	309.85	92.85	5.82	121	43.76	1.99
0.2-0.4	0.8	8.71	99.2	0.64	5.55	23.93	1.35
0.4-0.6	5.08	4.76	95.76	3.73	2.77	10.78	1.122
0.6-0.8	11.2	4.08	92.5	7.5	1.54	5.9	1.061
0.8-1	2.82	1.87	98.77	0.88	0.7	3.46	1.0359
All	5.8	97.8	95.8	3.7	39.23	17.62	1.315

Table 3.5: Full Range Capacity $n = 50$ - Aggressive Constraint Satisfaction

The model improves massively on the percentage of instances in which the KP constraint was violated. There is a notable increase in the approximation ratio across all quintiles. The mean constraint violation is reasonable, although the average across all quintiles is perturbed significantly by outliers in the low capacity instances.

This demonstrates an ability to use the Lagrangian step size to trade off average optimality of predictions with constraint satisfaction. This was not done in a principled way and it would be desirable to develop some methodology to choose models which balance these two concerns, and in a more general setting the relative importance of different constraints.

One issue I noticed during training is that in early training epochs, low capacity instances would result in massive constraint violations and the resulting gradient flow would dominate early training. I suspect this because the network weight initialisation results in roughly 50% of weights being active by default. I tried two approaches to remedy this:

- A naive curriculum learning approach in which instances were ordered from high capacity to low capacity. This was ineffective.
- Gradient clipping, which was effective.

While gradient clipping is effective at stabilising training, it's difficult to assess how it affects constraint satisfaction. A more principled approach might be to directly target the gradients flowing back from the constraint violation portion of the loss function. For example normalising the constraint gradients such that the gradient norm is in the same order of magnitude as the binary cross-entropy label loss (See [22] for example).

Chapter 4

Future Works

The next major step will be to look at the application of the Lagrangian Dual Framework to a more complicated mixed-integer linear programming problem. In particular I will be looking at the optimisation problem posed in [23].

At its core it is a scheduling problem. To give a brief (and somewhat incomplete) description, the problem is to schedule a set of classes over the span of one month, some of which are recurring and some of which are one-off. Recurring classes must all be scheduled. The one-off activities do not strictly need to be scheduled, but a cost is incurred for any that are scheduled outside of business hours, and a further cost if they are not scheduled at all. Each class also has some associated power draw and duration.

Forecasts for energy prices and solar loads are available inputs, along with the base load for the buildings within which the classes are held. Batteries are also available which can be used for storing power. The objective is to minimise the total cost of scheduling the classes, where the majority of the cost is incurred by drawing power from the grid. A full description is available in [23] and on this link.

This is clearly a much more complex problem than the knapsack problem. It has orders of magnitude more constraints and decision variables. It also requires continuous decision variables to track the battery charge, as well as strict equality constraints which must be satisfied for a solution to be sensible. By comparison the KP constraint was treated here as fairly weak, and in practice a mildly infeasible solution should be able to be converted to a reasonably optimal feasible solution using a computationally cheap greedy strategy. For the class scheduling problem it's not clear whether mildly infeasible solutions would always be salvageable.

The first step in tackling this problem is to generate training data. Generating instance data is not difficult, early iterations while focus on varying one parameter (E.g. the forecast of solar power or power prices) will keeping the rest of the data fixed between instances. [23] also has some discussion of the hardness of randomly generated instances which may be of use. I have developed multiple MILP models for solving these instances, the fastest of which is a column generation method which enumerates the possible schedules for each class and optimises over the feasible schedules. On my hardware, for small instances (50 recurring classes and 20 one-off classes), column generation can reduce the optimality gap to under 1% within ten minutes, and occasionally solve the instance to optimality before that. For large instances (200 recurring and 100 one-off classes) column generation typically reduces the optimality gap to under 3%, and often around 1%, in ten minutes.

At these speeds generating the labels for a large amount of instances is infeasible. My processor speed is a clear bottleneck, I am also limited by only having 16GB ram. This prevents me from allowing Gurobi to use

multiple threads since having multiples copies of the model on different threads is memory intensive. As a solution I intend to use the Bunya HPC cluster to generate labels for the training data.

Another avenue for consideration is to restrict the size and complexity of the problem such that instances can be solved in a more reasonable timeframe. This would need to be done in a way which retains the interesting aspects of the problem, and even with restrictions the generating solutions may still be infeasible on consumer hardware.

With solutions generated I will begin to look at training models and assessing their performance. Past here it is difficult to anticipate the nature of the work, since it depends on what technical details will prove troublesome.

Some possible areas of focus may be:

- Another mixed-integer linear programming problem, if one is found to be sufficiently different enough to both the knapsack and class scheduling problems to be interesting.
- Understanding how to balance constraint satisfaction and optimality of predicted solutions.
- Investigating more sophisticated methods of instance generation (possible via active learning).
- Understanding how to tune and adapt the LDF for large scale MILP problems.

The degree to which these are explored will largely depend on specific areas of difficulty in the class scheduling problem.

As a rough timeline for the remainder of the work to be completed I propose:

- **June/July** - Wrap up work on the Knapsack Problem and begin generating optimal solution for the class scheduling problem on Bunya.
- **August/September** - Use generated labels to develop models for predicting optimal solutions to the class scheduling problem.
- **October/November** - Compile results and prepare Thesis and seminar presentation. This period will also likely involve follow up work or exploration of ideas stemming from work on the class scheduling problem.

The final thesis will likely have the following chapters:

- Introduction
- Literature Review
- The Lagrangian Dual Framework
- Knapsack Problem
- Class Scheduling Problem
- Conclusion and Future Works

along with additional chapters for any other optimisation problems or technical aspects which warrant their own chapter.

Bibliography

[1] J. Hopfield and D. Tank, "Neural computation of decisions in optimization problems," *Biological cybernetics*, 1985.

[2] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi, "Exact combinatorial optimization with graph convolutional neural networks," 2019.

[3] Y. Tang, S. Agrawal, and Y. Faenza, "Reinforcement learning for integer programming: Learning to cut," *CoRR*, vol. abs/1906.04859, 2019.

[4] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: a methodological tour d'horizon," *CoRR*, vol. abs/1811.06128, 2018.

[5] N. Mazyavkina, S. Sviridov, S. Ivanov, and E. Burnaev, "Reinforcement learning for combinatorial optimization: A survey," 2020.

[6] G. Yehuda, M. Gabel, and A. Schuster, "It's not what machines can learn, it's what we cannot teach," 2020.

[7] S. Geisler, J. Sommer, J. Schuchardt, A. Bojchevski, and S. Günnemann, "Generalization of neural combinatorial solvers through the lens of adversarial robustness," 2022.

[8] F. Fioretto, T. W. K. Mak, and P. V. Hentenryck, "Predicting ac optimal power flows: Combining deep learning and lagrangian dual methods," 2019.

[9] P. L. Donti, D. Rolnick, and J. Z. Kolter, "Dc3: A learning method for optimization with hard constraints," 2021.

[10] F. Detassis, M. Lombardi, and M. Milano, "Teaching the old dog new tricks: Supervised learning with constraints," 2021.

[11] M. Abolghasemi, "The intersection of machine learning with forecasting and optimisation: theory and applications," 2022.

[12] J. Kotary, F. Fioretto, P. V. Hentenryck, and B. Wilder, "End-to-end constrained optimization learning: A survey," 2021.

[13] F. Fioretto, P. V. Hentenryck, T. W. Mak, C. Tran, F. Baldo, and M. Lombardi, "Lagrangian duality for constrained deep learning," 2020.

[14] D. D. Pisinger, H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack problems / Hans Kellerer, Ulrich Pferschy, David Pisinger*. Berlin ; New York: Springer, 2004.

[15] H. A. A. Nomer, K. A. Alnowibet, A. Elsayed, and A. W. Mohamed, "Neural knapsack: A neural network based solver for the knapsack problem," *IEEE Access*, vol. 8, pp. 224200–224210, 2020.

[16] C. Hertrich and M. Skutella, "Provably good solutions to the knapsack problem via neural networks of bounded size," 2021.

[17] D. Pisinger, "Where are the hard knapsack problems?," *Computers Operations Research*, vol. 32, no. 9, pp. 2271–2284, 2005.

[18] D. Pisinger, "Core problems in knapsack algorithms," *Operations Research*, vol. 47, 12 2002.

[19] K. Smith-Miles, J. Christiansen, and M. A. Muñoz, "Revisiting where are the hard knapsack problems? via instance space analysis," *Computers Operations Research*, vol. 128, p. 105184, 2021.

[20] J. K. Eshraghian, M. Ward, E. Neftci, X. Wang, G. Lenz, G. Dwivedi, M. Bennamoun, D. S. Jeong, and W. D. Lu, "Training spiking neural networks using lessons from deep learning," *arXiv preprint arXiv:2109.12894*, 2021.

[21] B. Settles, "Active learning literature survey," Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[22] Y. He, X. Feng, C. Cheng, G. Ji, Y. Guo, and J. Caverlee, "MetaBalance: Improving multi-task recommendations via adapting gradient magnitudes of auxiliary tasks," in *Proceedings of the ACM Web Conference 2022*, ACM, apr 2022.

[23] C. Bergmeir, F. de Nijs, A. Sriramulu, M. Abolghasemi, R. Bean, J. Betts, Q. Bui, N. T. Dinh, N. Einecke, R. Esmaeilbeigi, S. Ferraro, P. Galketiya, E. Genov, R. Glasgow, R. Godahewa, Y. Kang, S. Limmer, L. Magdalena, P. Montero-Manso, D. Peralta, Y. P. S. Kumar, A. Rosales-Pérez, J. Ruddick, A. Stratigakos, P. Stuckey, G. Tack, I. Triguero, and R. Yuan, "Comparison and evaluation of methods for a predict+optimize problem in renewable energy," 2022.