Machine Learning for Integer Programming

Approximating Solutions with the Lagrangrian Dual Framework

Mitchell Keegan

s47365357

November 2023

Acknowledgements

I would like the thank my academic supervisor, Mahdi Abolghasemi. His guidance and advice through the year made this project an enjoyable and rewarding experience.

Abstract

Integer Programming problems are a type of constrained discrete optimisation problem that are of significant practical importance. The computational cost of solving these problems may be prohibitively expensive for some applications. Recent years have seen a growing interest in the use of deep learning methods to approximate the solutions to such problems in a fraction of the time as traditional solvers. A core problem is how to enforce or encourage constraint satisfaction in predicted solutions. A promising approach for predicting solutions to constrained optimisation problems is the Lagrangian Dual Framework which builds on the method of Lagrangian Relaxation. This thesis develops neural networks models which use the Lagrangian Dual Framework to enforce constraint satisfaction while approximating the solution to two Integer Programming problems. The first is the Knapsack problem, on which strong results are obtained demonstrating consistent constraint satisfaction with minimal reduction in the optimality of solutions. The second is a complex class scheduling problem, on which weaker results are obtained demonstrating improved constraint satisfaction on battery charge storage constraints.

Contents

1 Introduction

Constrained optimisation considers problems of the form:

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

In the language of optimisation theory, 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 integer programming (IP) problems, in which decision variables are restricted to integer values, and the objectives and constraints take a linear structure. IP problems take the form:

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

IP problems 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 problems of practical interest exactly (or approximately) with reasonable efficiency. Even so, a common issue when considering IP problems is that solve times may be too slow in applications where solutions are required under strict time constraints. A promising remedy is to use fast inference machine learning models to approximate solutions [1, 2, 3].

The idea of applying machine learning to solve optimisation problems is not a new one, although recent years have seen a resurgence in interest with the rise of deep learning. An early example in 1985 was the use of Hopfield Networks to approximate the objective of the Travelling Salesman Problem [4]. 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 intractable. This has naturally led to renewed interest in the applications of machine learning to optimisation.

This thesis focuses on using supervised deep learning to predict solutions to IP problems. For broad reviews of recent advancements in the applications of machine learning to CO, including reinforcement learning and machine learning embedded in IP solvers, refer to [5], [6], and [7]. A core challenge in machine learning for optimisation is data generation. Solving IP problems is often computationally expensive which makes generating target labels for large datasets difficult. This issue is sidestepped in this thesis by focusing on problems for which dataset generation is feasible, albeit slow, but this problem represents an interesting area of research [7]. For the purpose of approximating solutions to constrained optimisation problems, a key weakness of neural networks is the inability to enforce constraint satisfaction on the model outputs.

Several techniques have been developed to address this weakness. One approach proposes using an iterative training routine in which a learner step that trains a model as usual is interleaved with a master step that

adjusts the target labels to something closer to the predicted solution while remaining feasible [8]. It makes no assumptions about the nature of the constraints and permits any supervised ML model in the learner step. Another approach is Deep Constraint Completion and Correction (DC3) [9]. DC3 uses differentiable processes, named completion and correction, to enforce the feasibility of solutions during training. It showed strong results with a high degree of constraint satisfaction on the AC optimal power flow problem (AC-OPF), which is a continuous optimisation problem. Neither approach has been applied to discrete optimisation problems, and it's not obvious that they could be extended to do so. A third promising approach to integrating constraints is based on Lagrangian Relaxation.

Lagrangian Relaxation is a method of solving constrained optimisation problems by relaxing constraints into the objective function scaled by Lagrange Multipliers. The Lagrangian Relaxation and related concepts can be generalised to allow application to arbitrary optimisation models [10]. The Lagrangian Dual Framework [11, 12] builds on this approach, using Lagrangian relaxation to encourage constraint satisfaction in neural network models during training. The LDF has been applied with encouraging results to the problems of AC-OPF [11], constrained predictors where constraints between samples are present [12], and the job shop scheduling problem [13].

This thesis considers the application of the LDF to two IP problems. The first is the Knapsack Problem, a classic CO problem.This is not the first attempt at predicting KP solutions using neural networks. For example GRU, CNN and feedforward neural networks have been designed for this purpose [14], but these models did not incorporate constraint satisfaction into the training procedure. Some research has been done to derive theoretical bounds on the depth and width of an RNN designed to mimic a dynamic programming algorithm for both exact and approximate solutions [15].

Three models were developed to approximate solutions to KP, one is a baseline fully connected network which is compared to another fully connected network that models the constraints using the LDF. A third model uses the LDF but also uses the baseline neural network as a pre-trained model. This thesis explores implementation details for applying the LDF to KP (and by extension IP problems), experimentally investigate the trade-off between the optimality of approximated solutions against constraint satisfaction, and discuss how to approach hyperparameter tuning and model selection in the context of this trade-off.

The second problem considered is a difficult class scheduling problem, the objetive of which is to create a class schedule which minimises the electricity cost associated with the schedule. Two batteries are available, there are also multiple constraints on how classes can be scheduled, significantly increasing the complexity of the problem. Two neural network models were developed to approximate class schedules, using a network architecture designed to exploit the structure of the problem. The first baseline model does not enforce constraint satisfaction at all while the second model uses the LDF to encourage solutions which do not violate constraints on the charge stored by the batteries. A novel method is proposed to allow for backpropogation through the one-hot encoding of logit vectors and results are reported indicating that this method may allow for the enforcement of constraints related to decision variable which were learnt as a multi-class classification problem.

2 Lagrangian Dual Framework

This section restates the formulation of the LDF [12] as relevant to the two problems considered.

Consider a general constrained optimisation problem with inequality constraints:

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

The violation-based Lagrangian function is:

$$f_\lambda(y) = f(y) + \sum_{i=1}^{m} \lambda_i \max(0, g_i(y)) \tag{2}$$

where $\lambda_i \geq 0$ denote the Lagrange multipliers associated with the inequality constraints. Another approach is to consider the satisfiability-based Lagrangian function:

$$f_\lambda(y) = f(y) + \sum_{i=1}^{m} \lambda_i g_i(y) \tag{3}$$

Both of these can be generalised by considering the function $\nu(g_i)$ which returns either the constraint satisfiability or the degree of constraint violation as required.

$$f_\lambda(y) = f(y) + \sum_{i=1}^{m} \lambda_i \nu(g_i(y)) \tag{4}$$

In this thesis $\nu(g_i)$ will always refer to the constraint violation degree $\max(0, g_i(y))$. The Lagrangian Relaxation is then:

$$LR_\lambda = \underset{y}{\text{argmin}}\, f_\lambda(y) \tag{5}$$

for some set of Lagrangian multipliers $\lambda = \{\lambda_1, \ldots, \lambda_m\}$. The solution 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 $\mathcal{O}$, the best set of Lagrangian multipliers can be found using the Lagrangian dual.

$$LD = \underset{\lambda \geq 0}{\text{argmax}}\, f(LR_\lambda) \tag{6}$$

The LDF leverages the Lagrangian Relaxation to improve constraint satisfaction in neural network models. Consider parametrising the original optimisation problem by the parameters of the objective function and constraints:

$$\begin{aligned} \mathcal{O}(d) = \underset{y}{\text{argmin}} \quad & f(y,d) \\ \text{subject to} \quad & g_i(y,d) \leq 0,\ i = 1,\dots,m \end{aligned} \tag{7}$$

The goal is the learn some parametric model $\mathcal{M}_w$ with weights w such that $\mathcal{M}_w \approx \mathcal{O}$. Here $\mathcal{M}_w$ is always a feedforward neural network.

The set of data is denoted $D = \{(d_l, y_l = \mathcal{O}(d_l)\}$ for $l = 1,\dots,n$. The learning problem is to solve:

$$\begin{aligned} w^* = \underset{w}{\text{argmin}} \quad & \sum_{l=1}^{n} \mathcal{L}(\mathcal{M}_w(d_l), y_l) \\ \text{subject to} \quad & g_i(\mathcal{M}_w(d_l), d_l) \leq 0,\ i = 1,\dots,m,\ l = 1,\dots,n \end{aligned} \tag{8}$$

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 samples, while being such that the output satisfies the constraints for all samples. To this end, relax the constraints into the loss function to form the Lagrangian loss function:

$$\mathcal{L}_\lambda(\hat{y}_l, y_l, d_l) = \mathcal{L}(\hat{y}_l, y_l) + \sum_{i=1}^{m} \lambda_i \nu(g_i(\hat{y}_l, d_l)) \tag{9}$$

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

$$w^*(\lambda) = \underset{w}{\text{argmin}} \sum_{l=1}^{n} \mathcal{L}_\lambda(\mathcal{M}_w(d_l), y_l, d_l) \tag{10}$$

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

$$\lambda^* = \underset{\lambda}{\text{argmax}} \min_{w} \sum_{l=1}^{n} \mathcal{L}_\lambda(\mathcal{M}_w(d_l), y_l, d_l) \tag{11}$$

The LDF implements subgradient optimisation to iteratively solve for w and λ. The training process is summarised in Algorithm 1. It takes in the following inputs: Training data D, number of training epochs n_{epochs}, Lagrangian step size s_i for $i = 1, ..., m$, and initial Lagrange multipliers λ_i^0 for $i = 1, ..., m$.

Algorithm 1 LDF Algorithm

Input: D, n_{epochs}, s_i, λ_i^0
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
  $\lambda_i^{k+1} \leftarrow \lambda_i^k + s_i \sum_{l=1}^{n} \nu_i(g_i(\hat{y}, d_l)), \quad i = 1, ..., m$
end for

3 The Knapsack Problem

Ths section formally introduces the Knapsack Problem, discusses methods of generating training instances, describes neural network architectures for approximation solutions to KP and the procedures used to train them. Issues around backpropogation when evaluating constraints during training are discussed and a solution based on surrogate gradients is introduced. Results showing strong improvements in constraint satisfaction by applying the LDF are presented and issues around model selection are discussed.

3.1 Problem Description

The Knapsack Problem is a classic problem in combinatorial optimisation. The core statement 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 can be defined as an integer programming (IP) problem as shown below:

$$\begin{aligned} x^* = \underset{x}{\text{argmax}} & \sum_{i=1}^{n} x_i v_i \\ \text{s.t.} & \sum_{i=1}^{n} x_i w_i \leq W \\ & x_i \in \{0,1\},\ i = 1, ..., n \end{aligned} \tag{12}$$

where $x_i = 1$ if the i^{th} item is chosen or $x_i = 0$ if not. KP and its variants find applications when limited resources must be allocated efficiently. Examples include cutting stock problems, investment allocation, and transport and logistics. There exists effective exact and approximate algorithms to solve KP including dynamic programming, branch and bound, and a variety of metaheuristic algorithms [16]. The intention of approximating solutions to KP is not to compete with these well established techniques, but to understand the application of the LDF to a relatively simple IP problem.

3.2 Instance Generation

Instances should be generated for training and testing which represent a wide variety of adequately difficult cases. The size of the knapsack capacity relative to the total item weights is used as a proxy for instance difficulty. $30,000$ instances were generated, each with $n = 500$ items from which $24,000$ instances were used for training while 3000 were set aside for testing and validation sets. The item weights and values are uncorrelated and uniformly distributed on $[0, 1]$. A method proposed in [17] is used to generate the instance capacities, where the capacity of the j^{th} instance is set to $W_j = \frac{j}{S+1} \sum_{i=1}^{n} w_i$, S being the total number of instances. This produces data with a full coverage of instance capacities and difficulties, from instances in which the optimal solution have few items to those which have almost all items. Target labels were generated for each instance using Gurobi 9.5.2 on a PC running Ubuntu Linux with an Intel i5-11400 processor.

3.3 Network Architecture and Training Procedure

Three neural network models were trained for approximating solutions to the KP: the first was a fully connected neural network (FC) used as a baseline model, the second model used the LDF to model the

constraints, and the third model used the LDF but also used the FC baseline as a pre-trained model. Equivalently the pre-trained LDF could be viewed as fixing the Lagrange multipliers at zero for some fixed amount of epochs or until the model converges before allowing them to be updated. All models were trained using PyTorch v1.12.1 with Numpy v1.23.5 on a PC running Ubuntu Linux with an Intel i5-11400 processor[1].

An input to the network is the parameters for a single instance. For a KP instance with n items, the parameters are the weights w_i, values v_i and capacity W. The input vector is then $[w_1, \ldots, w_n, v_1, \ldots, v_n, W] \in \mathbb{R}^{2n+1}$

All models had two hidden layers with widths 2048 and 1024, ReLU activation functions, and were trained using the Adam optimiser with default β values. While training all models batch normalization was applied on both hidden layers, a batch size of 256 was used, and the Lagrange multiplier was initialised to 1. Training was performed for 500 epochs. Hyperparameter optimisation was performed over the learning rate, maximum gradient norm, and Lagrangian step size. Values for hyperparameter optimisation are recorded in Table 1.

Model	Learning Rate	Lagrangian Step	Grad Norm
FC	$\{10^{-4}, 10^{-3}\}$	N/A	{1,10}
LDF	$\{10^{-5}, 10^{-4}, 10^{-3}\}$	$\{10^{-8}, 10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}\}$	{0.5,1,10}
Pre-trained LDF	$\{10^{-4}, 10^{-3}\}$	$\{10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}\}$	{1,10}

Table 1: KP Hyperparameter Optimisation Sets

Models were selected based on performance on the validation set. The chosen FC model was the one which minimised the AR, both LDF models were chosen to minimise the 1-loss (a custom metric proposed in Section 3.4). All results are reported on the test set. The chosen FC model was trained with a learning rate of 10^{-3} and maximum gradient norm of 10. The chosen LDF model was trained with a learning rate of 10^{-4}, Lagrangian step size of 10^{-7}, and maximum gradient norm of 0.5. The chosen pre-trained LDF model was trained with a learning rate of 10^{-4}, Lagrangian step size of 10^{-4}, and maximum gradient norm of 10.

A common problem in training the LDF model was exploding gradients originating from the constraint evaluation. The default weight initialisation produces an output with approximately half of the knapsack weights active. As a result, predictions early in the training process can significantly violate the capacity constraint in low capacity instances. Gradient clipping alleviates this, but it's unclear if this may distort the training process and reduce the relative importance of constraint satisfaction in learned models. This issue is much less pronounced in the pre-trained model but is still present. This made the pre-trained LDF models easier to train and more robust to changes in hyperparameters since they did not have the same degree of instability as the LDF models.

3.4 Output Decoding and 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. Binary cross entropy is used for the label portion of the loss function $\mathcal{L}(\hat{y}_l, y_l)$.

Care must be taken in using the model outputs to evaluate the knapsack constraint. At training time the logit outputs must be mapped to binary decision variables. A naive approach is to apply a sigmoid at the output to scale it into the range $[0, 1]$ and then round to zero or one. The problem is that the derivative of the round function is zero everywhere, meaning that informative gradients will not flow back from the constraint evaluation. Instead, a surrogate gradient is substituted in during the backwards pass, similar to methods used to deal with uninformative gradients in spiking neural networks [18]. The gradient of the sigmoid function centred at 0.5 is used as a surrogate with the form:

[1]All code available at https://github.com/mitchellkeegan/Thesis

$$\frac{d\sigma}{dx} = \frac{ke^{-k(x-0.5)}}{(e^{-k(x-0.5)}+1)^2} \tag{13}$$

The parameter k affects how tightly the function is distributed around 0.5 as shown in Figure 1. 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 round function to jump between 0 and 1. Values for k in the range $[5, 35]$ were tested, but did not seem to have much effect in practice. As such $k = 25$ was used for all experiments.

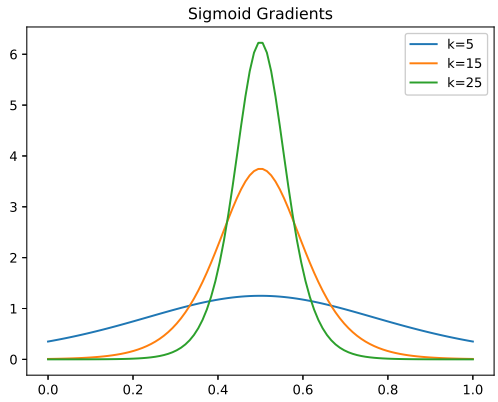

Figure 1: Surrogate Gradient

Rounding is also used at prediction time to decode the model outputs. It would be possible to decode the output into feasible solutions using a greedy algorithm [14], but for general IP problems it may not be obvious how to correct infeasible solutions, or it may be computationally expensive to do so. For this reason, although reasonable decoding schemes exist a minimal decoding scheme which simply rounds the outputs is used to investigate performance under minimal output post-processing. As in [14] the approximation ratio (AR) is used as a metric to evaluate the model. For a set of instances of size S it is defined as:

$$AR = \frac{1}{S}\sum_{j=1}^{S} max\left(\frac{f^*(x_j)}{f(x_j)}, \frac{f(x_j)}{f^*(x_j)}\right) \tag{14}$$

where $f^*(x_j)$ and $f(x_j)$ are the predicted and true optimal 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. The approximation ratio is then defined to be equal to two whenever this occurs. While arbitrary, the idea is that a strong predictor should have an approximation ratio approaching one and this represents a reasonable penalty for incorrect predictions. Regardless, instances where one of either $f^*(x_j) = 0$ or $f(x_j) = 0$ are rare and should not significantly influence results.

While the approximation ratio is important, for the purpose of model validation it only evaluates the optimality of predictions without considering constraint satisfaction. Since the Lagrangian multipliers update on each epoch, the loss $\mathcal{L}_\lambda(\hat{y}_l, y_l, d_l)$ may increase between epochs even as overall model performance is improving which makes it a flawed metric for comparing performance across epochs. To this end, I introduce the μ-loss, which replaces the Lagrange multiplier in the loss function with fixed μ values as shown below:

$$\mathcal{L}_\mu(\hat{y}_l, y_l, d_l) = \mathcal{L}(\hat{y}_l, y_l) + \mu\nu(g_i(\hat{y}_l, d_l)) \tag{15}$$

In choosing a model which minimises the μ-loss it should be possible to select for models which place higher relative priority on prediction optimality or constraint satisfaction as needed by varying μ. In practice model selection will depend on the specific application and the relative importance of optimality and constraint satisfaction.

3.5 Empirical Results

Table 2 reports results for the baseline FC model. All figures are reported as a percentage. For constraint violation, the percentage of instances in which the constraint was violated and the average violation percentage are both reported. The average violation is calculated only over the instances in which the constraint is violated. It also filters out extreme outliers that can occur in very low capacity (those in which the knapsack capacity is extremely small compared to the sum of the item weights) instances by ignoring any instances in which the violation is more than six standard deviations from the mean violation. For the objective statistics, the percentage of instances in which the predicted objective was under and above the optimal objective, and the average undershoot and overshoot (labelled Avg-O and Avg-U) are reported. Lastly, the approximation ratio is reported. Performance is broken down into quintiles by instance capacity relative to total item weight denoted $\alpha = \frac{W_j}{\sum_{i=1}^{n} w_{ij}}$ where w_{ij} and W_j are the weight of the i^{th} item and the knapsack capacity respectively in the j^{th} instance.

	Constraint Violation		*Objective Statistics*				
α	% Violated	Mean Violation	% Under	% Over	Avg-O	Avg-U	AR
0-0.2	64.7	105.7	64.1	35.9	39.1	13.6	1.254
0.2-0.4	66.1	8.77	56.8	43.2	3.26	4.49	1.0417
0.4-0.6	95.42	9.9	14.92	85.1	3.17	1.36	1.029
0.6-0.8	99.84	10.4	8.6	91.4	1.97	0.55	1.0185
0.8-1	100	7.55	3.17	96.83	0.66	0.11	1.0064
All	84.93	21.14	30	70	5.89	7.76	1.0703

Table 2: Baseline Neural Network Performance

The FC model shows strong performance on the AR. Performance is relatively poor on low capacity instances. It would be expected that low capacity instances are inherently harder since the inclusion or exclusion of a single item could be the difference between a near optimal solution and a poor solution. This is reflected in the outsized mean violation on low capacity instances. This gives a distorted view of performance on these instances that is not present when the constraint violation is considered in absolute terms. Rates of constraint violation are extremely high across all quintiles, reaching 100% in the highest capacity quintile. This reflects that the constraints are not explicitly modelled during learning. Table 3 reports results on the LDF model.

Constraint satisfaction is significantly improved in the LDF model. Across all instances, the capacity constraint is violated only 2.13% of the time, a significant improvement on the 85% violation rate reported on the FC model. These constraint violations exclusively occur in lower capacity instances, moderate to high capacity instance recorded no constraint violations whatsoever. This comes at some cost to the approximation ratio, but the increase is relatively minor, on the order of 10%. Table 4 reports results on the pre-trained LDF model.

The performance of the pre-trained LDF model is very similar to the base LDF model, with no clear indication

	Constraint Violation		*Objective Statistics*				
α	% Violated	Mean Violation	% Under	% Over	Avg-O	Avg-U	AR
0-0.2	10.48	195	96.8	3.16	113.2	30.35	1.5643
0.2-0.4	0.16	2.93	100	0	N/A	12.73	1.1521
0.4-0.6	0	N/A	100	0	N/A	8.46	1.0931
0.6-0.8	0	N/A	100	0	N/A	5.88	1.0628
0.8-1	0	N/A	100	0	N/A	2.62	1.027
All	2.13	191	99.4	0.6	113.24	12	1.1811

Table 3: LDF Neural Network Performance

	Constraint Violation		*Objective Statistics*				
α	% Violated	Mean Violation	% Under	% Over	Avg-O	Avg-U	AR
0-0.2	14	278	93.8	6.2	92.1	30.4	1.51
0.2-0.4	0.32	1.93	100	0	N/A	2.38	1.1894
0.4-0.6	0.17	1.38	100	0	N/A	8.78	1.09732
0.6-0.8	0	N/A	100	0	N/A	5.29	1.0562
0.8-1	0	N/A	100	0	N/A	2.37	1.0245
All	2.9	268	98.8	1.2	92.1	12.4	1.177

Table 4: Pre-trained LDF Neural Network Performance

that either model is strictly better. As noted previously training was found to be significantly easier with the pre-trained model. In particular, it was found to be more robust with respect to hyperparameter changes, and alleviated the strong dependence on gradient clipping to deal with exploding gradients.

The computational time required for training and generating predictions is also of interest, particularly in comparison to traditional solvers. The total epochs and time taken for the three models to converge during training are listed in Table 5. For the pre-trained LDF model these values include the pre-training time. The LDF model takes significantly longer to converge fully. Prediction times for the neural network models, averaged over the full set of 30,000 instances, took 1ms per instance. Gurobi on the same set of instances takes on average 5.7ms per instance, noting that Gurobi is used here for convenience but other methods may be significantly faster at solving KP. Prediction time could likely be further reduced by using a GPU, with the caveat that in reality it may not be useful to solve more than one instance at a time. More generally for harder IP problems (or harder KP instances) it would be expected that predictions generated by neural networks are significantly faster.

	FC	Pre-trained LDF	LDF
Epochs	16	46	250
Time (Minutes)	0.7	2.6	16

Table 5: Training Time for Neural Network Models

These results demonstrate an ability to trade-off optimality for constraint satisfaction by using the LDF. In LDF models poor performance is mostly concentrated in low capacity instances, moderate and high capacity instances achieve strong performance in terms of the AR without violating any constraints on the test set. It's likely that the unconstrained FC model achieves strong performance in terms of the approximation ratio by consistently violating the capacity constraint by a moderate amount. Enforcing constraint satisfaction reduces or removes these violations but does not directly help the model learn the weights in the optimal solution leading to a small increase in the AR. This gives rise to a trade-off between optimality and constraint satisfaction. It's not obvious to what degree this relationship would generalise to other IP problems, and in

fact experiments applying LDF to the job shop scheduling problem report impressive improvement in both optimality and constraint satisfaction [13].

4 Class Scheduling Problem

This section formally introduces the class scheduling problem, discusses methods for generating training instances and introduces a neural network architecture to approximate schedules given prices for grid power. A surrogate Jacobian is proposed for one-hot encodings of logit vectors to allow for backpropogation through one-hot encodings. Results show poor performance in terms of the approximation ratio, with some improvement in the satisfaction of battery storage constraints after implementing the LDF.

4.1 Problem Description

The second problem considered is a class scheduling problem (CSP) originally posed as a part of a predict and optimise challenge [19]. The core problem is to schedule a set of classes over the course of a month in 15 minutes intervals. A subset of the classes are designated as recurring, these classes have to be scheduled and take place at the same time every week. Other activities are designated as one-off. These do not need to be scheduled but failing to do so incurs a cost. Each of these classes has an associated power draw, duration, number of rooms used, and a list of other classes which must be scheduled before it can take place. There is also a maximum number of rooms available that limits how many classes can be scheduled at any given time.

A key element of the problem is managing power storage and consumption. Power is consumed by scheduled classes and the buildings have a base load. Power is supplied by solar panels free of charge, or from grid power at cost. There are also batteries available which may be charged or discharged in each timeslot. Grid power is the primary cost, followed by penalties incurred by failing to schedule one-off classes. The objective is to find a schedule which minimises this cost.

This problem is of significantly higher complexity than KP, requiring orders of magnitude more decision variables and constraints to model. CSP can be formally defined as shown below, solved using a priori column generation:

Sets

A	Activities
$A_r \subseteq A$	Recurring Activities
$A_o \subseteq A$	One-off Activities
T	Time periods in whole month
T^{bus}	Time periods which fall within 9am-5pm, M-F
$T^{off} = T \setminus T^{bus}$	Time periods which fall outside of 9am-5pm, M-F
$T^r = \{1, ..., 32\}$	Time period for recurring activities within a given day
$T^o = \{1, ..., 96\}$	Time period for one-off activities within a given day
$D^r = \{1, ..., 5\}$	Days in the week
$D^o = \{1, ..., 30\}$	Days in the month
B	Set of Batteries
K_a	Set of feasible schedules for class $a \in A$

The set of feasible schedules K_a is effectively the set of start times for which the class could start. In the case of the recurring classes this prevents them from being scheduled such that they run outside of business hours. When the schedules are generated any required data associated with the scheduled is also calculated.

Data

dur_a	Duration of activity $a \in A$
n^{small}	Number of small rooms available
n^{large}	Number of large rooms available
p_t^{base}	Base load at time $t \in T$
p_t^{solar}	Solar supply at time $t \in T$
$price_t$	Electricity cost at time $t \in T$
$small_{adt}^k$	Number of small rooms used by activity $a \in A$ on day $d \in D^{o/r}$ in time period $t \in T^{o/r}$ in schedule $k \in K_a$
$large_{adt}^k$	Number of small rooms used by activity $a \in A$ on day $d \in D^{o/r}$ in time period $t \in T^{o/r}$ in schedule $k \in K_a$
r_{adt}^k	Equal to $small_{adt}^k + large_{adt}^k$, total number of rooms used by activity $a \in A$ on day $d \in D^{o/r}$ in time period $t \in T^{o/r}$ in schedule $k \in K_a$
$value_a^k$	Value gained by choosing schedule $k \in K_a$ for activity $a \in A^o$ (incorporates penalties for activities scheduled outside of business hours)
p_{adt}^k	Power consumption of class $a \in A$ on day $d \in D^{o/r}$ in time period $t \in T^{o\setminus r}$ in schedule $k \in K_a$
$prec_a$	Set of activities which must be scheduled on a day before activity $a \in A$ is scheduled
$active_{ad}^{o/r}$	Set of activities and schedules $(a,k) \in A^{o/r} \times K_a$ which are active on day $d \in D^{o/r}$ at time $t \in T^{o/r}$
$G_{ad}^{o/r}$	Set of active schedules $k \in K_a$ for activity $a \in A^{o/r}$ on day $d \in D^{o/r}$.
eff_b	Efficiency of battery $b \in B$
cap_b	Maximum Capacity of battery $b \in B$
m_b	Maximum power of battery $b \in B$

Functions

$T2Tr(T)$	Maps subsets of T to the corresponding $(d,t) \in D^r \times T^r$ tuples used to index recurring activities.
$T2To(T)$	Maps subsets of T to the corresponding $(d,t) \in D^o \times T^o$ tuples used to index one-off activities

Variables

$\varepsilon_a^k \in \{0,1\}$	Equals 1 if schedule $k \in K_a$ is chosen for activity $a \in A$
p_t^{grid}	Grid supply at time $t \in T$
p_t^{class}	Total power demand from classes at $t \in T$
C_{bt}	Charge stored by battery $b \in B$ at time $t \in T$
$z_{bt}^d \in \{0,1\}$	Equals 1 if battery $b \in B$ is discharging at time $t \in T$
$z_{bt}^c \in \{0,1\}$	Equals 1 if battery $b \in B$ is charging at time $t \in T$

Objective

Note that the original problem as posed also included a peak load demand charge proportional to the square of the peak grid power over the entire month. In the interest of simplifying the problem this has been

excluded here.

$$\min \sum_{t \in T} \frac{0.25}{1000} p_t^{grid} price_t - \sum_{a \in A^o} \sum_{k \in K_a} \varepsilon_a^k value_a^k$$

Constraints

$$\sum_{k \in K_a} \varepsilon_a^k = 1 \qquad \forall a \in A^r \tag{c1}$$

$$\sum_{k \in K_a} \varepsilon_a^k \leq 1 \qquad \forall a \in A^o \tag{c2}$$

$$\mathbb{I}(t \in T^{bus}) \sum_{a \in A_r} \sum_{k \in K_a} \varepsilon_a^k small_{ad^r t^r}^k + \sum_{a \in A_o} \sum_{k \in K_a} \varepsilon_a^k small_{ad^o t^o}^k \leq n^{small} \qquad (d^r, t^r) \in T2Tr(t), (d^o, t^o) \in T2To(t), \forall t \in T \tag{c3}$$

$$\mathbb{I}(t \in T^{bus}) \sum_{a \in A_r} \sum_{k \in K_a} \varepsilon_a^k large_{ad^r t^r}^k + \sum_{a \in A_o} \sum_{k \in K_a} \varepsilon_a^k large_{ad^o t^o}^k \leq n^{large} \qquad (d^r, t^r) \in T2Tr(t), (d^o, t^o) \in T2To(t), \forall t \in T \tag{c4}$$

$$\mathbb{I}(t \in T^{bus}) \sum_{a \in A_r} \sum_{k \in K_a} \varepsilon_a^k p_{ad^r t^r}^k (r_{ad^r t^r}^k) + \sum_{a \in A_o} \sum_{k \in K_a} \varepsilon_a^k p_{ad^o t^o}^k (r_{ad^o t^o}^k) = p_t^{class} \qquad (d^r, t^r) \in T2Tr(t), (d^o, t^o) \in T2To(t), \forall t \in T \tag{c5}$$

$$|prec_a| \sum_{k \in G_{ad}^r} \varepsilon_a^k \leq \sum_{\substack{a' \in prec_a \\ d' \leq d \\ k \in G_{a'd'}^r}} \varepsilon_a^k \qquad \forall a \in A^r, \forall d \in D^r \tag{c6}$$

$$cap_b - m_b z_{b1}^d = C_{b1} \qquad \forall b \in B \tag{c7}$$

$$z_{bt}^c + z_{bt}^d \leq 1 \qquad \forall b \in B, t \in T \tag{c8}$$

$$C_{b,t-1} + m_b (z_{bt} - z_{bt}) = C_{bt} \qquad \forall b \in B, \forall t \in T \setminus \{1\} \tag{c9}$$

$$0 \leq C_{bt} \qquad \forall b \in B, t \in T \tag{c10}$$

$$C_{bt} \leq cap_b \qquad \forall b \in B, t \in T \tag{c11}$$

$$p_t^{grid} + p_t^{solar} + \sum_{b \in B} m_b \left(\sqrt{eff_b} z_{bt}^d - \frac{z_{bt}^c}{\sqrt{eff_b}} \right) = p_t^{class} + p_t^{base} \qquad \forall t \in T \tag{c12}$$

Constraint c1 enforces that each recurring activity must be scheduled exactly once. Constraint c2 enforces that each one-off activity can be scheduled at most once. Constraints c3 and c4 constrain the number of small and large rooms being used in any given time period. Constraint c5 calculates the total power demand from scheduled classes in each time period. Constraint c6 enforces the precedence constraints for each activity. Constraint c7 fixes the initial battery charge to the battery capacity minus any discharge in the first time period. Constraint c8 ensures that each battery cannot simultaneously charge and discharge. Constraint c9 relates the charge stored in the battery to the charge and discharge variables. Constraints c10 and c11

constrain the battery charges to be positive and less than the battery capacities. Constraint c12 links the power supply and power demand.

4.2 Instance Generation

Unlike KP, generating instance solutions is a computationally expensive task for CSP. The original challenge provided small and large instances. Small instances contained 50 recurring classes and 20 one-off classes, while large instances contain 200 recurring and 100 one-off classes. To ease the computational cost a reasonably easy small instance was chosen as a basis for instance generation[2].

To simplify the problem, all data except for the grid power price was fixed. Grid price time series were generated for each new instance using the following procedure:

1. Decompose the grid power p_t using an STL decomposition [20] into seasonal, trending, and residual components using the statmodels package [21] (with a seasonal period of one day) such that $p_t = s_t + t_t + r_t$.
2. For each new instance j, randomly sample (with replacement) 30 contiguous blocks each the size of one day. Concatenate these blocks to obtain a new residual r_t^j
3. Construct the grid power for each instance j as $p_t^j = s_t + t_t + r_t^j$

The residuals are sampled in blocks to preserve potential autocorrelation in the residual time series. Figure 2 plots examples of artificial data generated in this fashion overlayed on the original time series. All instances have 2880 time slots in which to schedule classes, 50 recurring activities, 20 one-off activities, and two batteries.

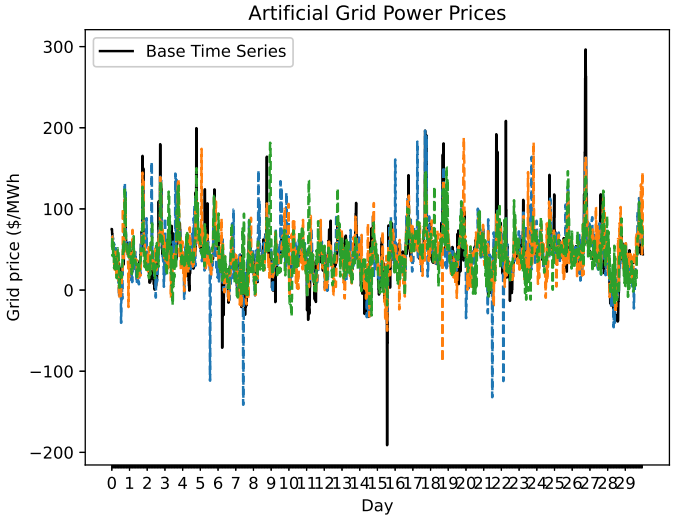

Figure 2: Artificial Grid Power Data

[2]The instance used as base was small instance 0 in phase 2. The challenge data can be downloaded from https://ieee-dataport.org/competitions/ieee-cis-technical-challenge-predictoptimize-renewable-energy-scheduling

10,000 instances were generated in this fashion, of which 8000 were used for training while 1000 were set aside for test and validation sets. Solving the optimisation model to optimality is difficult for most instances, as such instances were only solved to within a 1% optimality gap. This allowed most instances to be solved within 15 seconds including time spent loading data and building the model which made generating the dataset feasible. Target labels were generated for each instance using Gurobi v10.0.1 on the Bunya HPC cluster with an AMD epyc3 Milan processor.

4.3 Network Architecture and Training Procedure

Two neural network models were trained: the first was a fully connected neural network (FC) used as a baseline model. The second model uses the FC model as a base model and trained it further while applying the LDF to encourage constraint satisfaction. Since experiments on the Knapsack problem indicated that training a model from scratch using the LDF was less stable this was not attempted. Both models were trained using PyTorch v2.0.1 on a PC running Ubuntu Linux with an Intel i5-11400[3].

An input to the network is the data for a given instance, which is the grid power price and the other features which are fixed across instances.

The learning problem may be viewed as a multi-task learning problem with 72 tasks each corresponding to one of the 50 recurring activities, 20 one-off activities or two batteries. Each recurring task learns the start time of that activity. Since there are 160 possible start times (32 fifteen minute timeslots per day over five days) each recurring task is a 160-class classification problem. Similarly the one-off tasks are 2881-class classification problems, with 2880 classes for the possible timeslots and an extra class to represent when the activity is unscheduled. Each battery task represents 2880 different three class classification problems. At all 2880 timeslots it must classify the battery into one of three states; charging, discharging, or neither.

As is common in multi-task learning problems shared layers are used to extract features and reduce dimensionality before data is passed into the task-specific networks. The models can be broken up into the following components:

- A fully convolutional network (FCN) which acts as a time series feature extractor inspired by [22]. It has three convolutional layers, each with batch normalisation and a ReLU activation function. The three layers have 64, 128, and 64 channels respectively each with kernel sizes 3, 5, and 8. The input to the FCN has three channels, one for the electricity price and one each for the base load and solar supply time series. Finally a max pooling layer with a kernel size of 32 reduces the dimensionality of the output.
- The second block takes the feature vector outputted by the FCN and concatenates the remaining data. This is then inputted into to a feature mixer MLP. The feature mixer is a standard fully connected network with two hidden layers with widths 2048 and 1024, ReLU activation functions and batch normalisation. It outputs a vector of size 1024 which is then fed to the output networks.
- The third block consists of a set of fully connected neural networks, one for each task. Each task specific network has two hidden layers with widths 512 and 256, ReLU activation functions and batch normalisation.

Much of the architecture is motivated by a need to reduce the number of parameters in the model. The max pooling layer and feature mixer in particular are motivated by dimensionality reduction. The task specific networks are relatively small compared to the dimensionality of their output. It may be that they do not

[3]All code available at https://github.com/mitchellkeegan/Thesis

possess the expressivity required for the learning problem but larger networks were found to be impractical. As it is this model architecture has 89 trainable million parameters, which makes iterating on the architecture and hyperparameter optimisation a difficult task when training on CPU.

Both models were trained for 30 epochs using the Adam optimiser with default β parameters and a batch size of 256. The unconstrained FC model was trained with a learning rate of 10^{-3} and without gradient clipping. The LDF model was trained using the FC model as a base. Hyperparameter optimisation was performed over the learning rate, maximum gradient norm, and Lagrangian step size. Values for hyperparameter optimisation are recording in Table 6.

Model	Learning Rate	Lagrangian Step	Grad Norm
Pre-trained LDF	$\{10^{-4}, 10^{-3}\}$	$\{10^{-4}, 10^{-3}, 10^{-2}\}$	{1,10}

Table 6: Class Scheduling LDF Model Hyperparameter Optimisation Set

Both models were chosen based on performance on the validation set. The FC model chosen was the one which minimised the AR over the training procedure. The LDF model was chosen to minimise the 1-loss. The chosen LDF model was trained with a learning rate of 10^{-3}, Lagrangian step size of 10^{-3}, and maximum gradient norm of 1.

4.4 Output Decoding and Evaluation

The models are considered to have three outputs in $\mathbb{R}^{160\times 50}$, $\mathbb{R}^{2881\times 20}$, and $\mathbb{R}^{3\times 2881\times 2}$. These correspond to the recurring activities, one-off activities and batteries respectively. The cross entropy loss is used for the label portions of the loss function $\mathcal{L}(\hat{y}_l, y_l)$.

As was the case for the Knapsack problem, decoding the outputs for the purpose of constraint evaluation during training poses a challenge since it must be done in a way which permits backpropogation. Here I propose a surrogate Jacobian for the one-hot encoding of a softmaxxed output logit vector. Consider the one-hot function $f : \mathbb{R}^n \to \mathbb{R}^n$ which maps a softmaxxed logit vector z to a one-hot vector $s = f(z)$ which has zeros everywhere except for one in the index which maximises z. Denoting the index of the maximum and penultimate values in z as max & pen, the following intuition might be useful to construct a surrogate:

- When $i \neq max$, the partial derivative of s_i with respect to z_j should only be nonzero for $j = i$ and $j = max$. This is because s_i only changes if z_i increases enough or z_{max} decreases enough for s_i to become to maximum value.
- The nonzero partial derivatives of s_i should be in some way inversely proportional to $z_{max} - z_i$.
- The partial derivative of s_{max} with respect to z_j should be nonzero for all j. It should again be inversely proportional to $z_{max} - z_j$. The exception is when $j = max$, in which case it should be inversely proportional to $z_{max} - z_{pen}$.

This leads to the following definition of the partial derivatives of f, where $\sigma_f(z_i, z_j)$ is some function which is inversely proportional to the distance between its arguments:

$$\frac{\partial s_i}{\partial z_j} = \begin{cases} \begin{cases} -\sigma_f(z_i, z_{max}) & \text{if } j = max \\ \sigma_f(z_i, z_{max}) & \text{if } j = i \\ 0 & \text{otherwise} \end{cases} & \text{if } i \neq max \\ \begin{cases} \sigma_f(z_{pen}, z_{max}) & \text{if } j = max \\ -\sigma_f(z_j, z_{max}) & \text{otherwise} \end{cases} & \text{if } i = max \end{cases} \tag{16}$$

For $\sigma_f(z_i, z_j)$ I propose reusing the gradient of the sigmoid function centered at z_j and evaluated at z_i. Figure 3 plots σ_f for different values of z_{max}. A value of $k = 20$ was used for all experiments. The gradient is highest at inputs closest to z_{max}, indicating that small changes in the input may cause the output to jump between 0 and 1.

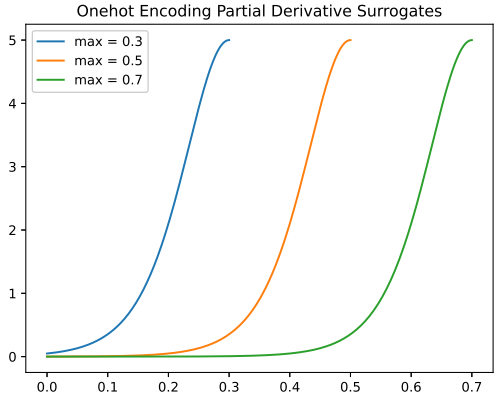

Figure 3: One-hot Partial Derivative Surrogate

To implement a surrogate gradient, Pyorch requires the implementation of the Jacobian vector product to be used during the backward pass through the network. This is easy to do for the one dimensional case but it is non-trivial to vectorise the operation for batched inputs. As such it was implemented used Pytorch's vmap function, which vectorises functions over new dimenions. An example implementation is provided in the Appendix. This provides a massive speedup in the forward pass compared to iterative application of the one dimensional version, although it's not clear how performant the backward pass is in comparison to a standard vectorised implementation.

At inference time the model outputs are decoded by the argmax function. The activity start times and the battery states can be used to infer all other variables of interest. No attempt is made to construct a feasible solution from the output, and such a decoding scheme is likely to be non-trivial.

The approximation ratio is used again with a slight modification. Since target labels were generated with a 1% optimality gap, the true objective values are not known. Consider the optimal objective value in the target solution to be an upper bound z_P. Given a MIPGap G a lower bound on the optimal objective is $z_D = z_P(1-G)$. Using the optimality gap provided by Gurobi the true optimal objective in the j^{th} instance can be bounded by $f(x_j) \in [z_D^j, z_P^j]$. This can be used to calculate an upper bound on the approximation ratio for a given instance. Denoting the predicted objective value of the j^{th} instance as $f^*(x_j)$ and the midpoint of z_D^j and z_P^j as z_M^j, partition the set of instances into $O^- = \{j | f^*(x_j) < z_M^j\}$ and $O^+ = \{j | f^*(x_j) > z_M^j\}$. For $j \in O^-$ an upper bound on the approximation ratio is $\frac{z_P}{f^*(x_j)}$, similarly for $j \in O^+$ an upper bound is

$\frac{f^*(x_j)}{z_D}$. The approximation ratio is then taken to be:

$$AR = \frac{1}{|O^+|} \sum_{j \in O^+} \frac{f^*(x_j)}{z_D} + \frac{1}{|O^-|} \sum_{j \in O^-} \frac{z_P}{f^*(x_j)} \tag{17}$$

Provided the MIPGap is relatively small this should be a reasonably tight bound.

4.5 Constraint Evaluation

Compared to KP, evaluating the constraints during training is non-trivial for CSP. Some constraints are implicitly satisfied by virtue of how the network outputs are interpreted. In general by posing the learning problem as a classification problem binary constraints are automatically satisfied. In the case of CSP, constraint (c8) on the battery state is automatically satisfied since each battery is not permitted to charge and discharge simultaneously. Similarly constraints (c1) and (c2) are satisfied since the recurring activities are scheduled exactly once and the one-off activities are scheduled at most once. Many of the remaining constraints only link decision variables and are used to reconstruct values of interest. Of the remaining constraints only the bounds on the battery charge (c10) and (c11) are integrated into the model via the LDF.

z^c_{bt} and z^d_{bt} can easily be retrieved from the one-hot encodings. Then C_{bt} can be reconstructed from constraints (c7) and (c8). Both constraints of interest are incorporated into the same constraint violation term:

$$\nu_b(\hat{y}) = \frac{1}{|B||T|} \sum_{\substack{b \in B \\ t \in T}} (\nu(-C_{bt}) + \nu(C_{bt} - cap_b))) \tag{18}$$

where $\hat{y}$ is the model output and recalling that $\nu(g) = \max\{0, g\}$. Even though technically the constraint violation could consider the constraints for each battery separately with two Lagrange multipliers, it is convenient to combine them. In this case the battery capacities are on a similar scale (150 and 420) so it should be reasonable to do so.

The loss function augmented with the Lagrangian relaxation of the battery capacity constraints is then:

$$\mathcal{L}_\lambda(\hat{y}, y, d) = \frac{1}{n} \sum_{l=1}^{n} \mathcal{L}(\hat{y}_l, y_l) + \frac{\lambda_b}{n} \sum_{l=1}^{n} \nu_b(\hat{y}_l) \tag{19}$$

using the notation introduced in Section 2 and where λ_b is the Lagrange multiplier associated with the battery capacity constraints. At the end of each training epoch the Lagrange multiplier is updated by:

$$\lambda_b = \lambda_b + \frac{s_b}{n} \sum_{i=1}^{n} \nu_b(\hat{y}_l) \tag{20}$$

where s_b is the associated Lagrangian step size.

The remaining constraints are (c3) and (c4), which bound the number of available rooms, and (c6) which enforces that the activity precedence constraints. Evaluating c3 and c4 using one-hot encodings of start

times is challenging but likely possible by posing the constraint as it might be formulated in an IP problem. Evaluating (c6) using the one-hot encodings, especially when considering that the precedence constraints operate over days and not time periods.

Another option for learning start times is to consider a regression problem instead of a classification problem, using the mean squared error between the true and predicted start times as a loss function [13]. For CSP this may be an easier learning problem due to the significantly lower dimensionality of the outputs, but it presents two issues. It implies an ordering on the time periods which is not always true for recurring activities, since adjacent time periods may actually be split across different days. The second issue is that an extra binary classification task would be required to indicate when one-off activities are unscheduled. Evaluating (c6) would be significantly easier in this case if applying the precedence constraints over time periods instead of days.

4.6 Empirical Results

Table 7 reports the constraint violation statistics for the baseline FC model. 'Battery Min or Capacity' is equivalent to $\nu(-C_{bt}) + \nu(C_{bt} - cap_b)$, combining both battery storage constraints. Results are reported over all iterations of the constraint (E.g. if a constraint exists for all $t \in T$ there will be T versions of the constraint per instance) and all instances.

The percentage violated columns report how often the given constraint is violated, taking the mean and the maximum over all instances. The violation degree columns report the magnitude of constraint violation, with the mean column giving the average over all iterations of the constraint and all instances, calculated only over constraints which were violated. The average max column reports the average over all instances of the maximum degree of violation over all constraint iterations. For a sense of scale, the battery capacities were 150 and 420 for the two batteries, and there are at most 10 small rooms and 6 large rooms. The activities with precedence sets have an average of 4 and maximum of 15 activities in their precedence set.

	Percentage Violated		*Violation Degree*	
	Mean	Max	Mean	Average Max
Battery Min	39.8	99.1	899.4	2169.3
Battery Capacity	49.7	88	1511.6	3465.4
Battery Min or Capacity	89.5	99.1	1239.6	3691.1
Small Rooms	1.6	3.5	2.7	5.1
Large Rooms	0.3	1.2	1.4	1.1
Precedence	47.1	61.4	3.1	10.9

Table 7: Baseline Neural Network Performance

Constraint satisfaction is poor as expected. The solutions predicted by the model regularly violate the battery charge constraints by a large degree. In almost 90% of timeslots the batteries either have negative charge or are charged above capacity. The constraints on the number of rooms are present in all time periods $t \in T$ which distorts how often they appears to be violated, in practice rooms are only ever likely to be overbooked during business hours. The model attempts to use more small rooms than is available in on average 46 time periods, totalling 11 hours overbooked during the month. The precedence constraints are similarly distorted, since only 70% of activities actually have a precedence set. Table 8 reports the performance of the LDF model which attempts to enforce satisfaction of the battery storage constraints.

The results indicate that the LDF has some effectiveness in enforcing the battery storage constraints. The average and maximum violation degrees are significantly lower, although still severe. The batteries are

	Percentage Violated		*Violation Degree*	
	Mean	Max	Mean	Average Max
Battery Min	27.5	33.8	158	660.4
Battery Capacity	30.5	47.7	209.1	840.1
Battery Min or Capacity	58	71.8	184.8	840.2
Small Rooms	2.1	3.4	2.7	5.1
Large Rooms	0.1	0.9	1.1	0.2
Precedence	52	60	3.2	14.1

Table 8: LDF Performance

negatively charged or charged above capacity in 58% of instances. As expected there is no significant difference in the other constraints.

Table 9 reports the average, maximum and minimum approximations ratio for the two models across all instances. The optimality results are fairly poor. The nature of the problem means that any solution which schedules all recurring classes will have an approximation ratio somewhere between 1.2 and 1.3, neither model improves substantially on this. Strict enforcement of the battery storage constraints may improve optimality. Since both models tend to overcharge which typically means using grid power at cost. The converse is true for discharging below zero, but the models appear to be somewhat biased towards overcharging.

	Mean	Max	Min
FC	1.136	1.189	1.085
LDF	1.177	1.133	1.267

Table 9: Model Prediction Approximation Ratios

Table 10 reports the epochs taken to converge and the total training time for each mode. The reported values for the LDF model include the time spent pre-training the unconstrained model. Per epoch training the LDF model is approximately 30% slower, although it appears to converge within a similar number of epochs. Much of this extra time is spent evaluating the constraints the Lagrangian relaxation or to calculate the approximation ratio. Averaged over the validation set, the neural network models take on average 2.4ms. This is a significant improvement over Gurobi which takes on average 10 seconds (or 15 seconds if including time spent building the model). The prediction time should also be fixed, whereas for harder instances of CSP Gurobi may be unable to find near optimal solutions in a reasonable period of time.

	FC	LDF
Epochs	17	32
Time (Minutes)	31	42

Table 10: Training Time for Neural Network Models

5 Conclusions and Future Work

First this paper investigated using neural networks to approximate solutions to the Knapsack Problem while committing to the constraints. It experimentally demonstrated that the Lagrangian Dual Framework could be used to strongly encourage satisfaction of the knapsack capacity constraint with a reasonably small reduction in the optimality of predicted solutions. It discussed issues in interpreting neural network outputs as solutions to integer programming problems and evaluating their performance, particularly in the context of model validation. It also demonstrated that using a pre-trained neural network as a base model may alleviate problems with exploding gradients found in the LDF model.

Next, the LDF was applied to complicated class scheduling problem. The results on the problem were much weaker than those on KP, failing to achieve strong results in optimality or constraint satisfaction. It was demonstrated that a surrogate Jacobian for the one-hot encoding of a logit vector might be viable as a means to allow backpropogation through one-hot encoding functions, being used successfully to encourage satisfaction of battery storage constraints.

There are many avenues to consider for improving the quality of approximated solutions to CSP:

- It is not clear that the network architecture has sufficient expressivity to learn the optimisation problem. In particular the task-specific output networks are relatively small and it is difficult to increase their size without either reducing the size of the model's internal representations or greatly increasing the number of trainable parameters. New architectures might be explored which better exploit the problem structure while minimising the number of trainable parameters.
- Generate training instances which vary in more than just the grid price. In practice the solar supply and base load would also be forecasted making them natural candidates as inputs.
- More work could be done to better understand the surrogate Jacobian of the one-hot encoding function. It may be that different values of k or different σ_f functions provide stronger results.
- Working with one-hot encodings can be difficult for some constraints, learning activity start times directly may make it easier to encourage satisfaction of those constraints and provide an easier learning problem.

More generally applying the LDF to a broader range of IP problems may . The specific nature of any given optimiation problem can lead to unique problems in decoding outputs, evaluating constraints during training and reconstructing feasible solutions from model outputs. Novel applications could also be explored, either in IP problems of practical interest or as a fast heuristic embedded in an exact solver.

There are also more fundamental questions on the learnability of discrete optimisation problems. Some work has been done in this vein [13, 23, 24]. This represents an interesting research direction with potential implications for the limits on what can be achieved by models which approximate solutions to discrete optimisation problems.

References

[1] M. Abolghasemi, B. Abbasi, T. Babaei, and Z. HosseiniFard, "How to effectively use machine learning models to predict the solutions for optimization problems: lessons from loss function," *arXiv preprint arXiv:2105.06618*, 2021.

[2] M. Abolghasemi, B. Abbasi, and Z. HosseiniFard, "Machine learning for satisficing operational decision making: A case study in blood supply chain," *International Journal of Forecasting*, 2023.

[3] M. Abolghasemi and R. Esmaeilbeigi, "State-of-the-art predictive and prescriptive analytics for ieee cis 3rd technical challenge," *arXiv preprint arXiv:2112.03595*, 2021.

[4] J. Hopfield and D. Tank, ""neural" computation of decisions in optimization problems," *Biological cybernetics*, vol. 52, no. 3, pp. 141–152, 1985.

[5] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: A methodological tour d'horizon," *European Journal of Operational Research*, vol. 290, no. 2, pp. 405–421, 2021.

[6] M. Abolghasemi, "The intersection of machine learning with forecasting and optimisation: theory and applications," pp. 313–339, 2023.

[7] J. Kotary, F. Fioretto, P. van Hentenryck, and B. Wilder, "End-to-end constrained optimization learning: A survey," in *IJCAI International Joint Conference on Artificial Intelligence*, pp. 4475–4482, 2021.

[8] F. Detassis, M. Lombardi, and M. Milano, "Teaching the old dog new tricks: Supervised learning with constraints," *Proceedings of the ... AAAI Conference on Artificial Intelligence*, vol. 35, no. 5, pp. 3742–3749, 2021.

[9] P. Donti, D. Rolnick, and J. Z. Kolter, "Dc3: A learning method for optimization with hard constraints," in *International Conference on Learning Representations*, 2021.

[10] D. Fontaine, LaurentMichel, and P. Van Hentenryck, "Constraint-based lagrangian relaxation," in *Principles and Practice of Constraint Programming* (B. O'Sullivan, ed.), (Cham), pp. 324–339, Springer International Publishing, 2014.

[11] F. Fioretto, T. W. Mak, and P. Van Hentenryck, "Predicting ac optimal power flows: Combining deep learning and lagrangian dual methods," *Proceedings of the ... AAAI Conference on Artificial Intelligence*, vol. 34, no. 1, pp. 630–637, 2020.

[12] F. Fioretto, P. Van Hentenryck, T. W. K. Mak, C. Tran, F. Baldo, and M. Lombardi, "Lagrangian duality for constrained deep learning," in *Machine Learning and Knowledge Discovery in Databases. Applied Data Science and Demo Track* (Y. Dong, G. Ifrim, D. Mladenić, C. Saunders, and S. Van Hoecke, eds.), (Cham), pp. 118–135, Springer International Publishing, 2021.

[13] J. Kotary, F. Fioretto, and P. V. Hentenryck, "Fast approximations for job shop scheduling: A lagrangian dual deep learning method," *Proceedings of the ... AAAI Conference on Artificial Intelligence*, vol. 36, no. 7, pp. 7239–7246, 2022.

[14] 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.

[15] C. Hertrich and M. Skutella, "Provably good solutions to the knapsack problem via neural networks of bounded size," *INFORMS journal on computing*, 2023.

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

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

[18] 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.

[19] 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, M. Luis, 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," *arXiv.org*, 2022.

[20] R. J. Hyndman and G. Athanasopoulos, *Forecasting : principles and practice / Rob J. Hyndman and George Athanasopoulos*. Melbourne: OTexts, third print edition ed., 2021.

[21] S. Seabold and J. Perktold, "statsmodels: Econometric and statistical modeling with python," in *9th Python in Science Conference*, 2010.

[22] Z. Wang, W. Yan, and T. Oates, "Time series classification from scratch with deep neural networks: A strong baseline," in *2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 1578–1585, IEEE, 2017.

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

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

A One-hot Encoding Surrogate Jacobian Implementation

A.1 Reference Implementation

A reference implementation is provided since the vmap implementation does not allow for conditional statements or indexing on the maximum index which makes it unintuitive.

```
def sigmoid_grad(diffs,k):
    # Diffs should equal x - x_max
    return k*torch.exp(-k*(diffs))/(torch.exp(-k*(diffs))+1)**2

class Onehot1d_REFERENCE(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        # Assume input is of size (num_classes)
        ctx.save_for_backward(input)
        n_classes = input.shape[0]
        top2 = torch.topk(input, 2, dim=0)
        ctx.max_val, ctx.max_idx = top2[0][0], top2[1][0]
        ctx.penul_val = top2[0][1]
        return torch.nn.functional.one_hot(ctx.max_idx,n_classes).float()

    @staticmethod
    def backward(ctx, grad_output):
        n_classes = grad_output.shape[0]

        (input,) = ctx.saved_tensors
        grad_input = grad_output.clone()
        max_val = ctx.max_val
        max_idx = ctx.max_idx
        penul_val = ctx.penul_val
        grad = torch.zeros_like(grad_output)

        # The j^th element of diffs should be z_j - z_max, exceptt for when j=z_max in which
                                                    case it is z_pen - z_max
        diffs = input - max_val
        diffs[max_idx] = penul_val - max_val

        # j^th element of sigs should be sig(z_j,z_max), except for when j=z_max in which
                                                  case it is sig(z_pen,z_max)
        sigs = sigmoid_grad(diffs,20)

        grad = torch.zeros_like(grad_output)

        #
        mod_diff = -1*torch.ones_like(diffs)
        mod_diff[max_idx] = 1

        for idx in range(n_classes):
            if idx == max_idx:
                grad[idx] = torch.dot(mod_diff * sigs,grad_output)
            else:
                grad[idx] = sigs[idx]*(grad_output[idx] - grad_output[max_idx])
        return grad, None
```

A.2 vmap Implementation

```
def sigmoid_grad(diffs,k):
    # Diffs should equal x - x_max
    return k*torch.exp(-k*(diffs))/(torch.exp(-k*(diffs))+1)**2

class Onehot1d(torch.autograd.Function):
    generate_vmap_rule = True
    @staticmethod
    def forward(input):
        # Input is of size (num_classes)
        n_classes = input.shape[0]
        top2 = torch.topk(input, 2, dim=0)
        max_val, max_idx = top2[0][0], top2[1][0]
        penul_val = top2[0][1]
        onehot_out = torch.nn.functional.one_hot(max_idx,n_classes)

        diffs = (input - max_val) + onehot_out * (penul_val - max_val)
        sigs = sigmoid_grad(diffs,20)

        return onehot_out.float(), max_idx, sigs

    @staticmethod
    def setup_context(ctx,input,outputs):
        onehot, max_idx, diffs = outputs
        ctx.mark_non_differentiable(max_idx,diffs)
        ctx.save_for_backward(onehot, max_idx, diffs)

    @staticmethod
    def backward(ctx, grad_output, _0, _1):
        # vmap function does not allow indexing on tensors
        # the onehot encoding is used as a proxy for normal indexing
        n_classes = grad_output.shape[0]
        onehot, max_idx, diffs = ctx.saved_tensors

        grad = torch.zeros_like(grad_output)
        mod_diff = -1*torch.ones_like(diffs) + 2*onehot
        # vmap does not allow conditional statements, strange expression below is implicitly
                                                    allowing us to check if idx ==
                                                   max_idx
        for idx in range(n_classes):
            grad[idx] = (onehot[idx] * torch.dot(mod_diff*diffs,grad_output) +
                         (torch.ones_like(onehot)-onehot)[idx] * diffs[idx]*(grad_output[idx
                                                                    ] - torch.sum(
                                                                    grad_output*onehot)))
        return grad

# Vectorise onehot1d over three batch dimensions
MyOnehot = torch.vmap(torch.vmap(torch.vmap(MyOnehot1d.apply)))
```