An Investigation into Predict-and-Optimize Machine Learning

Thomas Puppels

$$
\begin{aligned}
\max_{x} \quad & \begin{bmatrix} 1 & 4 & 3 \end{bmatrix} x \\
\text{s.t.} \quad & \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} x \leq 1, \\
& x \in \{0,1\}^3
\end{aligned}
$$

Faculty of Electrical Engineering, Mathematics, & Computer Science
Department of Software Technology
Algorithmics group

An Investigation into Predict-and-Optimize Machine Learning

Thomas Puppels

Supervisor, Thesis Committee Member Dr. N. Yorke-Smith
Department of Software Technology
Delft University of Technology

Thesis Committee Member Dr. F.A. Oliehoek
Department of Intelligent Systems
Delft University of Technology

Thesis Committee Member Dr. S.E. Verwer
Department of Cybersecurity
Delft University of Technology

September 2020

Thomas Puppels
An Investigation into Predict-and-Optimize Machine Learning
September 2020
Thesis Committee:
Supervisor: Dr. N. Yorke-Smith
Dr. F.A. Oliehoek and Dr. S.E. Verwer
Delft University of Technology
Algorithmics group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics, & Computer Science

Abstract

Predict-and-Optimize (PnO) is a relatively new machine learning paradigm that has attracted recent interest; it concerns the prediction of parameters that are used in an optimization problem. In return, the optimizer makes an optimal decision based on those predicted values. Standard machine learning algorithms use loss functions such as mean squared error and cross-entropy as measures of accuracy. However, the predictions made by estimators trained via such loss functions may not necessarily cause the optimizer to make good decisions. A number of approaches have been suggested over the past few years on how to most effectively tackle the PnO-setting, such as Smart "Predict, then Optimize" (SPO) and the Quadratic Programming Task Loss (QPTL). We investigate an experiment of the paper that introduced the latter, and find that an estimator used as baseline approach was set back by two factors: a class imbalance, and a training duration that was too short. However, QPTL still outperforms the base-line approach. We consider the use of the Gumbel-Softmax Straight-Through Estimator for SPO and QPTL when training neural networks on a multi-class classification dataset (MNIST) in a PnO-setting. We compare the results for SPO and QPTL for different output activation functions (linear output, sigmoid output, gumbel-softmax output) when predicting the objective parameters in 0-1 unweighted Knapsack problems and Bipartite Matching problems using this dataset. We find that neural networks trained via SPO with a linear output tend to show best performance, and that neural networks trained via QPTL are relatively unaffected by the output activation function of choice. Finally we find that PnO approaches, SPO in particular, can see large performance increases by constructing a large number of optimisation problems from a small pool of training data.

Acknowledgement

I would like to thank a number of people that aided in the creation of this thesis. First is my supervisor Dr. Neil Yorke-Smith, who was always readily available to offer advice and input. I do not think this thesis would be here without his support. Then there are a number of other scholars I would like to thank; Dr. Bryan Wilder, for his responses to some of the questions I sent him regarding Wilder et al. [46]; Dr. Emir Demirović, who was also happy to help with some questions regarding Demirović et al. [15], and was even willing to meet with me in-person during his visit to TU Delft (where he became an Assistant Professor); Dr. Willem Waegeman, an authority on Multi-Target prediction who was very willing to correspond on the subject and its applicability to Predict-and-Optimize. Lastly, I would like to thank my parents, Gerwin Puppels and Katarina Radošević, for displaying endless patience and support over the duration of this thesis.

Contents

Introduction 1

In recent years, more and more interest and research effort has been diverted to the intersection of Machine Learning (ML) and (Combinatorial) Optimization (CO). There is a particularly large amount of directions one can take: constraint learning, where constraints that make up the combinatorial problem are learnt and/or updated [8, 10], either through active learning [9] or passive learning [7]; the use of ML-based heuristics during solving of the combinatorial problem [1, 28]; and end-to-end learning of (potentially optimal) solutions to combinatorial problems [42, 5]. An excellent survey further describing such directions can be found in Bengio et al. [6].

This thesis is concerned with a direction that has only relatively recently found more interest [46, 19, 21, 36, 13]; namely that of the prediction of objective parameters present within combinatorial optimization problems (hereby labeled as "Predict-and-Optimize" (PnO) [19, 36]), after which those problems are solved to obtain solutions. The goal is to obtain solutions that are good given the *true* objective parameters. Approaches specifically adapted for the PnO setting (from here on loosely referred to as PnO approaches) have shown that they can make predictions of those parameters that might not be very accurate according to conventional performance measures, but can still induce good solutions [46, 19]. Such approaches generally train by using an existing dataset with samples and associated labels, and then use the predicted labels of those samples as the objective parameters of a certain type of optimization problem, after which they solve that optimization problem to obtain gradient information. Here, we call such samples *item-samples*, and an optimisation problem of which the objective function is parameterised by the labels of a group of such item samples will be called a *problem-sample*. When the labels of a group of samples are used like that, we call that group of samples a *problem-sample* - an individual sample is called an *item-sample*.

In this thesis, we investigate various different topics with respect to PnO, that do not directly relate to each other. As such, there is no one red line present through the entirety of the thesis, and the following research questions are mostly contained in their own chapters.

1.1 Research Questions

At the time of starting this thesis project, relatively little research was present on this topic. We started with a short investigation and repetition of the QPTL-framework introduced in Wilder et al. [46]. Here we investigate the results obtained in that paper and point at some factors that influenced performance of conventional approaches. Afterwards, we explore the PnO-setting with a multi-class classification dataset, which to our knowledge has not yet been done before, and attempt to adapt existing PnO approaches for this type of dataset. We also investigate if this is beneficial in the first place. Finally, we evaluate the performance of PnO-approaches when we perform problem-sample resampling from a limited pool of data. As such, we ask the following research questions:

RQ1: *What are the effects of class imbalance on the QPTL-framework when compared to conventional machine learning approaches?*

RQ2: *How does pre-training affect QPTL performance?*

RQ3: *How should one tackle multi-class classification datasets using PnO approaches, when using neural networks?*

RQ4: *How are PnO approaches affected when the objective parameters between problem-samples are identical versus when they are randomly selected?*

RQ5: *How does problem-sample resampling affect the performance of PnO approaches? Does the use of a Multiple-Input Multiple-Output (MIMO) network further improve performance?*

The research questions **RQ1**, **RQ2** are contained in Chapter 3, the research questions **RQ3** and **RQ4** in Chapter 4, and the research question **RQ5** in Chapter 5.

Preliminaries and Related Work 2

2.1 Combinatorial Optimisation

In combinatorial optimisation, optimal solutions to an optimisation problem have to be found from a finite set of (feasible) solutions. Such a solution x_{sol} generally consists of a selection of objects with reward r that fulfill all requirements of the problem, and has a certain objective value $O(r, x_{sol})$. The objective function $O(r, x)$ is the function to minimize or optimize, with r a set parameter. When O is maximal (or minimal, depending on the problem), the associated solution is optimal. Typically the solution space of a combinatorial optimisation problem grows exponentially with the number of objects present, making it not possible in practice to determine all possible (feasible) combinations x of objects and their associated objective value $O(r, x)$. In that case, the combinatorial problem belongs to the complexity-class NP. Combinatorial problems are frequently described using integer programs, often with binary variables.

2.1.1 Integer Programming

An Integer Program (IP) is a representation of an optimisation problem that involves only integers. If the IP only features linear constraints, then it is an Integer Linear Program (ILP). An ILP can be written as follows [27]:

$$\begin{aligned} \max_{x} \quad & r^T x \\ \text{s.t.} \quad & Ax = b, \\ & x \geq 0, \\ & x \in \mathbb{Z}^n \end{aligned} \tag{2.1}$$

with r, x n-dimensional vectors, b an m-dimensional vector, and A an $m \times n$ matrix. We refer to it as P. The objective function of P is the function $O(r, x) = r^T x$. $O(r, x)$ is the value that we aim to maximize as much as possible. We label r the objective parameters. If r is known, we can formulate $OPT(r)$ as the optimal objective value given r and $OPT^*(r)$ as the optimal solution x_{opt} induced by r after solving the problem. That is, $OPT(r) = O(r, OPT^*(r))$.

If we drop the integrality constraint of P, it means that any optimal solution x_{frac} can now have fractional values. This fractional solution obtains a solution value $O_{frac} = O(r, x_{frac})$. Likewise, the optimal solution x_{int} to the original problem *with* the integrality constraint has an objective value $O_{int} = O(r, x_{int})$. In a maximisation problem, the *integrality gap* is then equal to $I_{gap} = \frac{O_{frac}}{O_{int}}$. It indicates how much *better* the optimal fractional solution is than the optimal integer solution. When the (square, integer) constraint matrix A is *totally unimodular* and r is integral, the integrality gap is equal to one[27]. When this occurs, the solution to the optimisation problem without the integrality constraint is also integral[27].

Definition 1. *Unimodularity*
"A square, integer matrix B is called unimodular (UM) if its determinant det $(B) = \pm 1$." [27]

Definition 2. *Total Unimodularity*
"An integer matrix A is called totally unimodular (TUM) if every square, nonsingular submatrix of A is UM." [27]

In fact, the objective value of this integral solution must be of the same quality as the one returned by the original ILP. This is a rather desired property when it comes to solving ILPs, as it suffices to simply solve the relaxation of the ILP (Equation 2.1 but without the integrality constraint), which can be done in polynomial time. Examples of well-known problems with a TU constraint matrix are the assignment problem and the bipartite matching problem.
Once the ILP has been formulated, it can be solved using dedicated solvers like CPLEX or GUROBI.

2.1.2 Knapsack

The knapsack (KP) problem has many variants; in this thesis we concern ourselves primarily with the unweighted KP problem. The KP problem describes the problem of wanting to bring the most valuable items with limited capacity. In the unweighted knapsack problem, all items have the same weight, and the problem is solvable to optimality in polynomial time, due to the relaxation having an integral solution. When weight between items differs, however, the problem turns into an $\mathcal{NP}$-hard problem and the solution of the relaxation is not necessarily integral. In this thesis, we specifically focus on the unweighted 0-1 KP problem, which means each item can only be chosen once at most.
Both the unweighted and weighted 0-1 knapsack problem with n items can be formulated as an ILP as follows:

$$\begin{aligned} \max_{x} \quad & r^T x \\ \text{s.t.} \quad & Wx \leq c, \\ & \quad x \in \{0,1\}^n \end{aligned} \tag{2.2}$$

with W a $1 \times n$ matrix containing the weights of each item, c the capacity of the knapsack, and r a $1 \times n$ matrix with the rewards for each item.

2.1.3 Maximum Bipartite Matching

Maximum Bipartite Matching (simply Bipartite Matching (BM) from here on), involves the presence of two sets of nodes (S_1, S_2 with $|S_1| = n, |S_2| = m$) and a number $0 < k \leq n \cdot m$ of available edges between the two sets of nodes (a bipartite graph). The goal is to pick the edges with the highest value, without any two edges being connected to the same node. That is, it should be impossible to move from a node $n_1 \in S_1$ to a node $n_2 \in S_2$ and then move back to a node $n_3 \in S_1, n_3 \neq n_1$. The solution value is simply the sum of the value of all edges that are picked - if all edges have the same weight, then the optimal solution is the one that is able to choose the most edges.

2.2 Machine Learning

Machine learning (ML) is a field of study that involves the automatic learning of patterns underlying given data, such that good predictions can be made on novel data using an ML model f_ω, with ω the parameters of the ML model. The most general example of (supervised) ML is where a large volume of data S is available, consisting of *samples* s each with their own *label* r. The sample is usually represented by a row of data (which consist of elements that can be binary, numerical, categorical and more) called the feature-vector, and when we are talking about a sample, we are generally talking about the feature-vector that represents that sample. If the labels are numeric, we speak of regression; if instead these labels belong to one of a select number of groups, we speak of classification. Binary classification, for example, involves predicting a label with only two possible options. The samples are subdivided into three datasets: the training set S_{trn}, the validation set S_{val}, and the test set S_{test}. The machine learning model is allowed to use the samples in S_{trn} for *training*. Here, the model f_ω is allowed to alter ω such that a loss function $\mathcal{L}(f_\omega(s), r)$ is minimized, with $f_\omega(s)$ the predicted label of sample s with label r. Typically in regression problems the Mean Squared Error (MSE) is used (see Equation 2.3), whereas in classification problems the cross-entropy loss-function is used. An algorithm like gradient descent is used to update ω such that $\mathcal{L}$ returns a lower value across the training set.

$$\mathcal{L}_{MSE}(r, f_\omega(s)) = \frac{1}{|S_{trn}|} \sum_{i=1}^{|S_{trn}|} (f_\omega(s_i) - r_i)^2 \tag{2.3}$$

Once all samples in S_{trn} have been evaluated and weight updates have taken place, a single *epoch* has been completed. One can set a pre-defined number of epochs or use the validation set to determine when to stop. The validation set is used to prevent overfitting, where the ML model f_ω learns patterns that are specific only to the training set, to improve generalization to other datasets. Finally, after training is complete, the performance of f_ω on the test set is evaluated, typically also using the loss function $\mathcal{L}$ but often-times others are used in addition (for example, the accuracy of an ML model might be displayed alongside the obtained cross-entropy on test set).

2.3 Predict-and-Optimize

As mentioned in Section 2.2, ML models are trained and evaluated by common loss functions such as MSE. The ML model learns to make predictions in such a way that the MSE is minimized. For difficult problems (the underlying patterns are very complex, the features are very noisy which obfuscate the underlying pattern, the number of samples is very low, etc.), it is difficult for ML models to predict perfectly. Then, the way in which they make mistakes (i.e., which samples end up being labeled wrongly or the degree to which they are labeled wrongly) is highly dependent on the loss function used during training. Wilder et al. [46] explores this in-depth, but here we give a simple example (the general idea of which comes from "Example 1." from Demirović et al. [15]).
Consider the unweighted knapsack problem with capacity $c = 1$ and two items: item s_1 has value $r_1 = 6$, and item s_2 has value $r_2 = 5$. Assume we have two ML

models f_1, f_2: f_1 makes the predictions $f_1(s_1) = 5, f_1(r_2) = 6$ and f_2 makes the predictions $f_1(s_1) = 8, f_1(r_2) = 1$. If we would determine the better estimator by using the MSE loss function, we would find that that f_1 is preferred:

$$\mathcal{L}_{MSE}(r, f_1(s)) = \frac{1}{2}((f_1(s_1) - r_1)^2 + (f_1(s_2) - r_2)^2) = \frac{1}{2}((5-6)^2 + (6-5)^2) = 1$$

$$\mathcal{L}_{MSE}(r, f_2(s)) = \frac{1}{2}((f_2(s_1) - r_1)^2 + (f_2(s_2) - r_2)^2) = \frac{1}{2}((8-6)^2 + (1-5)^2) = 10$$

However, in the KP setting the predictions made by f_1 would cause the solver to choose item a_1 (as $f_1(s_1) > f_1(s_2)$) which has lower value than item s_2, whereas the predictions made by f_2 lead to choosing item s_2 (as $f_2(s_1) < f_2(s_2)$). As such, if we want our estimator to induce good decisions in the KP problem, f_2 is the better estimator.

This illustrates the fact that conventional ML loss functions might not be good indicators of performance for ML models when it comes to the final task that they are employed in. [15] That is, they do not necessarily minimize the *"task loss"* [16], which in the context of ILPs is the objective function. It has led to the development of the field which has been called "Predict-and-Optimize"[19, 36] (PnO). It recognizes the fact that predictions will nearly always lead to errors and that such errors should be handled in a task-specific manner. In the case of PnO, the task at hand is generally the optimal solving of an optimization problem.

Formally speaking, and referring back to Section 2.2, we have a dataset S consisting of samples s with labels r. However, now those samples are used to construct combinatorial optimisation problems. To simplify discussion in the future, we refer to a sample s_i with an associated label r_i as an *item-sample* (as in, it is a typical ML sample that is also an *item* in the knapsack problem).

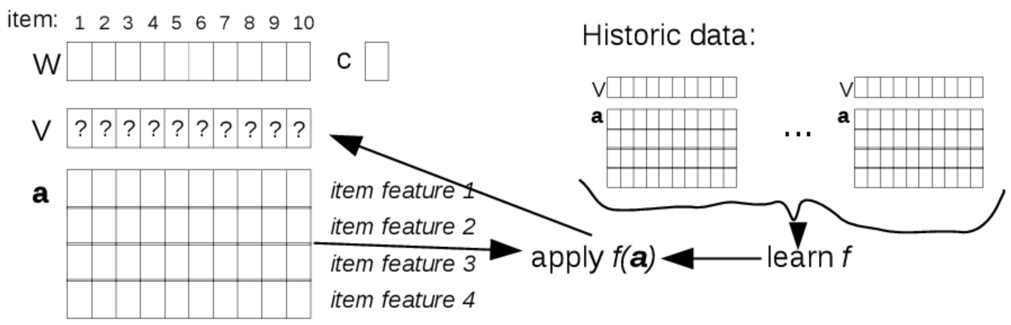


Figure 2.1: Diagram that illustrates a PnO setting with the knapsack problem. Note that this picture uses a, v to refer to a sample and its label, rather than s, r. On the right we see prior available data, with samples $\mathbf{a}$ that are 4-dimensional feature-vectors (the columns) and with known value v. Such a sample is an *item-sample* and by itself is nothing more than a typical machine learning sample. These prior samples are used to train an ML-estimator f. On the left we see new data with unknown labels constructed in the form of a knapsack problem, with weights W, capacity c, and unknown value v. Such an optimisation problem of which the objective function is determined by its item-samples we call a *problem-sample*. Performance of f is measured by the quality of the decision its predictions induce - choosing more valuable items is better. Diagram taken from Demirović et al. [13].

When we say that we construct combinatorial optimisation problems using item-samples, we mean that we use the labels of those samples as the objective parameters of the optimisation problem. That is, the vector r in Equation 2.1 consists of all values belonging to a given number of samples s. This is made more clear in Figure 2.1. If we have one such optimisation problem constructed using known objective parameters r we can compute the regret R incurred by solving that optimisation problem with the predicted labels $f_\omega(s) = \hat{r}$ instead as follows

$$R = OPT(r) - r \cdot OPT^*(\hat{r}) \tag{2.4}$$

The regret tells us on how much potential value we missed out. Note that this equation has been labeled the SPO-loss by Elmachtoub and Grigas [19] - they seem to be the first to use this particular function as performance indicator in a PnO setting. It is the value associated with the optimal decision minus the value associated with the decision made based on predicted values. The regret is frequently used as performance indicator in PnO literature. The following sections will detail the main approaches to PnO that have been suggested in the literature.

2.3.1 Overview of various approaches to PnO

In this thesis, we primarily restrict ourselves to two main general approaches that have been suggested in the literature for this particular problem-setting:

- "Decision-focused Learning" from [46], which has been dubbed in following papers [13, 36] as "Quadratic-Programming Task Loss (QPTL)", which we do as well.
- Smart "Predict, then Optimise" (SPO) from [19]

These approaches have received the most attention, which is why we do so. Demirović et al. [13] defines these approaches as *direct* approaches, as the listed approaches directly incorporate the optimisation problem as loss function. We will frequently refer to such approach as "PnO approaches" throughout the thesis. Demirović et al. [13] also defines *indirect* approaches as: "Indirect methods use a standard learning method and loss function that is independent of the optimisation problem". When we refer to "conventional approaches", "standard approaches" and "traditional approaches", such methods is what we mean. Training a neural network on a dataset using the MSE as loss function would be a standard approach.

Quadratic Programming Task Loss (QPTL)

QPTL, introduced in Wilder et al. [46], is a gradient-based method that is similar to the method used in [16], but applies it to combinatorial problems. It uses the quadratic relaxation of the combinatorial problem to differentiate the optimal solution of that problem with respect to the predicted objective function values. The relaxation is necessary, because the solution value associated with a discrete decision is not differentiable with respect to the machine learning parameters ω. Considering a number of item-samples s of which the labels r are used as objective parameter in a problem-sample, Wilder et al. [46] writes the gradient of the objective value versus the ML model parameters as:

$$\frac{dr \cdot OPT_R^*(f_\omega(s))}{d\omega} = \frac{dr \cdot OPT_R^*(f_\omega(s))}{dOPT_R^*(f_\omega(s))} \frac{dOPT_R^*(f_\omega(s))}{df_\omega(s)} \frac{df_\omega(s)}{d\omega} \tag{2.5}$$

using the chain rule (adapted from the first equation listed in the "General Framework" section of Wilder et al. [46]). To obtain the second term, Wilder et al. [46]

relaxes the combinatorial optimisation problem OPT to OPT_R, and then differentiate the Karush-Kuhn-Tucker conditions. Wilder et al. [46] obtains the second gradient, which is the gradient of the optimal solution induced by the predictions versus the predictions made by f_ω, by solving the following equation system:

$$\begin{bmatrix} \nabla_x^2 OPT_R(f_\omega(s)) & A^T \\ diag(\lambda)A & diag(Ax-b) \end{bmatrix} \begin{bmatrix} \frac{dx}{df_\omega(s)} \\ \frac{d\lambda}{df_\omega(s)} \end{bmatrix} = \begin{bmatrix} \frac{d\nabla_x O(x, f_\omega(s))}{df_\omega(s)} \\ 0 \end{bmatrix} \tag{2.6}$$

with $x(= OPT_R^*(f_\omega(s))), \lambda$ being primal and dual solutions to OPT_R that satisfy the KKT conditions (and are therefore optimal solutions).

With respect to linear programming, they restrict themselves to (I)LPs for which the relaxation has an integer optimal solution (primarily those with a totally unimodular constraint matrix). The reason for doing so is likely because QPTL uses a relaxation of the original problem. If the optima between the relaxation and the original problem would differ, gradient descent would cause the machine learning model to learn predictions that would be good only in the relaxed setting, but not in the integral setting. Wilder et al. [46] add a weighted quadratic term to the original problem objective to ensure that $\nabla_x^2 OPT_R(f_\omega(s))$ reduces to γI rather than 0, which would prohibit solving the system of equations. That is, the ILP in Equation 2.1 is transformed to a Quadratic Program:

$$\begin{aligned} \max_x \quad & r^T x - \gamma \|x\|_2^2 \\ \text{s.t.} \quad & Ax = b, \\ & x \geq 0 \end{aligned} \tag{2.7}$$

Over the course of training, the quadratic term γ is reduced to make the quadratic problem more and more similar to the original problem. During testing, it is set to 0.

Wilder et al. [46] analyzes the performance of their method in three different problems, two of which take place in the submodular maximisation domain, which is outside of the scope of this thesis. The remaining experiment is a Bipartite Matching experiment. QPTL performs nearly 70% better than the next best method (that is, it finds 70% more matches). In a related paper ([29]), QPTL is applied in a real-world setting (prescribing interventions for Tuberculosis patients), obtaining a 15% performance over a network trained via a conventional loss function.

In Ferber et al. [21], the requirement that the relaxation of the combinatorial problem has to have an integral optimal solution is lifted. This is done by crafting a surrogate S for the relaxation of the original combinatorial problem, with an integral solution. The relaxation of this problem is repeatedly solved; each time a non-integral solution is found, a new constraint is added to the problem that excludes the found continuous solution without excluding feasible integral solutions. When the returned solution is integral, S is complete. Afterwards, the standard QPTL approach can be applied. The whole procedure is called "MIPaaL" [21]. Note that the crafting of the surrogate can require an exponential number of cuts.

Smart "Predict, then Optimise" (SPO)

In Elmachtoub and Grigas [19], the Smart "Predict, then Optimise" (SPO) framework is introduced. The core idea of the framework is the introduction of the SPO loss (which is simply the regret as in Equation 2.4) to use for training estimators. The framework applies to convex optimization problems, both combinatorial and mixed-integer.[19] Because differentiation of the regret is difficult due to non-convexity and discontinuity, Elmachtoub and Grigas [19] develops the SPO+ loss as a surrogate loss function. The SPO+-loss is written as follows [19] for a group of item-samples s with labels r, with $\hat{r} = f_\omega(s)$:

$$r \cdot OPT^*(-(r - 2\hat{r})) + 2\hat{r} \cdot OPT^*(r) - OPT(r) \tag{2.8}$$

The SPO+ loss is greater or equal than the SPO loss and convex. In addition, Elmachtoub and Grigas [19] also show that

$$(2)(OPT^*(r) - OPT^*(2\hat{r} - r)) \tag{2.9}$$

is a *subgradient* of the SPO+ loss. The subgradient can be used to perform backpropagation, as seen in Algorithm 1 (taken from Algorithm 2 (with minor alterations) of [36], idea suggested by [19]).

Algorithm 1 Gradient descent step in the SPO+ approach, using a single problem-sample, containing item-samples s with labels r, estimator f with weights ω and learning rate α.

```
1: $\hat{r} \leftarrow f_\omega(s)$
2: $x_{spo} \leftarrow OPT^*(2\hat{r} - r)$
3: $x_{opt} \leftarrow OPT^*(r)$
4: $\nabla\mathcal{G} \leftarrow x_{opt} - x_{spo}$
5: $\omega \leftarrow \omega - \alpha \cdot \nabla\mathcal{G} \cdot \frac{d\hat{r}}{d\omega}$
```

As pointed out in [36], the subgradient $\nabla\mathcal{G}$ is simply the difference between the optimal solution given true labels x_{opt} and x_{spo}. If a particular item-sample is chosen in the optimal solution but not in the other, that part of the gradient is 1, which implies that f_ω should predict a greater value for that sample.
Elmachtoub and Grigas [19] show that linear machine learning models trained using the SPO+ loss strongly outperform those that are not, when the target to be predicted becomes more and more non-linear with respect to the features. Mandi et al. [36] continue this line of work by evaluating how to adapt it for hard combinatorial problems. Mandi et al. [36] recognizes that one of the main difficulties in the PnO setting is that all thus-far proposed methods generally require the solving of optimisation problems during training of the machine learning model. This makes it infeasible to train machine learning models by gradient descent in the PnO setting when the optimization problems can take exponential time to solve. Mandi et al. [36] investigates the use of approximation algorithms instead of exact solvers to solve the optimisation problems during backpropagation (which they call SPO-relax). Whereas Ferber et al. [21] transforms the originally $\mathcal{NP}$-hard problem into an optimisation problem solvable in polynomial time, Mandi et al. [36] use an approximation algorithm that solves the $\mathcal{NP}$-hard combinatorial problem in polynomial time, which returns a decision that is within some approximation bound of the optimal decision. Mandi et al. [36] also investigates warm-starting of the learning, by first training the PnO model with a conventional loss function, after

which the SPO+ loss is utilized. They also investigate warm-starting of the solving. Minimal benefits were observed from warm-starting the learning. Finally, they compare their method to the QPTL approach from [46]. In their experiments, SPO+ either outperforms or performs equal to QPTL, while being faster to train.
More recently, a follow-up paper to [19] was released as [20]; which extends the SPO method to decision trees. In fact, they manage to train decision trees using the SPO-loss (thought to be intractable), rather than the surrogate SPO+ loss. They do so by exploiting properties unique to decision trees. The SPOTrees find superior decisions, and require lower complexity for good performance than their non-SPO equivalents.
In the future, when we may refer to the "SPO approach", we mean the SPO+ approach.

Others

Some other papers cannot easily be classified in one of these broader approaches. Demirović et al. [13] explores the applicability of existing PnO methods (namely, SPO and QPTL) to the (un)weighted knapsack problem in comparison to conventional machine learning approaches, and finds that existing PnO frameworks primarily performed well in optimisation problems that are convex or close to convex. Demirović et al. [15] introduces a framework wherein the optimal parameters for the machine learning algorithm can be found given a ranking optimisation problem and lists many notable properties of PnO problems. Demirović et al. [14] explores the solution structure for the optimisation problems in PnO methods, and performs machine learning using coordinate descent and dynamic programming over piecewise linear functions. More specifically, a PnO method is provided that applies to any combinatorial problem that can be solved using dynamic programming. They replace the objective vector of a CO problem by a vector consisting of parameterised linear equations and analyze how the solution of the optimisation problem changes with the input parameter for the linear equations - points at which the solution changes are labeled *transition points*.

2.4 Other work on combining Machine Learning and Combinatorial optimisation

The PnO setting is not the only direction of research that focuses on integrating machine learning in combinatorial optimisation. A large body of research exists in the direction of *constraint acquisition*, where constraints are either learnt passively from negative and positive examples [7]; via active learning [9] or through querying an user [8]. In Kolb et al. [32], implicit constraints are detected from tabular data and are used to aid in auto-completion and error-detection.
Another alternative approach is directly embedding trained ML models within a Combinatorial optimisation Problem. This can be useful when a certain phenomenon is too difficult to simulate directly, or is computationally intractable to solve. One such methodology has been labeled Empirical Model Learning (EML) [35]. An example is given within the paper of *thermal-load dispatching*. Tasks must be assigned to CPU cores so as to either prevent overheating (in one variant of the problem), or have as many cores as possible operating at high efficiency (in the

other variant). The difficulty here lies in the fact that the temperature of a core is highly dependent on the temperature of other cores in the nearby vicinity. Neural networks are trained to predict the temperature of a given core, and embedded inside the optimisation problem directly through the use of Neuron Constraints.
Rather than embedding entire ML models, *Constraint Programs* (CPs) can also be continuously updated using newly obtained data (CPs are very similar to ILPs, but its constraints are more general, and it relies on logic rather than math to find the optimal solution). An example is the Inductive Constraint Programming (ICON) loop paradigm [37, 24, 23]. Here, data is obtained over time and analyzed to revise and/or update constraints and optimisation criteria. The loop consists of a CP component, an ML component, and a World component. The ML component gets updated using data provided by the World component and receives feedback from the CP component with regards to whether there are any feasible solutions. Predictions are made by the ML component; in a carpooling example, it is used predicted how likely users of the car pooling are to agree to pair up with one another. As user preference changes, so do the weights inside the CP model.
Finally, some authors have used ML methods for preference elicitation to learn weights that are used later in combinatorial optimisation problems. An example is Constructive Preference Elicitation (CPE) [17], where the goal is to produce an optimal configuration (with regards to an user's preferences) for a product from scratch. In [41], this is accomplished by suggesting the user $K \geq 2$ maximally diverse configurations with high utility, with the user replying with their most favoured one. In Coactive Learning [39] a single configuration is presented to the user, after which the user improves it slightly.

An Investigation into QPTL 3

3.1 Research Questions

In this chapter, we investigate the QPTL method proposed in [46]. We verify the results found, and address the following research questions:

RQ1: *What are the effects of class imbalance on the QPTL-framework when compared to conventional machine learning approaches?*

RQ2: *How does pre-training affect QPTL performance?*

3.2 Background

Wilder et al. [46] performs three experiments to show the applicability of their PnO method. Two of them belong to the submodular maximization domain, which is an optimisation paradigm not considered in this thesis. The remaining experiment belongs to the integer programming domain, which is our main interest. Wilder et al. [46] used the cora dataset [38],a citation dataset, to construct 27 different BM problems-samples, with feature-vectors that represent an edge between nodes as item-samples. The bipartite graphs have 50 nodes on each side. It must be predicted whether an edge (citation) exists between two nodes (papers). Wilder et al. [46] stresses that this particular machine learning problem is rather difficult, as the features are not informative enough to make accurate predictions. The problem-samples are divided into a typical train-test split of 80% and 20% respectively, meaning that the training set consists of 22 problem-samples after rounding, and the test set consists of five problem-samples. Wilder et al. [46] uses one-layer (linear) and two-layer neural networks (with a hidden layer of size 200) as base predictors, after which their performance is compared when trained conventionally and when trained via QPTL. The networks were trained using Adam [30] with a learning rate of $1e^{-3}$. The results were averaged over 30 random splits.
The code underlying the experiment can be found on `https://github.com/bwilder0/aaai_melding_code/blob/master/matching.py`. Future chapters also use some parts of this code (the definition of the Bipartite Matching constraint matrix and the computation of the QP- and LP-loss (which uses a differentiable QP solver from Amos and Kolter [2] in addition, and is thus also used in following chapters)). Neural network training was done using the Pytorch library [40], which is what will be used in future chapters as well.

3.2.1 Reproduction of Results

We began by first rerunning the BM experiment as is, with the sole exception that batches during training were made equivalent and number of training epochs. In the original code, the neural networks trained with the conventional ML loss functions used standard Stochastic Gradient Descent (SGD) [11] with randomly selected batches of batch size 100 for 2 epochs, whereas the networks trained via QPTL used batches that were equivalent to the item-samples that were present in the problem-samples and were trained for 12 epochs. For the QPTL-approach, one iteration uses all the item-samples in a given problem-sample. These "batches" stayed equal throughout all epochs, whereas the former had varying batches. In our code, all networks are trained using the item-samples present in the problem-samples during each backward-forward pass, and were trained for 12 epochs.

Table 3.1: Result comparison after repeating the bipartite matching experiment in [46]. Results are shown as 95% confidence intervals around the mean. Reported values refer to test set. '1L' refers to a one layer neural net; '2L' refers to a two-layer neural net. '-ML' refers to a conventionally trained network, '-QPTL' to a network trained via the QPTL approach.

Neural Net	Solution Quality (Sol)			Cross-Entropy (CE)	
	Wilder et al. [46]	Repetition	Optimal	Wilder et al. [46]	Repetition
1L-ML	2.99 ± 0.76	3.01 ± 0.26	$40.01(-0.56, 0.59)$	$\mathbf{0.696 \pm 0.001}$	$\mathbf{0.224(-0.003, 0.004)}$
2L-ML	3.49 ± 0.32	3.31 ± 0.25	$40.01(-0.56, 0.59)$	0.223 ± 0.005	0.226 ± 0.005
1L-QPTL	2.50 ± 0.56	$2.95(-0.31, 0.29)$	$40.01(-0.57, 0.59)$	$\mathbf{0.994 \pm 0.002}$	$\mathbf{1.46 \pm 0.011}$
2L-QPTL	6.15 ± 0.38	$7.40(-0.43, 0.44)$	$40.01(-0.56, 0.59)$	0.689 ± 0.004	$0.30\ (-0.021, 0.020)$

Table 3.2: Result comparison of repeating the bipartite matching experiment in [46] for AUC. Results are shown as 95% confidence intervals around the mean.

Neural Net	AUC	
	Wilder et al. [46]	Repetition
1L-ML	0.499 ± 0.013	0.496 ± 0.005
2L-ML	0.498 ± 0.007	0.495 ± 0.007
1L-QPTL	0.501 ± 0.011	0.502 ± 0.007
2L-QPTL	0.560 ± 0.006	$0.599(-0.007, 0.008)$

The comparison of the repetition of the experiment can be seen in Tables 3.1 and 3.1. All reported results are on the test set. Most values are very similar between the two, with the exception of the 2L-QPTL network; the 2L-QPTL network performs even better in our repetition of the experiments. The 95% confidence intervals do not even meet between the two results. The reason for this is that the code uploaded on `https://github.com/bwilder0/aaai_melding_code/blob/master/matching.py` has the QPTL-network use different parameters than listed in Wilder et al. [46]. Namely, it uses a hidden layer of size 500 (rather than 200) and a learning rate of $1e^{-4}$ (rather than $1e^{-3}$). It is likely that the learning rate in particular has a strong influence on the results, as there are very few batches per iteration. The results in Wilder et al. [46] are thus better than listed in the paper. Note that the values for the cross-entropy loss also vary for the single-layer networks; this is likely due to the fact that the original code had a minor error where an extra sigmoid output activation was added to the end of the neural network. It seems as if there is a strong relationship between AUC and solution quality.[21] note this as well, and find the same in their diverse bipartite matching experiment (which uses the same dataset), and state that "predictions learned by MIPaaL may sometimes also be accurate in a traditional sense", but do not explain why that may be the case.

There exists a class imbalance of roughly a 1:20 positive:negative sample ratio. We decided to see if there would be a solution quality increase for the ML-networks on par with that of NN-QPTL by resolving the class imbalance present in the training set, and if there was, evaluate how large the increase is. We noticed that the original experiments trained the neural networks for only (a pre-defined) twelve epochs, with no clear indication why this number was chosen. We decided to investigate whether training for a longer time would affect results and also show the learning curves in the process. In addition, we wondered if pre-training the QPTL-networks using a conventional ML loss function (in this case binary cross-entropy) would positively or negatively affect their performance (note that this particular experiment was independently performed from Mandi et al. [36] (who investigated this on a separate dataset) and Ferber et al. [21] (who investigated this using the same dataset and also using QPTL (in reality MIPaaL, but this can be considered a QPTL-variant), but in the context of *diverse* bipartite matching), which were not available at the time this experiment was performed).

3.3 Experiments

3.3.1 Increasing training time

With regards to increasing the training time, we simply increased the number of epochs during training to 100 epochs and recorded the learning curve on the training and test set. We adjust the quadratic relaxation parameter γ as follows; the original code used a starting value of 0.1 and multiplied it by 0.8 after every epoch. We use the same starting value; to make sure that the final value of gamma is the same as it is after training for 12 epochs, we set the gamma-multiplier to $(\frac{\gamma_{end}}{\gamma_{start}})^{1/n_{epochs}} =\sim 0.97$ for 100 epochs. The same procedure is done for following chapters, except the end start value was set to 0.005 ($0.8^{12} = 0.0069$).

3.3.2 Conventional Pre-training followed by QPTL

Another thought was to perform conventional training first, and then follow-up with the QPTL approach (pre-training). That is, we first train the QPTL-network using the cross-entropy loss for 100 epochs, and then switch to training via QPTL for 100 epochs. The idea behind this is to first let the neural network get a good sense of (conventional) accuracy first, before applying QPTL for subtle adjustments in the prediction to improve performance in terms of the solution value obtained on the problem-samples.

3.3.3 Fixing Class Imbalance

Here, we use various sampling methods to resolve the class imbalance in the underlying dataset, for the ML-method only. Note that is impossible to retain the available problem-samples when ameliorating the class imbalance; item-samples would be removed that exist in problem-samples, or additional item-samples would be introduced that do not belong to any problem-sample. We use the three sampling methods to resolve the class imbalance described in Table 3.3. [22] In all cases, we either decreased the number of samples, or increased the number of item-samples in the training set until a 1:1 ratio of positive and negative item-samples was obtained (from a roughly 1:20 ratio). Afterwards, we test on the problem-samples in the test set (for which no class imbalance has been resolved).

Table 3.3: Table describing sampling methods to combat class imbalance

Definition	Explanation
Downsampling	Adjusts the majority class so that the number of majority samples are equal to the minority class (or any other preferred ratio). Essentially, we discard certain training samples of the majority class.
Upsampling	Adjusts the minority class so that the number of minority samples are equal to the majority class. Additional samples are created by cloning existing samples.
SMOTE	Similar to upsampling, but instead of randomly copying samples, additional samples are synthetically generated by interpolating between existing samples.

3.4 Results

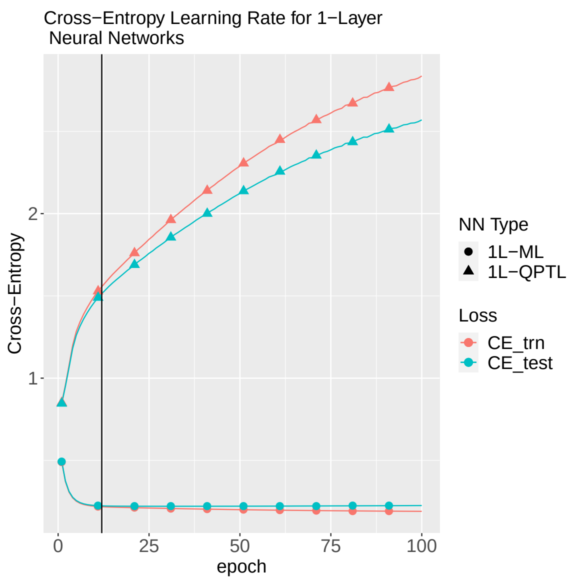
All experiments use the CPLEX 12.9 solver and are run on the TU Delft HPC cluster. This is true for all experiments in other chapters as well.

3.4.1 Bipartite Matching

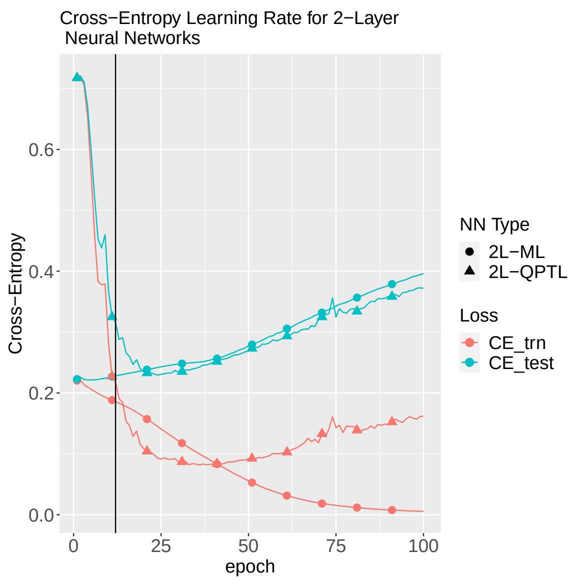
Table 3.4: Mean values with bootstrapped 95% confidence interval for Bipartite Matching problem after training for 100 epochs.

	CE (train)	**CE (test)**	**Sol (train)**	**Sol (test)**	**AUC**	**Time/Iteration**
1L-ML	0.19 (0.00)	0.23 (0.00)	2.89 (0.10)	2.76 (-0.26, 0.25)	0.49 (0.01)	0.01 (0.00)
2L-ML	0.01 (0.00)	0.40 (0.01)	40.36 (0.13)	4.79 (0.31)	0.55 (-0.00, 0.01)	0.30 (0.00)
1L-QPTL	2.84 (-0.03, 0.04)	2.57 (0.05)	15.70 (-0.19, 0.21)	3.01 (0.23)	0.50 (0.01)	3.68 (0.07)
2L-QPTL	0.16 (0.01)	0.37 (0.01)	40.29 (0.14)	6.31 (-0.37, 0.35)	0.58 (0.01)	3.91 (0.08)
1L-QPTL-pre	0.23 (0.00)	0.28 (0.01)	9.63 (-0.15, 0.14)	2.95 (0.23)	0.50 (0.00)	3.73 (0.08)
2L-QPTL-pre	0.08 (0.00)	0.64 (0.02)	40.35 (0.13)	4.91 (-0.32, 0.31)	0.54 (0.00)	3.93 (0.08)

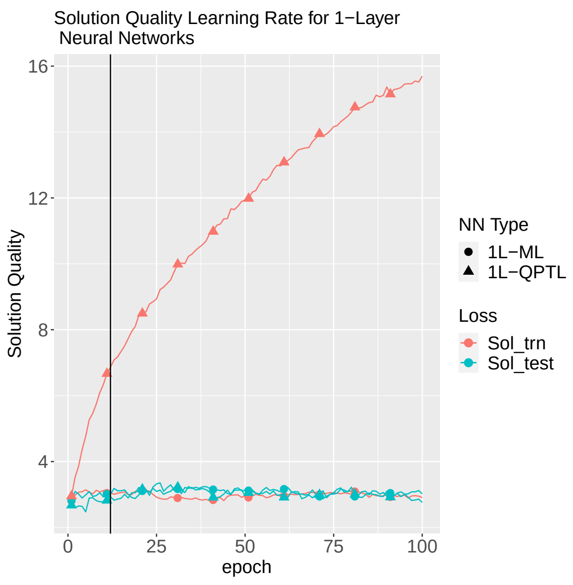
We evaluate the effects of training for a longer amount of time in the same bipartite domain as in Wilder et al. [46]. We report the results after training for 100 epochs in Table 3.4 and display the learning curves for the training and test set in Figure 3.1. Table 3.4 shows much better performance for the 2L-ML network than Table 3.1 does, increasing performance by nearly 50%. The 2L-network however, shows decreased performance, likely as a result of overfitting. All 1-layer networks are relatively unaffected by the change in training duration. The pre-trained networks can only be said to perform on par with the conventionally trained networks. This is a rather big issue, as indicated by the amount of time an iteration takes. For the single-layer QPTL-networks, training takes nearly 200 times longer than the ML-networks. This gap closes to around 20 times more time for the 2-layer networks, which is still a significant time increase.
In Figure 3.1, we analyze the learning curves on the training and test for the ML and QPTL networks. The vertical green line indicates the amount of time the networks were trained in Wilder et al. [46]. In Figures 3.1a and 3.1b the CE learning curve over time is shown, for the one-layer and two-layer networks respectively. In the former, we see that the ML-network stabilizes relatively quickly, whereas the



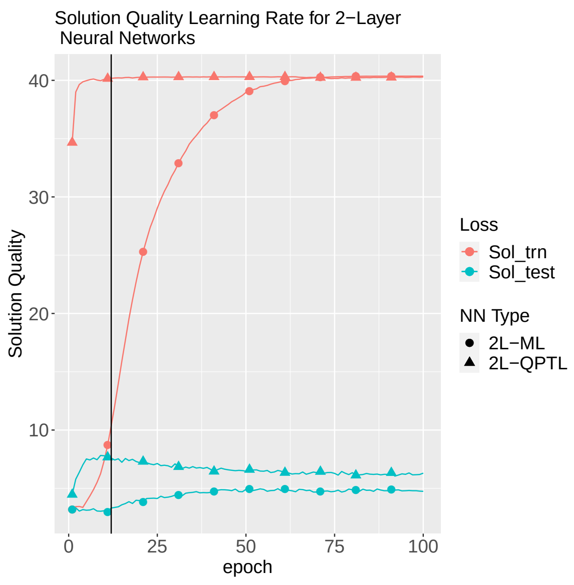
(a) Cross-entropy learning rates for 1-layer net



(b) Cross-entropy learning rate for 2-layer net



(c) Objective value learning rate for 1-layer net



(d) Objective value learning rate for 2-layer net

Figure 3.1: Mean Learning Rates for Cross-Entropy and Solution Quality values. The black line indicates the number of epochs for which Wilder et al. trained.

QPTL-network only performs worse and worse over time. In the latter figure, this is not as clear-cut; here, we see that the QPTL network actually learns to reduce the cross-entropy rate up until about 25 epochs, after which it goes up again. In fact, the QPTL-network performs better on the test set than the ML-network does, due to overfitting on the training set. In Figures 3.1c and 3.1d we see the learning curves for the solution quality instead. In the former we see that only the QPTL network manages to improve its performance on the training set - although that does not lead to a significant performance increase on the test set. In the latter, we see that the QPTL-network very quickly achieves the maximum possible solution quality value on the training set; within 3 or 4 epochs or so. Note, however, that the ML-network also managed to achieve this value, although at a slower pace. This indicates that Wilder et al. [46] did not train the ML-network for long enough to achieve good results using the ML-network and highlights the importance of using a validation set. Another interesting observation is present between Figures 3.1b and 3.1d: the solution quality of the ML-network increases *while the cross-entropy loss increases*. So while the ML-network is becoming less accurate by its own measure, its predictions still manage to lead to better solutions.

3.4.2 Class imbalance

In this section we show the results obtained from enforcing an equal split between the positive classes and the negative classes. The ML-networks were trained using the item-samples after the various sampling methods had been applied to them. Because problem-samples can no longer be retained after resolving the class-imbalance, Stochastic Gradient Descent [11] was employed instead, with batch size 2500 for all networks (except 2L-down, for which we used a batch size of 100), for all ML-types. The results can be seen in Table 3.5.

Table 3.5: Mean + Bootstrapped 95% confidence interval for Bipartite Matching for ML-Networks after fixing class imbalance. Means are across all problem-samples in the test set.

	CE	Sol	AUC
1L	0.23 (0.00)	2.71 (-0.25, 0.27)	0.50 (0.01)
1L-Up	0.66 (0.01)	2.43 (-0.32, 0.31)	0.51 (0.01)
1L-Down	0.81 (0.02)	2.59 (-0.28, 0.26)	0.51 (0.01)
1L-SMOTE	0.63 (0.02)	2.69 (0.29)	0.51 (0.01)
2L	0.48 (0.01)	5.07 (-0.34, 0.31)	0.54 (0.01)
2L-Up	0.55 (0.02)	5.95 (-0.34, 0.35)	0.58 (0.01)
2L-Down	1.97 (0.11)	4.89 (-0.35, 0.33)	0.55 (0.01)
2L-SMOTE	0.62 (0.02)	6.21 (-0.41, 0.45)	0.57 (0.01)

Note: -'Up' refers to upsampling, '-Down' refers to downsampling, '-SMOTE' refers to the SMOTE procedure.

Table 3.5 shows the results for the conventional network after applying the various sampling methods. The sampling methods do not seem to aid one-layer networks in particular, as they all achieve about the same performance in terms of solution quality. However, a different story is told by the results for the two-layer networks; here, all two-layer networks that used one of the sampling methods showed improvements over the baseline. The SMOTE procedure returned the best results, although the confidence intervals for 2L-Up and 2L-SMOTE are overlapping, indicating similar performance. Downsampling performs poorly; throwing away samples via downsampling is bound to lose information. Using upsampling conferred roughly a 25% increase in terms of solution quality over base the baseline. Note that these results were also obtained after training for 100 epochs, so combining upsampling and increasing training time nearly *doubled* the solution quality (referring back to Table 3.1) of the 2L-ML network. However, the 2L-ML network is still outperformed by the 2L-QPTL network, although the gap has been closed to a large degree. The remaining gap may be present due to optimal performance not necessarily requiring a 50:50 split and, again, due to different choice of distribution of errors.

3.5 Conclusion

In this chapter, we investigated the results of Wilder et al. [46] w.r.t their Bipartite Matching experiment. To answer the research questions established at the start:

RQ1: *What are the effects of class imbalance on the QPTL-framework when compared to conventional machine learning approaches?* Resolving the class imbalance present in the Bipartite Matching experiment of [46] led to a performance increase of more than 35% in terms of solution quality for the 2L-ML network (after taking into account the performance increase obtained from the longer training duration). The reason is likely that the ML-network is able to better take into account the importance of positive samples; if no positive predictions are made in the problem-sample, no good solution value can be obtained, as the solver either selects item-samples randomly, or does not select any at all. For a severe class imbalance (in favour of samples with a negative label), the ML-network is more likely to label a particular (unknown) sample as negative, simply because it has seen more of those samples. This puts a bit of a damper on the results obtained by 2L-QPTL,as initially it more than doubled the performance of the 2L-ML network (7.40 versus 3.31), whereas now it increases performance by roughly 17% (7.40 versus 6.21). Wilder et al. [46] states that "no accuracy measure is well-correlated with solution quality". While that may be true for the other experiments performed, for the bipartite matching experiment there does seem to be a correlation between AUC and solution quality (Ferber et al. [21] also note this but do not seek a further explanation for the reason behind this). In fact, the QPTL-procedure in the bipartite matching setting could be to some extent considered as a differentiable proxy for AUC maximization, and it is likely that QPTL does not work as well on datasets that do not have a class imbalance present in the bipartite matching setting.

RQ2: *How does pre-training affect QPTL performance?*We found that pre-training causes worse performance than simply using QPTL alone. This is likely because the optimal weight configuration for an ML-network and for a QPTL-network are so different that pre-training merely manages to move the starting configuration (before training via the QPTL approach) of the network further away from the optimal QPTL-configuration. This is consistent with other findings; Ferber et al. [21] shows similarly (poor) results when it comes to pre-training and Mandi et al. [36] report no difference in performance.

3.5.1 Discussion

We found that training for a longer duration positively benefited the 2L-ML network, with an increased performance of nearly 50% compared to baseline results. Another interesting finding is that the performance of the 2L-ML network in terms of mean solution quality on the test set can increase, despite its cross-entropy loss on the test set increasing. This is particularly interesting because it was not able to take into account the structure of the problem-samples. This serves to further highlight the discrepancy between conventional accuracy measures and the resulting solution quality induced.

4 Predict-and-Optimize for Multi-Class Classification

4.1 Research Questions

RQ3: *How should one tackle multi-class classification datasets using PnO approaches, when using neural networks?*

RQ4: *How are PnO approaches affected when the objective parameters between problem-samples are identical versus when they are randomly selected?*

4.1.1 Motivation

RQ3

Currently, PnO research has primarily been applied in a binary classification or regression setting context [13, 19, 46]. One would think that the multi-class classification setting should be straightforward as well, as it is a typical problem encountered in an ML context. However, PnO approaches face a problem during training: their predictions are typically fed to a combinatorial optimization solver. Conventionally, neural networks in a multi-class classification setting output a probability distribution for each class using the `softmax` function (see Equation 4.1, with **x** indicating an n-dimensional vector) [25]

$$\texttt{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{4.1}$$

That is, not a singular value. In contrast, in regression or binary classification the output is a single value, and therefore they can easily be used as input. In regression, a predicted numerical value should attempt to be close to the real value of a sample. In binary classification, a value in the interval (0,1) (due to the sigmoid output of the neural network) is returned; convenient, because this can be used directly in the solver as indication of how likely the item-sample is to be 1 or 0 (the degree of closeness to 1 can at the same time also indicate how likely it is to be 1).
One may ask why it is not simply possible to use the `argmax` function on the probability distribution that the neural network outputs in a multi-class classification setting, to obtain the predicted label that is most likely. The `argmax` of a vector outputs the indices of the vector with the largest values, and as such can be used to find the predicted class of a sample by applying it to the obtained confidence scores. However, the `argmax` cannot be used during gradient-based training with back-propagation as in the QPTL approach, as the argmax function is not differentiable.

$$\texttt{argmax}(\mathbf{x}) = \{i : x_i \geq x_j \quad i \in [1 \dots n], \forall j \in [1 \dots n]\} \tag{4.2}$$

More formally, the term $f_\omega(s)$ in Equation 2.5 is replaced with $\texttt{argmax}(f_\omega(s))$, which means the third gradient is no longer defined. The same is true for the SPO-approach: the obtained subgradient only refers to the predictions after the argmax has been applied to the neural network output. Because of the increased number of network outputs (each of which outputs a confidence score corresponding to a label), it is no longer clear how to construct the sub-gradient. So how *should* we approach prediction in a PnO-setting for multi-class classification? Should we attempt to find a fix such that can we still use probability distributions, or should our neural networks output ignore the fact that our dataset is a classification problem at heart and simply treat it as a regression problem (i.e., simply use a linear output rather than a softmax output)? This question is particularly relevant when it is considered that previous research has already shown that there is a strong disconnect between the predicted values of item-samples and their real values when it comes to their use in an optimization setting.

RQ4

We consider training and testing with problem-samples that all have the exact same identical objective parameters and training and testing with problem-samples that all have *different* objective parameters. We call these two scenarios S_I and S_D respectively. We investigate these two scenarios to see if different PnO approaches perform better in one of the two settings, which allows us to make statements about how the homogeneity of the problem samples affects the performance of such approaches. That could be useful as indicator for when to use which PnO approach. As example, imagine that our problem-samples are all knapsack problems of size 10 where the item-samples have the labels $[0, \ldots, 9]$. This would be scenario S_I. When the objective parameter differs between problem-samples (there may be multiples of one class and none of another, for example), we are evaluating results in S_D.

4.2 Experiments

We evaluate the performance of neural networks with a linear output (normally used for regression tasks), and neural networks with a sigmoid output (normally used for binary classification tasks) that are trained via PnO approaches. In addition, we use the Gumbel-Softmax Straight-Through Estimator [26] for SPO and QPTL to allow for discrete multi-class prediction and evaluate their performance. As baseline reference we conventionally train a network with a softmax output (typical for multi-class classification) using the cross-entropy loss function.
We evaluate this performance across the two scenarios described earlier.

4.2.1 Choice of optimization problems

We decided to keep working with 0-1 Bipartite Matching (BM) problems, and we use unit-weighted 0-1 KP problems (with capacity 3 (note that [13] considers KP problems with capacity $10\%, 30\%, 50\%$ - our setting thus also uses a capacity of 30%, but we do not base this choice on their results) as additional problem domain. Both of these problems have seen plenty of attention [13, 46, 36, 21]. Note, however, that those papers were usually interested in the performance of PnO approaches versus standard ML approaches. Here, we are primarily concerned with the differences in performance of a neural network trained with the same PnO approach, but with different output activation functions. Both Mandi et al. [36] and [13] have made comparisons in the performance of QPTL and SPO before, and that is continued here and in the next chapter. Note that Demirović et al. [13] has shown that PnO is rather difficult in the KP domain. Ferber et al. [21] has also shown that QPTL performs relatively poorly in the KP domain, where they put forward that the poor performance may be due to the fact that the constraints are "not as combinatorial".

4.2.2 Data Generation

We utilize the MNIST dataset [33] to create our item-samples and problem-samples. The MNIST dataset is a hand-written digit dataset on which even simple neural networks can obtain good results (in 1998, a neural network without a hidden layer, i.e., a linear classifier already obtained a 12% test error rate [33]). The images of the digits were normalized first before the experiments were performed. This particular classification dataset was chosen because its labels can also easily be seen as numerical values. Because the classification task is rather simple, and no research has yet been done on how well convolutional neural networks perform in a PnO-setting, we opted to evaluate only the performance of non-convolutional neural networks. This meant that we flattened the 28×28 images to obtain feature-vectors of size 784. It is clear what the item-samples are in this case; they are simply the images themselves (or rather, the feature-vectors that make up the images). Their labels are their values. But how do we construct the problem-samples? The procedure per problem-domain is listed in Table 4.1.

Table 4.1: Table that describes data-construction for both problem domains.

Problem Domain	Data-Construction
KP	Each item is represented by an image. Problems of size ten (meaning the objective parameter is size 10) are constructed either by taking one random sample from each class (the identical setting) or by randomly selecting ten samples from the entirety of the dataset (the different setting).
BM	Ten nodes are on each side. Each node is represented by an image, which can be either randomly sampled per class (meaning that that every class appears exactly once on each side – the identical setting) or from the entirety of the dataset (the different setting). In this setting we create two new additional different variants; in the concatenated setting (BM(C) from here on), the item-samples are the edges between the nodes. The feature-vectors that represent these edges are the concatenation of the feature-vectors of each node (from left to right). The label of the item-samples is the multiplication of the concatenated digits. In the unconcatenated setting (BM(U)), the item-samples are the nodes (individual images) themselves. The images are first individually predicted, after which the predictions are multiplied with each other. To make more clear what is meant by concatenated data, see the visual example in Figure 4.1.

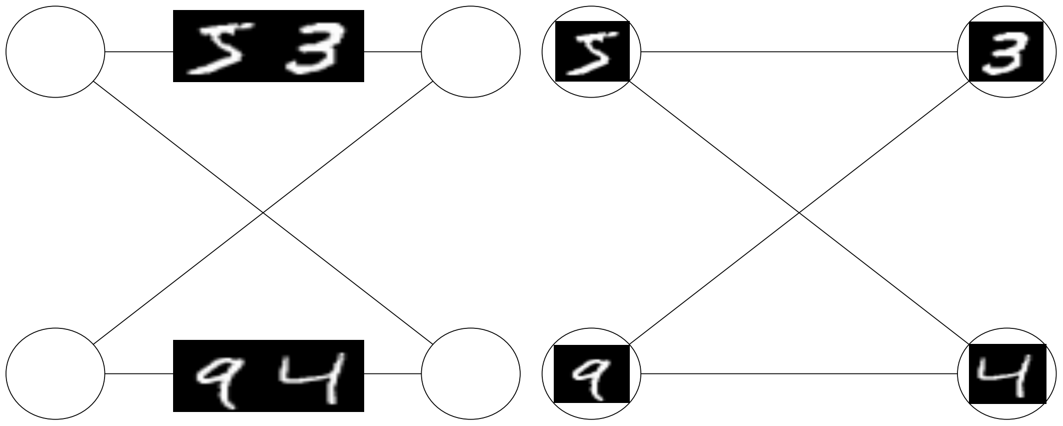

(a) Concatenated dataset for bipartite matching. (b) Unconcatenated dataset for bipartite matching.

Figure 4.1: A visual example that shows the difference between the concatenated and unconcatenated bipartite matching datasets. In Figure 4.1a, predictions are made using edges, which consist of concatenated images. The downside here is that this introduces a class imbalance, as the label of the concatenated images is the multiplication of the two digits. In Figure 4.1b, neural networks are fed the individual images (representing nodes), of which the predictions are then multiplied with the predictions of the nodes on the other side to compute the values of the edges, which are the input for the objective value of the optimization problem. This could complicate gradient descent for PnO-based approaches.

We created 10 different datasets, with different levels of identical Gaussian noise; the Gaussian noise added had mean 0 and a standard deviation from 0 to 900, with increments of 100. Note that this is similar to the experiment performed in [19], where they increase the degree of the polynomial to make their artificial dataset increasingly more non-linear (and thus harder, for linear estimators). Here, we make the datasets harder by adding more and more noise (although do notice Elmachtoub and Grigas [19] evaluates performance between two levels of noise). In the experiments of Elmachtoub and Grigas [19], performance between an SPO-trained estimator and conventionally-trained estimators were compared - here, comparisons are made between SPO-trained networks and QPTL-trained networks with different output activations, and the conventionally-trained network is used as reference.

4.2.3 Neural Network Training Specifications

Nine different neural networks were trained. An explanation of each neural network can be seen in Table 4.2. All neural networks had a single hidden layer, and were trained for 50 epochs, using Adam with a learning rate of $1e^{-4}$. The reason for the single hidden layer is that the previous section, based on [46], has shown good performance with simple networks. In addition, the MNIST dataset is generally considered to be an 'easy' dataset. The problem-samples created for each problem domain are split up in a 60:20:20 train:validation:test set ratio. In the BM domain, the training set consisted of 1893 problem-samples and the validation and test set consisted of 631 problem-samples each. In the KP domain, the training set consisted of 3787 problem-samples, and the validation and test set consisted of

1263 problem-samples each. The validation set is used to select the learnt model with the best mean performance on the validation set, across all epochs. Note that performance on the validation set is measured by the mean solution value obtained across all problem-samples present in the validations set for *all* neural networks, even those that are trained according to the conventional ML paradigm. This is because that is the ultimate goal we are interested in, and it would be unfair for comparison purposes to instead select conventionally trained neural networks by their respective loss functions instead. This is identical to the approach used in [36]. We keep track of the learnt weights that display best performance on the validation set, and use those weights as our final network for evaluation on the test set. Results are averaged across 10 runs, with different random seeds. As such, between runs different training, validation, and test splits are randomly selected (Monte-Carlo Cross-Validation [18]).

Table 4.2: Table that describes data-construction for both problem domains.

Network Name	Explanation
NN-ML-C	A neural network that is trained using the CE loss. This neural network has the same amount of outputs as the number of classes available. "C" for classification.
NN-QPTL-R	A neural network that is trained using the QPTL approach, with a single numerical output. σ is multiplied by the following equation after each epoch: $$(\frac{\sigma_{end}}{\sigma_{start}})^{\frac{1}{n_{epochs}}}$$ This also holds for all following QPTL-variations. "R" for "Regression"
NN-QPTL-B	A neural network that is trained using the QPTL loss-function with a sigmoid output, which restricts the neural network output to a value between $[0,1]$. "B" for "Binary (Classification)".
NN-QPTL-C	A neural network that is trained using the QPTL approach, with a straight-through gumbel-softmax output. This allows the neural network to sample from the estimated class distribution. The temperature is slowly decreased, making the sampling more and more accurate as training time increases. The temperature used in the gumbel-softmax distribution τ starts at 1 and ends at 0.2. τ is multiplied by the following equation after each epoch: $$(\frac{\tau_{end}}{\tau_{start}})^{\frac{1}{n_{epochs}}}$$
NN-QPTL-C-Fixed	Same as above, but the temperature is held constant.
NN-SPO-B	A neural network that is trained using the SPO+-loss, with a sigmoid output, which restricts the neural network output to a value between $[0,1]$. Note that the sigmoid activation function is also applied to the class labels.
NN-SPO-R	A neural network that is trained using the SPO+-loss, with a linear output, which means the neural network has an unrestricted numerical output.
NN-SPO-C	A neural network that is trained using the SPO+-loss, with a straight-through gumbel-softmax output. The same annealing procedure is used as for NN-QPTL-C
NN-SPO-C-fixed	Same as above, but the temperature is held constant.

4.2.4 Multi-Class Classification with QPTL and SPO

Now we describe in greater detail how we obtain discrete decisions from the probability distribution output by the NN-QPTL-C networks; we do so by using a Gumbel-Softmax activation function as output rather than a softmax output. The Gumbel-Softmax distribution is a continuous distribution that can be used to approximate sampling from a categorical distribution[26]. The authors show that the softmax function can be used "as a continuous, differentiable approximation to argmax, and generate k-dimensional sample vectors $y \in \nabla^{k-1}$ where

$$y_i = \frac{e^{(log(\pi_i)+g_i)/\tau}}{\sum_{j=1}^{k} e^{(log(\pi_j)+g_j)/\tau}} \quad \text{for } i = 1, \ldots, k \tag{4.3}$$

". π denotes class probabilities, τ denotes the temperature and $g_1 \ldots g_k$ denote samples that are i.i.d. drawn from Gumbel$(0,1)$ (paraphrased from [26]). The closer τ gets to 0, the closer y gets to being a true one-hot sample. The Gumbel-Softmax distribution is fully differentiable as long as $\tau > 0$ and therefore backpropagation can be used during neural network training. The authors note, however, that Gumbel-Softmax samples cannot be equated to true discrete samples from the underlying categorical distribution; the lower the temperature becomes, the closer to discrete the samples, but at the price of greater variance of the gradients. This Gumbel-Softmax distribution is visualized in Figure 4.2. In the experiments the authors perform, they either slowly anneal the temperature to a small value (which is also one of the approaches used here) or keep it fixed. This, however, has not yet resolved the issue of requiring discrete samples for use in the combinatorial solver. Luckily, Jang et al. [26] also provide the Straight-Through (ST) Gumbel-Softmax Estimator, where the `argmax` is used during the forward pass and the Gumbel-Softmax distribution during the backward pass. One important thing to note here is that the QPTL approach already features a parameter that reduces per epoch, γ (indicating the degree of quadratic relaxation), which could complicate training. Weight updates learnt at a certain temperature τ may no longer be (as) relevant for when γ is lowered afterwards. Note that Jang et al. [26] also uses the MNIST dataset (although binarized), but their experiments are generally of a different nature. Namely, they primarily focus on generative modeling. However, they also perform a structured (output) prediction task (prediction of the lower half of binary MNIST images), which has been linked to the PnO-setting by [19, 46].This is because the PnO-setting can also be seen as directly predicting a solution based on the features of the item-samples, rather than using intermediate predictions of the objective parameter first and then use a solver to optimize, Of course, one must make sure to retain feasibility. To our knowledge, the Gumbel-Softmax estimator has not been used for multi-class classification in a PnO setting (i.e. its predictions are used as objective parameters of a discrete optimization problem) yet. However, the Gumbel-Softmax estimator has seen use in other machine learning paradigms such as deep reinforcement learning to sample discrete actions after which a reward is returned (see for example [44]), which can be said to be of a similar nature. This could be seen as a more general setting. Note that Liu et al. [34] use the Gumbel-Softmax to directly optimize over combinatorial problems on graphs - however, here the graph to optimize over is given as input (with node labels known) and instead the *solutions* are directly predicted. A punishment variable is introduced such that infeasible solutions are discouraged. In our setting, node labels (in as far

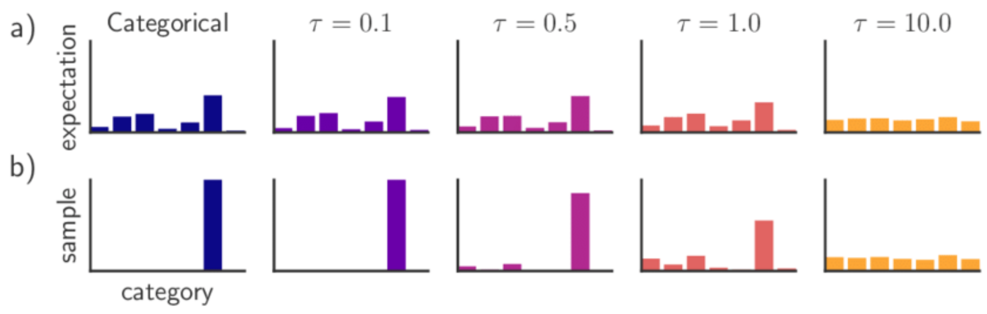


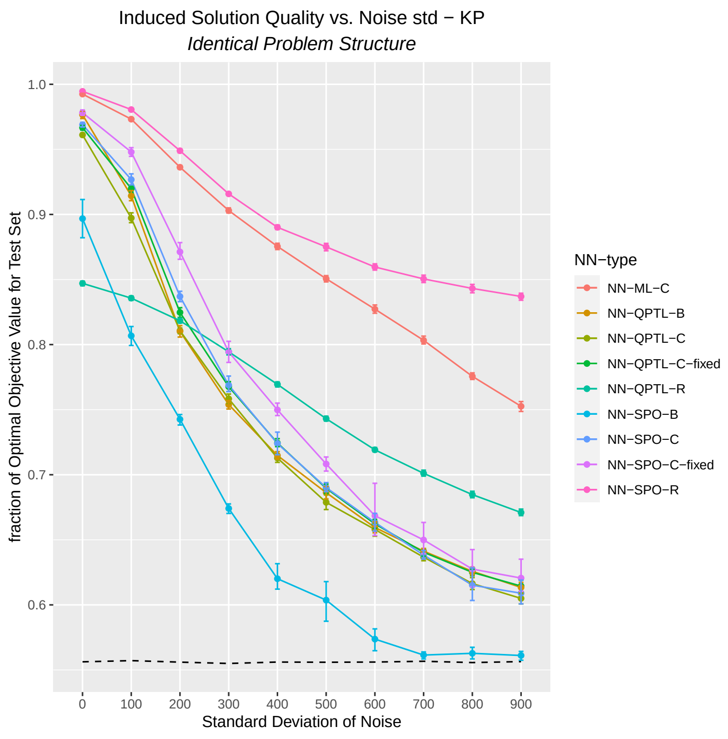
Figure 4.2: Figure that illustrates the change in the Gumbel-Softmax distribution as the temperature τ increases. Figure taken from Jang et al. [26].

as this term applies to our problems) are unknown, and we predict the node labels which induce solutions. In this case, the induced solutions can never be infeasible. Note that Balghiti et al. [3] obtains generalization bounds by transforming Predict-and-Optimize to multi-class classification problems (with their optimal solutions the class), but to our understanding this is different from considering an actual multi-class classification dataset as objective parameters and how predictions should be made by a neural network.

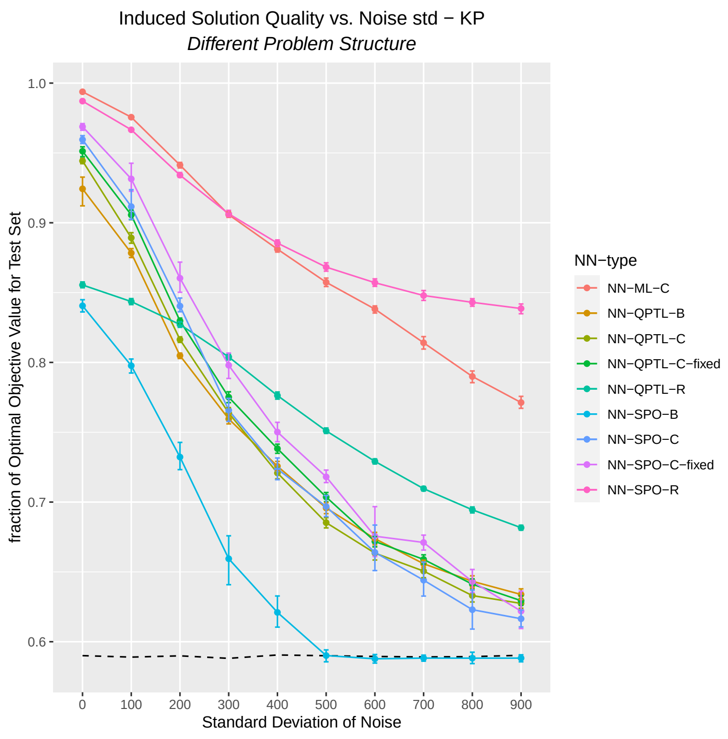
4.3 Results

We will be using the term "Induced Solution Quality" to signify the quality of the decision that is made using the predictions output by a neural network. It is expressed as fraction of the optimal solution quality that could have been made had the right predictions been made. The black lines in the plots indicate the Induced Solution Quality when the classes are uniformly randomly sampled from the available classes.

4.3.1 KP Domain



(a) Identical Problems



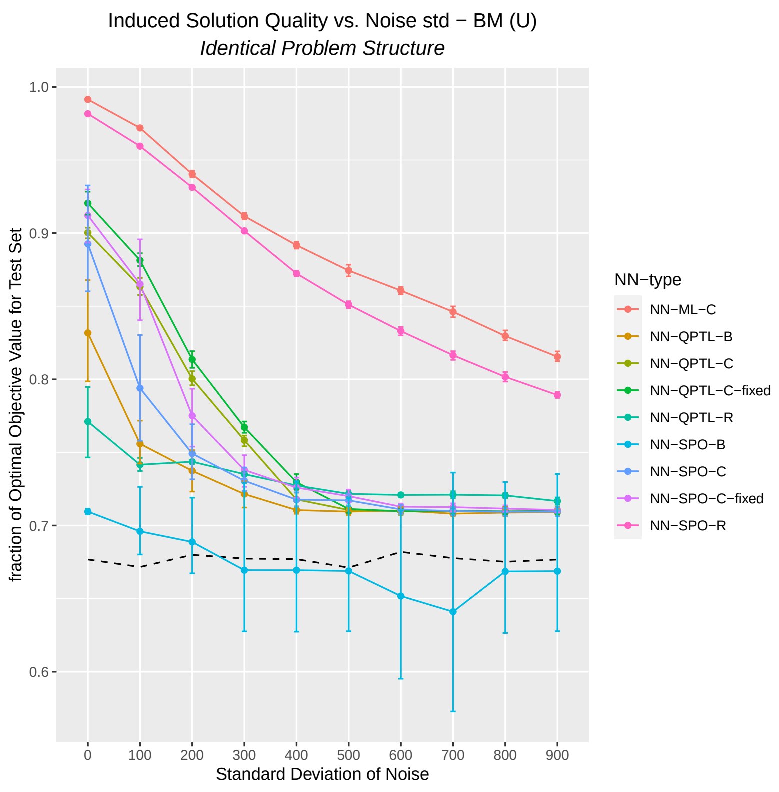
(b) Different Problems

Figure 4.3: PnO with knapsack using MNIST. Black line indicates the solution quality of random predictions. True for following pictures as well.

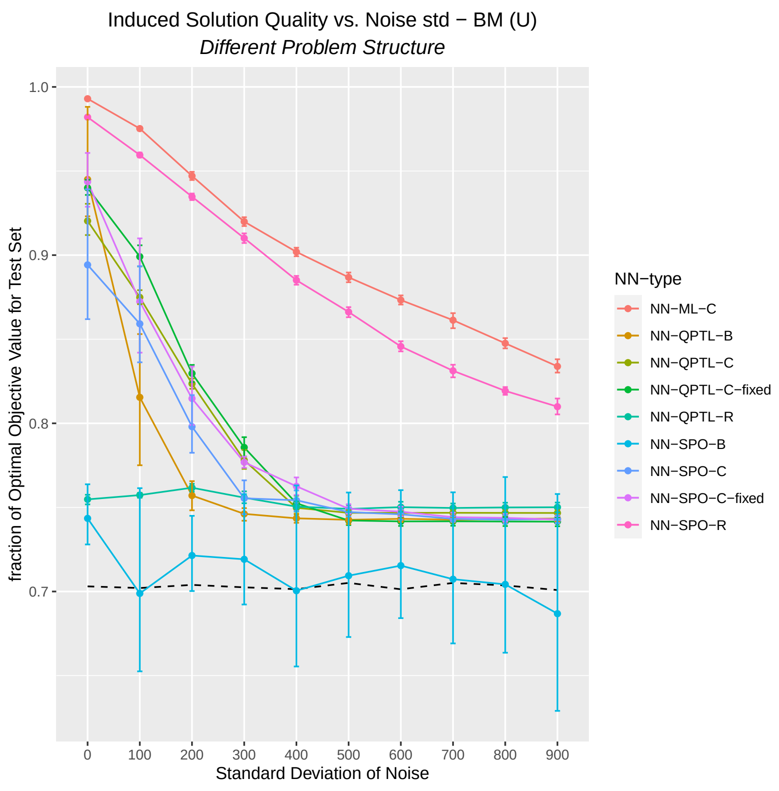
In Figure 4.3, the results for the knapsack-domain are displayed, where problems have identical problem structure. That is, the problem-samples contain one sample

per class, of which there are 10. As such, the optimal solution value is $9+8+7=24$ for each problem. Solution quality is indicated by fraction of the optimal solution value obtained. We see that that both the NN-ML-C and NN-SPO-R networks tend to perform best: at all noise levels except for zero noise, NN-SPO-R outclasses NN-ML-C. The other networks perform rather poorly compared to those two. All Gumbel-Softmax networks show fairly similar performance, with NN-SPO-C showing the best performance. However, whereas NN-SPO-R clearly outclasses the NN-SPO-C variants and NN-SPO-B, all QPTL-networks perform on par with each other. NN-QPTL-B and NN-QPTL-C(-fixed) show very similar performance. The odd one out is NN-QPTL-R - the likely reason is that NN-QPTL-R tends to cause crashes. This is likely due to an observation by Demirović et al. [13] where gradients become very large. We recalled this at too late a point, and did not implement their proposed solution. Predictions become so large that the solver encounters an integer overflow. We catch this exception by skipping the backward-forward propagation for that particular problem-sample when it happens. Note that it is impossible for NN-QPTL-B and NN-QPTL-C to causes crashes in such a manner, as their outputs have been restricted to predefined values. We note that this behaviour was also not seen in NN-SPO-R. It seems as if this phenomenon has a regularizing effect, however, because at higher noise levels NN-QPTL-R outperforms the other QPTL-variants. It is likely that if this issue did not occur, NN-QPTL-R would show similar performance as NN-QPTL-B and NN-QPTL-C. As mentioned, the overall poor performance of the QPTL-networks is not particularly weird for the KP-setting.
The reason that NN-SPO-C performs poorly compared to NN-SPO-R may be due to the stochasticity of the Gumbel-Softmax output activation functions. NN-SPO-B likely performs poorly due to reduced expressivity - the class labels are all fairly close to 1 after the sigmoid activation function has been applied to them. As such, NN-SPO-B is less able to distinguish between classes due to precision issues.

4.3.2 BM Domain

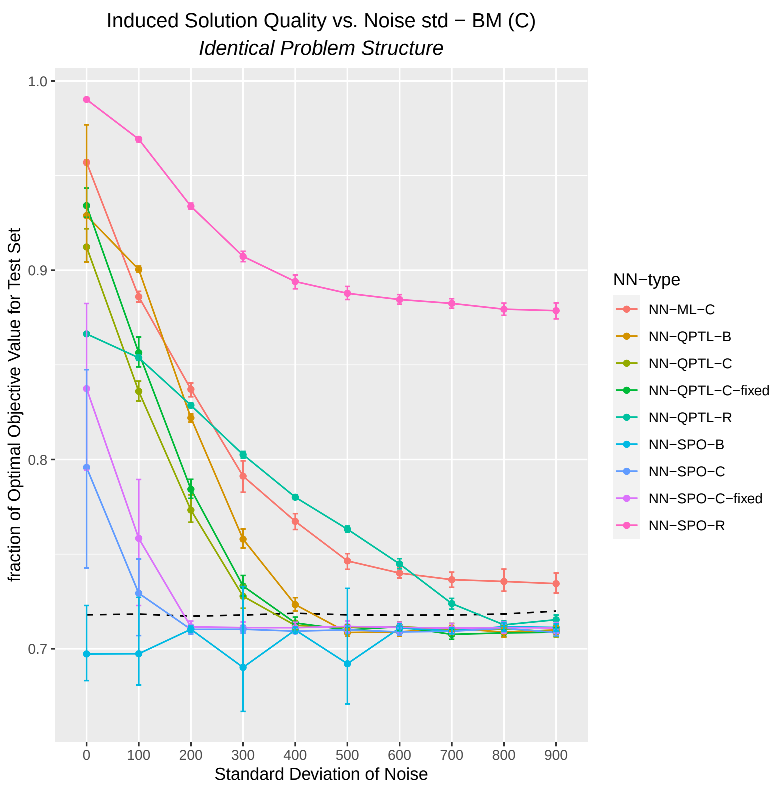


(a) Identical Problems

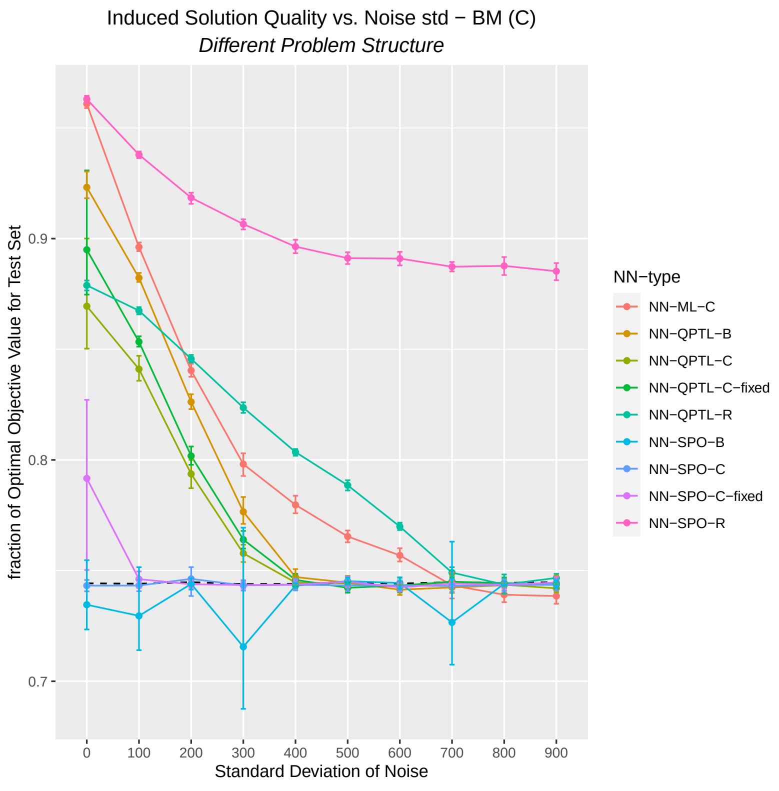


(b) Different Problems

Figure 4.4: PnO with knapsack using MNIST



(a) Identical Problems



(b) Different Problems

Figure 4.5: PnO with BM using MNIST

Moving on to the BM(U) domain (displayed in Figure 4.4), we see that NN-ML-C performs best, followed closely by NN-SPO-R. The NN-SPO-C and NN-QPTL-C variants seem relatively unperturbed by the unconcatenated setting, but NN-QPTL-B, NN-QPTL-R and NN-SPO-B show relatively poor performance in this setting. This may partially be explained by the fact that gradient descent is complicated by the multiplication of predictions (that is, the predictions of the nodes on the left side are multiplied with the predictions of the nodes on the right side). That, however, does not explain the good performance of NN-SPO-R.
However, things take a turn for the better for PnO-approaches in the BM(C) setting (Figure 4.5). Here, the majority of PnO methods perform better when compared to NN-ML-C. In fact, NN-SPO-R again completely outperforms NN-ML-C at all levels of noise. This highlights the importance of framing the problem correctly, but also again shows that PnO approaches are less negatively affected by the presence of class imbalances than standard approaches are in the PnO setting. The SPO-variants besides NN-SPO-R perform more poorly than their QPTL-counterparts. The poorer performance of the NN-SPO-C variants may be explained by the increased number of degenerate solutions (due to the class imbalance) (which is a problem, as such solutions might induce different regret [19]. Note again that the QPTL-variants show similar performance, with NN-QPTL-R outperforming NN-ML-C for some values of noise.

4.3.3 Effects of Optimization Problem Design

In this section we analyze whether ML-networks and PnO-networks see significant differences in performance when moving from the identical problem structure setting to the different problem structure setting. We use NN-SPO-R and NN-QPTL-R as representatives of PnO approaches.
We show t-test results for the difference in performance for NN-ML-C, NN-SPO-R, and NN-QPTL-R when they train and test on problem-samples with identical objective parameters and random objective parameters in the BM(C) domain, in Tables 4.3, 4.4, and 4.5. We notice that all approaches show rather marginal changes in performance. The SPO approach sees a performance decrease at low levels of noise, perhaps because of the increased incidence of degenerate solutions. QPTL consistently sees a statistically significant performance increase across all noise levels. The same is true for the ML approach, however for multiple noise values this is not statistically significant.

Table 4.3: Welch two-sample t-test results for NN-SPO-R in BM(C) domain, comparing results between identical (I) and different (D) objective parameters. Mean Difference is results of D subtracted by results of I.

Std	Mean Difference	p-value
0	-0.027	0.000
100	-0.031	0.000
200	-0.015	0.000
300	-0.001	0.689
400	0.002	0.370
500	0.003	0.173
600	0.006	0.008
700	0.005	0.015
800	0.008	0.009
900	0.007	0.046

Table 4.4: Welch two-sample t-test results for QPTL in BM(C) domain, comparing results between identical (I) and different (D) objective parameters.

std	estimate	p.value
0	0.013	0
100	0.014	0
200	0.017	0
300	0.021	0
400	0.023	0
500	0.025	0
600	0.025	0
700	0.025	0
800	0.031	0
900	0.031	0

Table 4.5: Welch two-sample t-test results for ML in BM(C) domain, comparing results between identical (I) and different (D) objective parameters.

std	estimate	p.value
0	0.004	0.004
100	0.010	0.000
200	0.003	0.207
300	0.007	0.206
400	0.012	0.001
500	0.019	0.000
600	0.017	0.000
700	0.007	0.091
800	0.004	0.354
900	0.004	0.251

We also show t-test results for the difference in performance for NN-ML-C, NN-SPO-R, and NN-QPTL-R when they train and test on problem-samples with identical objective parameters and different objective parameters in the KP domain, in Tables 4.6, 4.7, and 4.8. Here, performance differences are even more marginal, although SPO nearly always sees a very slight performance decrease.

Table 4.6: Welch two-sample t-test results for SPO in KP domain, comparing results between identical (I) and different (D) objective parameters.

Std	Mean Difference	p-value
0	-0.007	0.000
100	-0.014	0.000
200	-0.015	0.000
300	-0.009	0.000
400	-0.005	0.008
500	-0.007	0.007
600	-0.003	0.207
700	-0.003	0.308
800	0.000	0.960
900	0.002	0.505

Table 4.7: Welch two-sample t-test results for QPTL in KP domain, comparing results between identical (I) and different (D) objective parameters.

std	estimate	p.value
0	0.008	0.000
100	0.008	0.000
200	0.009	0.000
300	0.009	0.000
400	0.007	0.001
500	0.008	0.000
600	0.010	0.000
700	0.009	0.000
800	0.010	0.000
900	0.011	0.000

Table 4.8: Welch two-sample t-test results for ML in KP domain, comparing results between identical (I) and different (D) objective parameters.

std	estimate	p.value
0	0.001	0.023
100	0.002	0.007
200	0.005	0.002
300	0.003	0.087
400	0.006	0.005
500	0.007	0.005
600	0.011	0.000
700	0.011	0.002
800	0.014	0.000
900	0.019	0.000

These results seems to imply that NN-SPO-R generally prefers for the objective parameter of problem-samples to remain identical between problem samples, although performance differences are slight. The likely cause is the increased incidence of degenerate solutions with the same optimal objective value. QPTL and ML generally see a slight performance increase. This may be because it is easier to induce better solutions in general (looking at Figure 4.3, we see that the black line that indicates the quality of solutions induced by random predictions is higher in the Different setting than the Identical setting).

4.4 Conclusion

RQ3: *How should one tackle multi-class classification datasets using PnO approaches, when using neural networks?* For the SPO approach it is clear from the experiments that NN-SPO-R is preferred over the other alternatives. For the QPTL-framework however, we find that all variants show similar performance. We therefore suggest to use NN-QPTL-B as default variant for QPTL; as it prevents integer overflows from occuring during solving, and is applicable to every single type of dataset (regression, binary classification, and multi-class classification). The exception is when labels of samples can be negative - because the sigmoid output activation function is constrained to the range (0,1), it might still cause the solver to pick items that worsen performance. In that case, we suggest the use of NN-QPTL-R.

RQ4: *How are PnO approaches affected when the objective parameters between problem-samples are identical versus when they are randomly selected?* We have evaluated the effects of the structure of the objective parameters on the performance of PnO networks compared to conventional ML approaches. We considered two settings, one where the objective parameters were identical across all problem-samples (setting S_I), and another were they were randomly selected (setting S_D). We considered NN-SPO-R and NN-QPTL-R as representatives of the PnO approach and NN-ML-C as representative of conventional ML approaches. We have shown that both the NN-QPTL-R network and NN-SPO-R network see significant differences in performance when comparing results from setting S_I with those from S_D. These results seem to indicate that NN-SPO-R prefers the objective parameters between problem-samples to be identical, while NN-QPTL-R sees a significant increase when objective parameters differ between problem-samples (likely because it is easier to obtain solutions of higher quality in S_D). However, the changes in performance are rather marginal.

5 Problem-Sample Resampling in Predict-and-Optimize

5.1 Research Question

This chapter seek to answer the following research question:

RQ5: *How does problem-sample resampling affect the performance of PnO approaches? Does the use of a Multiple-Input Multiple-Output (MIMO) network further improve performance?*

5.1.1 Motivation

The idea behind **RQ5** is that PnO-methods could see performance increases by constructing multiple problem-samples from the same pool of item-samples, because it learns to average out the desirability of an item-sample when it is seen in conjunction with many different samples. For example, in a knapsack-context, whether an item-sample x with value five is desirable or not is dependent on the other item-samples available. Therefore, seeing x in multiple problem-samples with different item-samples should help to average out the desirability of x. To our knowledge, in current literature datasets have been used such that each item-sample appears in only one problem-sample. For example, Mandi et al. [36] uses 552 predefined training instances which consist of daily half-hour energy prices. The bipartite matching experiment from Wilder et al. [46] constructs twenty-seven different bipartite matching problems by partitioning the cora dataset [38]. Thus what is suggested in this chapter is that repeating the partitioning multiple times and adding all of these partitioned problem-samples together in the training set could improve the performance of PnO-based networks while standard networks should be relatively unaffected in terms of solution quality performance. After all, the set of item-samples does not change, only the set of problem-samples. We call the repeated creation of problem-samples given the same pool of item-samples *problem-sample resampling*. Note that Elmachtoub and Grigas [19] investigates the effect of the training set size on SPO (and finds that SPO works better than competitors when more data is available). However, that involves the addition of new labeled data to the training set, whereas constructing more problem samples from the same small pool of data does not. Wilder et al. [45] investigates decision-focused learning on graphs, where they use a method that differs somewhat from QPTL in that the optimization problem is not solved by a solver during training. Furthermore, they use a different network structure ("ClusterNet") which uses a graph embedding layer. In their work they find that ClusterNet is extremely sample-efficient. If the same holds true for QPTL, then we should not expect to see very good performance increases for QPTL when additional problem-samples are sampled from the same small dataset, as it will already perform very well from the start.

To make this more clear, consider the knapsack problem in Equation 5.1.

$$\begin{aligned} \max_{x} \quad & \begin{bmatrix}1 & 2 & 3\end{bmatrix} x \\ \text{s.t.} \quad & \begin{bmatrix}1 & 1 & 1\end{bmatrix} x \leq 1, \\ & x \in \{0,1\}^3 \end{aligned} \tag{5.1}$$

This is a knapsack problem where only one item is allowed to be picked. Clearly this should be the item with value 3. This particular knapsack problem might give the impression that the desired item-sample (the item-sample with label 3) is of a very high value, as it should be chosen. However, that is only true within the context of this particular knapsack problem. Given that the MNIST dataset contains numbers in the range from zero to nine, the real value of the sample with value three should not be particularly high; the predicted value should only be high when the other item-samples involved in the problem-sample have a lower value than three. By having this item-sample appear in other problem-samples, with other item-samples, networks trained via PnO approaches should be able to better estimate an item's desirability. Further improvements could be gained by employing a Multiple-Input Multiple-Output (MIMO) network. Such a network predicts the value of multiple item-samples at once and can take into account the presence of other item-samples available. As such, it does not have to rely on being able to average out the desirability of an item-sample, but can change its prediction for a given item-sample based on the other item-samples available in the problem-sample. This is a form of *multi-output* learning [43]. Multiple-output learning has been used before in a PnO context; but Demirović et al. [13] trains a multiple-output regressor via a traditional loss function, Wilder et al. [46] performs multilabel classifiation for each invididual item-sample but does not predict these labels for all item-samples at once (diverse recommendation setting), whereas Elmachtoub and Grigas [19] represents the entirety of the objective parameter of a problem-sample by a single feature-vector (there are no item-samples to speak of), and does not contrast performance of the multi-output learner with that of a single-output learner. In our case, each individual reward contained in the objective parameter (so not every reward) is associated with a single feature-vector (as is more typically seen in PnO literature [15][46][36]) and we predict the labels of all item-samples present in the problem-sample at once. Wilder et al. [45] performs semi-supervised learning on graphs using Graph Convolutional Networks [31], which can be likened to multi-output learning. We can employ a special type of multiple-output network called "Deep Sets", which is permutation-equivariant and as such can reduce the number of problem-samples required for training.[47] The curse of dimensionality [4] dictates that the number of samples required for an estimator to achieve a certain level of performance measure scales exponentially with the number of features: because predicting the labels of n item-samples with m features each at once can be seen as having feature vectors of size $n \cdot m$, it is clear that a MIMO-network requires far more samples than a standard network to predict well. Luckily, we can artifically construct those samples through problem-sample resampling.

5.2 Experiments

We performed two experiments, which differ only by the types of neural networks used. In the first experiment, we focus on the neural network architectures used thus far in the thesis; the features of a single item-sample involved in a problem-sample are separately used as input to the network, and the output is the value of that item-sample. The second experiment uses neural networks that take in the features of all item-samples involved in a problem-sample at once, and predicts the value for each one of them at the same time. This means that the second experiment uses a MIMO-framework.

5.2.1 Data Generation

We perform the following experiment, again using the MNIST dataset: We start with a pool of $10 * n$ item-samples for the training and validation set respectively, with n the number of samples per class. That is, when n is 10, there are 100 item-samples present, 10 of each class.
We then vary the number of repeats r (the number of times we repeat each sample). For example, if r is 10, then each item-sample that was present in the original dataset now exists in it 10 times (note that if r is one, every sample exists in the original dataset once - the dataset does not change). We essentially scale up the entire dataset. From this pool of samples we randomly draw item-samples to construct problem-samples. Note that the construction of problem-samples using item-samples has to occur after the splitting of the training, validation and test sets. Otherwise, it becomes difficult to separate problem-samples into the training, validation and test sets because the item-samples contained within a problem-sample might belong to different sets. Note that this is similar to Stochastic Gradient Descent (SGD) [11]. In SGD, batches of samples are drawn randomly from a dataset of samples during training. These batches could also be used to construct new problem-samples repeatedly during training. However, this is particularly problematic for SPO, as this requires solving both the problem-sample with true objective parameters and predicted parameters repeatedly during training, rather than solving the problem-sample with predicted objective parameters once. This can be prohibitive when problem-samples grow very large or the optimization problem is particularly complex. Thus, we still randomly draw a large number of batches from the dataset of item-samples *beforehand*, but we solve them once, and then every epoch we train using only that set of problem-samples. Note that 10 repeats of problem-sample resampling would be equal to the first 10 epochs of Stochastic Gradient Descent - however, the following epochs would have different problem-samples compared to the dataset that used problem-sample resampling.
The remainder of the MNIST dataset is used to construct problem-samples without repetition, which are added to the test set. Note that the construction of problem-samples using item-samples has to occur after the splitting of the training, validation and test sets. Otherwise, it becomes difficult to separate problem-samples into the training, validation and test sets because the item-samples contained within a problem-sample might belong to different sets.
The validation set is used as in Chapter 4, to choose the saved weights of the network that have lead to the highest mean performance in terms of solution quality across the validation set.

In the experiments, n takes the values $\{10, 100\}$ and r takes the values $\{1, 10, 100\}$. That is, we vary between taking 10 and 100 samples per class from the MNIST-dataset for the training and validation set, and vary between 1, 10, 100 problem-sample resamples from the same pool of item-samples. We do so to simultaneously investigate the effects of low sample sizes (making it harder to learn), and the effects of the number of resamples on the performance of each network. We repeat each experiment 10 times, with different random seeds and we perform the experiments both in the KP domain and the BM domain. Whereas before we were interested in the effect of different output activation functions on performance, now we are interested in the effects of problem-sample resampling.

5.2.2 Neural Network Specifications

All neural networks had a single hidden layer. All neural networks were trained with a learning rate of $1e^{-4}$ using Adam [30]. All neural networks were trained for 100 epochs, except when both the number of resamples and the number of samples per class were 100 – in this scenario, they were trained for only 10 epochs (due to the large amount of time the experiment would have taken otherwise). All neural networks feature a single hidden layer; in the MIMO-setting this is a permutation-equivariant layer as laid out by [47]. The experiments were performed using the MNIST dataset with added Gaussian noise with a standard deviation of 200 and mean zero. We chose to use a relatively low value of noise because the limited amount of item-samples would by itself already make making accurate predictions rather hard. For both the training and validation set, n samples per class were randomly selected and shuffled into one pool of item-samples. Then, problem-samples were constructed from this pool of item-samples by randomly selecting 10 item-samples in the KP-domain, and 20 item-samples (10 nodes per side) in the BM-domain, until no more samples were left. This process was repeated r times. The problem samples in the test set were constructed using the remaining samples available in the MNIST dataset, again by randomly selecting 10, 20 item-samples per problem-sample. Note that this process is not repeated for the test set.

Deep Sets

In the MIMO-setting, we use the Deep Set architecture of [47]. The reason for this is that Deep Sets are permutation-equivariant neural networks; the features of each item-sample always lead to the same predicted value (with regards to all other item-samples available in the problem-sample), irrespective of the order in which the item-samples appear. More formally, consider the feature-vectors of n item-samples as $x_0, x_1, \ldots x_n$ and let P be a permutation of the range $0 \ldots n$. Then permutation-equivariance of a neural network f can be defined as follows (adapted from [47]):

$$f([x_{P(1)}, \ldots, x_{P(n)}]) = [f(x_{P(1)}), \ldots, f(x_{P(n)})] \tag{5.2}$$

Consider again the KP problem in Equation 5.1. The values in the objective function are the targets to be predicted, and have corresponding feature-vectors fv_1, fv_2, fv_3. Multi-target prediction involves concatenation of all feature vectors, and in this case that would be $fv_1 + fv_2 + fv_3$ (with $+$ indicating concatenation), which should lead to a prediction of $1, 2, 3$. However, standard MIMO-networks would have to separately learn that order is irrelevant for the target output. That is, that the feature-

vector $fv_2 + fv_1 + fv_3$ should lead to the prediction $2, 1, 3$. That is exactly what permutation-equivariance means, and as such we use the Deep Set architecture from [47] for this experiment.
The following Lemma, quoted from [47], shows how permutation-equivariance can be achieved in a neural-network layer, with Θ being the weights of the layer:

Lemma 1. *"The function $f_\Theta(x) = \sigma(\Theta x)$ for $\Theta \in \mathbb{R}^{M \times M}$ is permutation equivariant, iff all the off-diagonal elements of Θ are tied together and all the diagonal elements are equal as well. That is,*

$$\Theta = \lambda \mathbf{I} + \gamma(\mathbf{1}\mathbf{1}^T) \qquad \lambda, \gamma \in \mathbb{R} \quad \mathbf{1} = [1, \ldots, 1]^T \in \mathcal{R}^M$$

."

given that $x \in \mathcal{R}^N$. If $x \in R^{N \times M}, y \in R^{N \times M'}$ holds, matrix multiplication can be used instead to yield (quoted from [47]):

$$f(x) = \sigma(\Lambda - \mathbf{1}\mathbf{1}^T \Gamma) \tag{5.3}$$

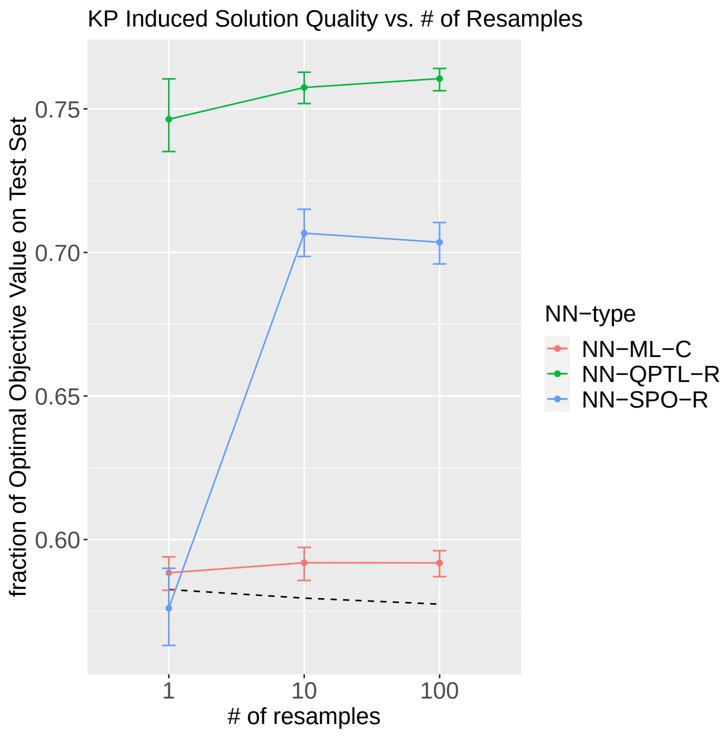
This is the case in our setting; we have N item-samples in one problem-sample, with M features, for which we expect N different single-valued outputs $M' = 1$. Note that both the unweighted KP-problem and the BM-problem are permutation-equivariant with respect to their objective parameters. Namely, the BM-problem is permutation-equivariant with respect to each side of the matching problem. As such we evaluate the MIMO-networks only in the BM(U) setting for the BM-problem. To our knowledge this particular type of multi-output network has not yet been applied to Predict-and-Optimize settings in particular, but the general idea of permutation-equivariant prediction making obviously comes from Zaheer et al. [47].

5.3 Results

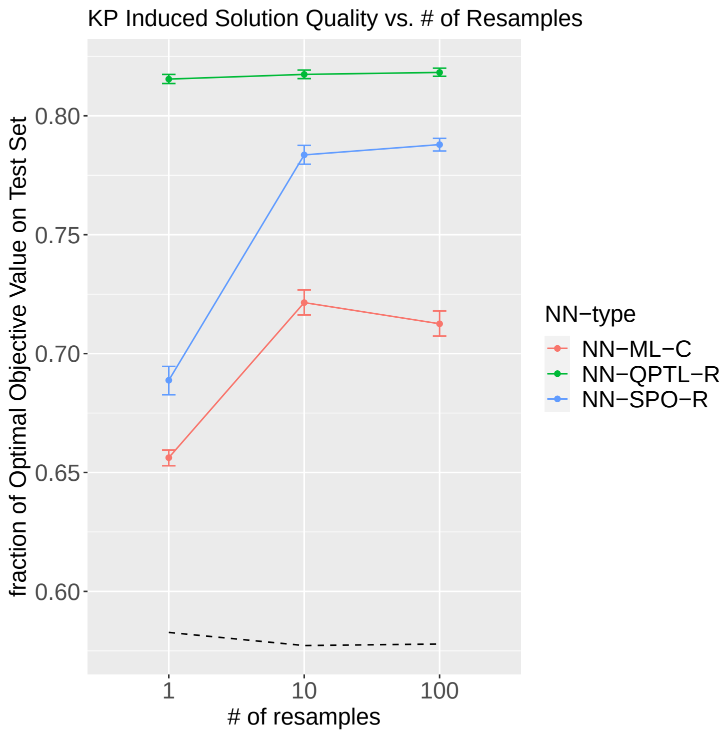
All results are shown with 95% bootstrapped confidence intervals around the mean.

5.3.1 KP domain

Figure 5.1 shows the performance of the neural networks for 10 and 100 samples per class in the knapsack domain, using standard neural networks. Directing our attention to Figure 5.1a, when the number of resamples is increased, little to no benefit is seen for NN-ML-C. In fact its performance is only slightly better than inputting random values as objective parameter of the problem samples. NN-QPTL-R also sees little performance increase when the number of resamples is increased; but it performs by far the best right from the start (NN-QPTL-B (not shown) always performed more poorly than NN-QPTL-R), corroborating the sample-efficiency noticed by [45]. NN-SPO-R, however, does see a large increase in performance of roughly 17 percentage-points when the number of resamples is increased from one to 10, after which it sees no further increase in performance. This seems to imply that repeatedly constructing problem-samples from the same pool of item-samples can indeed improve performance of PnO approaches, but that this does not infinitely extend. Moving on to Figure 5.1b, where the number of samples per class is increased to 100, we see a somewhat similar trend occur. Here NN-ML-C does see improvements in performance as the number of resamples increases to 10. We suspect that this is because increasing the number of problem-samples can



(a) 10 samples per class

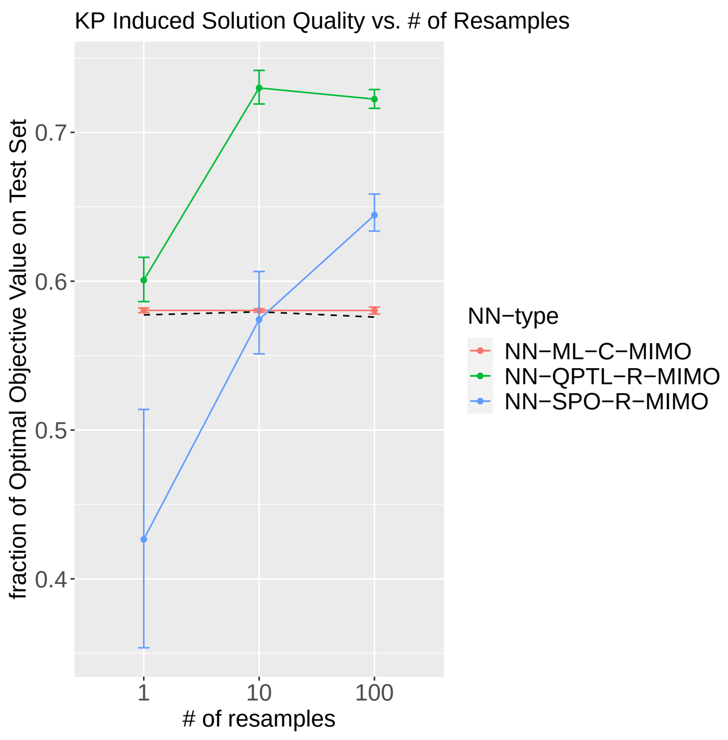


(b) 100 samples per class

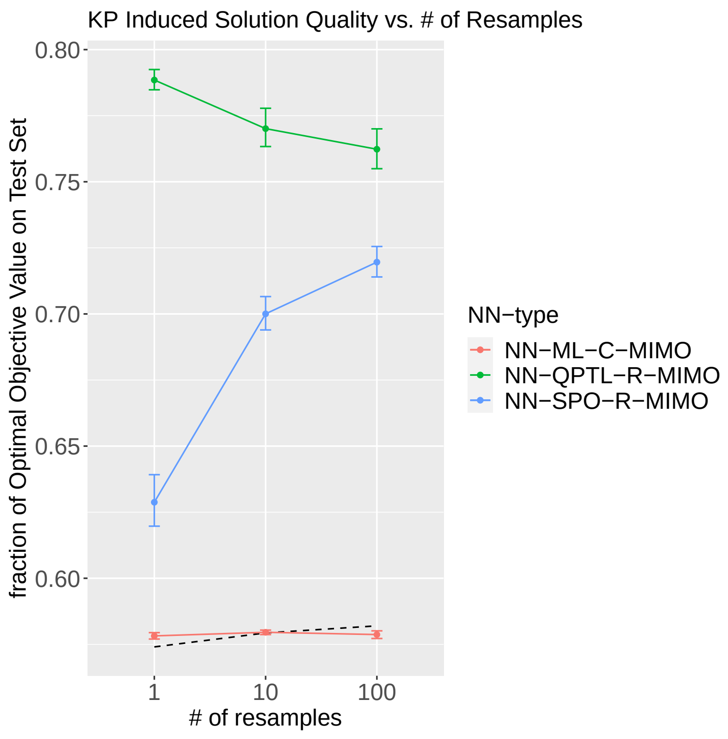
Figure 5.1: PnO with problem-resampling in KP-domain

also be seen as increasing the number of epochs; after all, the problem-samples that are resampled only contain item-samples that have already been seen. The reason why this increase was not seen in Figure 5.1a was because the small number of samples did not contain enough information. We tested if NN-SPO-R would see an increase in performance for $r = 1, n = 10$ if the number of epochs was increased to 1000 (as increasing the number of resamples can be seen as training for 10 times more epochs); this was not the case. We still see no increase in performance for NN-QPTL-R across the number of resamples but it still performs best overall. NN-SPO-R does see an increase in performance as the number of resamples increases from 1 to 10, albeit not as drastic as the jump seen in Figure 5.1a.

In addition it is interesting to see that back in Figure 4.3, NN-ML-C completely dominated the other networks at this level of noise. Clearly the lower sample size makes it much harder to predict accurately, while the PnO networks are much less perturbed. Even more interesting is how well QPTL performs when contrasted with its performance in Figure 4.3. At only 100 samples per class, it performs nearly as well as it did there.



(a) 10 samples per class



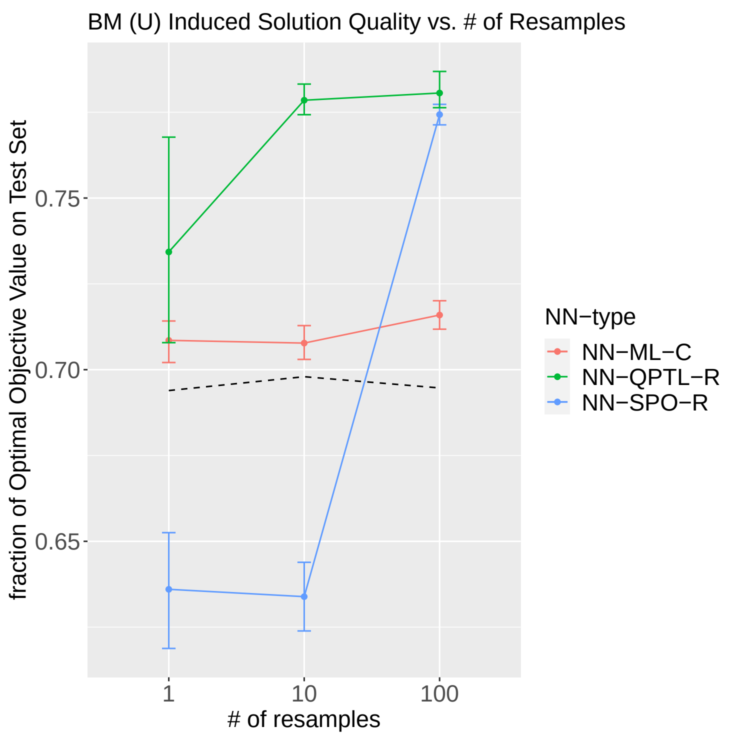
(b) 100 samples per class

Figure 5.2: PnO with problem-resampling in KP-domain - MIMO

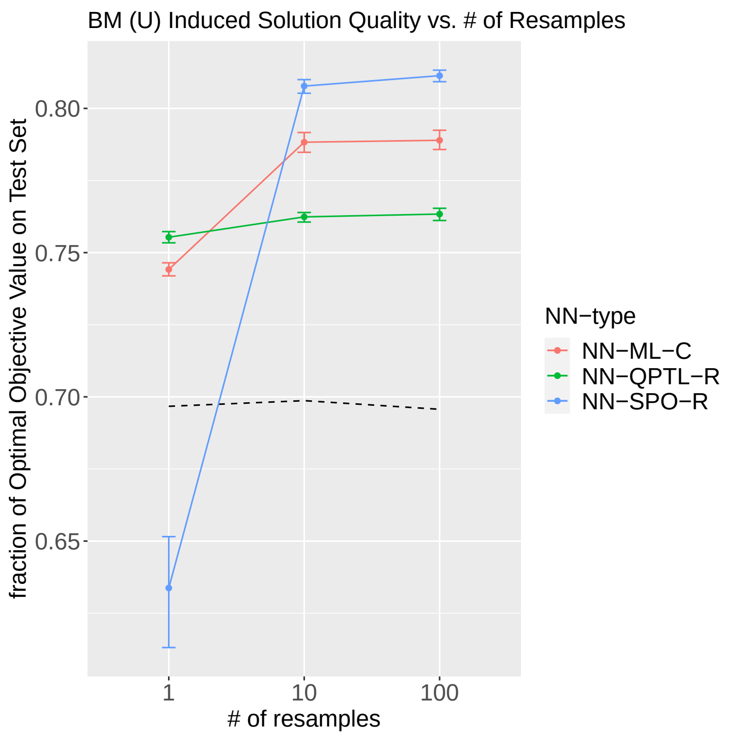
Now we evaluate the performance of the MIMO networks in the knapsack domain, seen in Figure 5.2. Note that we see marked decreases in overall performance across the majority of the networks, with NN-SPO-R-MIMO performing worse than random when the number of resamples is lower. This can occur when many negative values are predicted as part of the objective parameter of problem samples, causing the solver to pick no item-samples or very few item-samples. In Figure 5.2a, we see that NN-SPO-R-MIMO performs in a clear pattern; it performs (very) poorly for a low number of resamples and gradually starts performing better; moving from 1 to 10 resamples it sees an increase of 0.15 percentage-points, and a further increase of 0.07 moving from 10 to 100 resamples. This again seems to confirm the hypothesis that utilizing repeats can improve performance of a PnO approach. The NN-QPTL-R-MIMO network shows a large improvement of roughly 0.125 percentage-points as the number of resamples is increased from 1 to 10, after which no further improvement is seen in moving from 10 to 100 resamples. The NN-ML-C-MIMO network is seemingly completely unaffected by increasing the number of resamples. As we increase the number of samples from 10 to 100, we see that these trends still hold for SPO in Figure 5.2b, albeit to a lesser extent. Funnily enough, the performance of NN-QPTL-R-MIMO actually decreases. All neural networks seem to be relatively unaffected by increasing the number of resamples.

5.3.2 BM domain

In Figure 5.3a, we see improvements in terms of performance as the number of resamples increases for NN-QPTL-R, and NN-SPO-R. NN-QPTL-R sees an increase of roughly 6 percentage-points moving from 1 to 10 resamples, although the initial confidence interval is very large. NN-SPO-R performs very poorly initially, but then skyrockets in performance as the number of resamples is increased. NN-ML-C is nearly completely unaffected by increasing the number of resamples.
In Figure 5.4, NN-SPO-R-MIMO performs very poorly, performing below random, whereas the other two networks do not show an increase as the number of resamples is increased.
No real performance increase is seen for NN-ML-C-MIMO and NN-QPTL-R-MIMO as the number of resamples is increased. This is the case when the number of samples per class is 10 as well as 100. NN-SPO-R-MIMO shows overall poor performance.

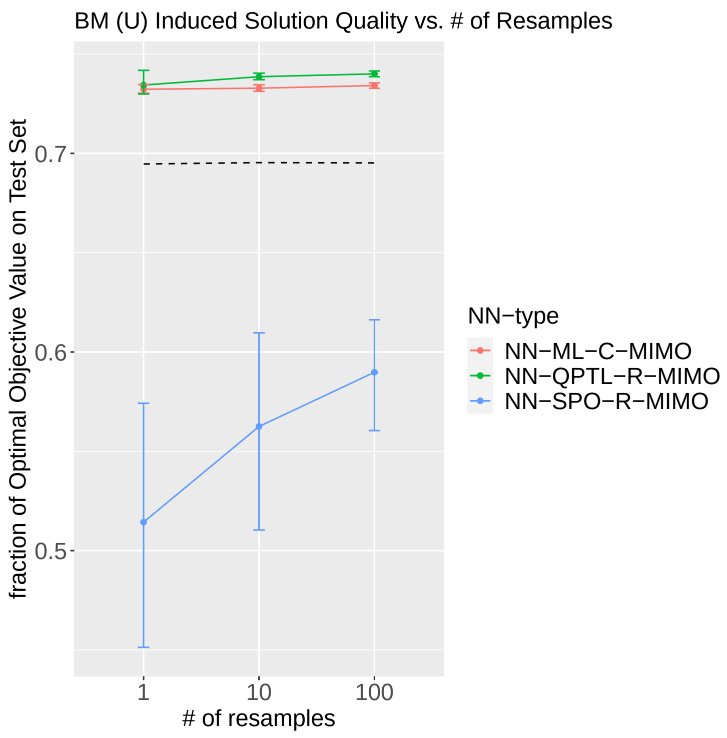


(a) 10 samples per class

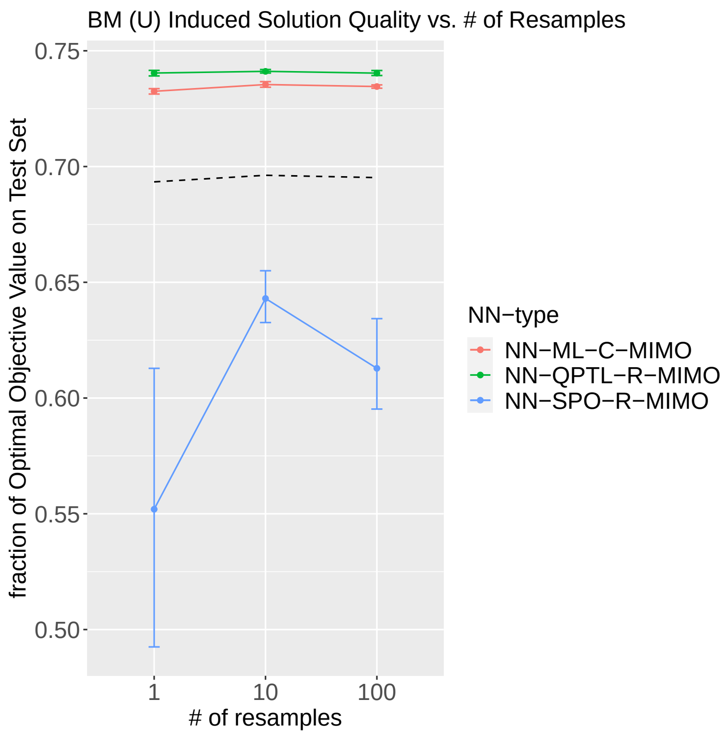


(b) 100 samples per class

Figure 5.3: PnO with problem-resampling in BM(U) domain



(a) 10 samples per class



(b) 100 samples per class

Figure 5.4: PnO with problem-resampling in BM(U) domain - MIMO

It is unclear why NN-SPO-R-MIMO shows such poor performance compared to NN-SPO-R in this domain. This may be due to the fact that the -MIMO networks in general find it harder to make good predictions compared to their non-MIMO counterparts, but then we would expect to see the same poor performance for the other networks.

We now consider the BM(C) dataset, for which only the results for 10 samples per class are shown. In Figure 5.5, we see that all networks perform near-random. NN-SPO-R starts off very poorly, and then sees performance increases as the number of samples increases - however, this only brings it up to par with random performance.

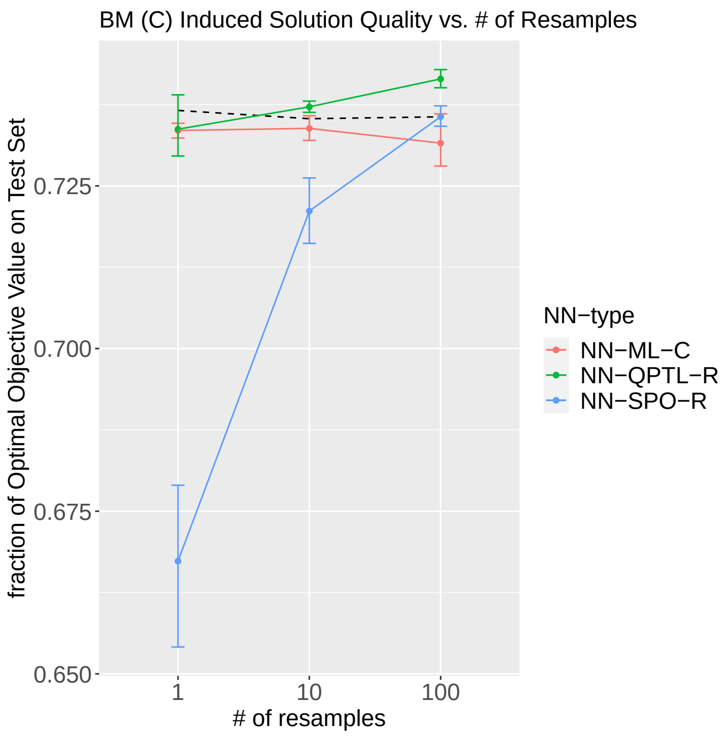


Figure 5.5: PnO with problem-resampling in BM(C) domain (10 samples per class)

5.4 Discussion

The beneficial effects of problem-sample resampling primarily show themselves in Figure 5.1a (which shows NN-SPO-R showing a large performance improvement when the number of resamples increases from 1 to 10), Figure 5.2a (which shows both NN-QPTL-R-MIMO and NN-SPO-R-MIMO showing performance improvements as the number of resamples is increased, while NN-ML-C-MIMO sees none) and Figure 5.3 (where NN-SPO-R sees large increases in performance in both figures). It is particularly interesting to link Figure 5.1a back to Figure 4.3. Here, at a standard deviation of noise level of 200, NN-QPTL-R learnt to make predictions that, when used as objective parameter for the problem-samples in the test set, induced solutions with a mean objective value of 82.5% of the optimal objective value. With only 100 samples total, however, its predictions already induced solutions with a mean objective value nearing 76% (as seen in Figure 4.3)a. For 1000 samples total, it reached nearly 82.5% This implies that, while there are may not be enough samples to learn to make good predictions using conventional loss functions (as is evident from the relatively poorer performance of NN-ML-C), there are enough samples to learn to make predictions that induce decent solutions using a MIMO network. The poor performance of problem-sample resampling in the BM-domain may be due to the fact that fewer problem-samples are created at a time. When we resample once, we create BM problems until there are no more samples left. Our BM problem-samples require 10 item-samples on each side of the bipartite graph (20 item-sample total), and thus everytime we resample from the pool, we create 5 new problem-samples.In contrast, everytime we perform problem-sample resampling in the KP-domain, we create 10 new problem-samples.

5.5 Conclusion

RQ5: *How does repeatedly using the same item-samples to construct different decision problems affect performance of PnO approaches vs traditional methods? Does the use of MIMO networks further improve performance?*

Resampling problem-samples from the same pool of item-samples can positively affect the performance of PnO approaches compared to conventional ML approaches, especially when the number of samples is low. This is particularly prominent for NN-SPO-R, which tends to start off poorly, but sees large performance increases as the number of resamples increases, at its best seeing an increase of 13 percentage-points in the KP-domain and nearly 20 percentage-points in the BM-domain. Elmachtoub and Grigas [19] shows that SPO benefits more than competitors when more labeled training data becomes available, we show that a similar trend holds when constructing more problem-samples from the same pool of labeled training data. NN-QPTL-R sees less benefit from resampling in general, but performs best overall in the majority of experiments done. While PnO approaches that utilize the MIMO-network still see performance gains as the number of resamples increases, their overall performance is lackluster compared to the non-MIMO-networks. A reason for the poor performance of the MIMO-network could have been due to the curse of dimensionality. Because the MIMO networks predict multiple item-samples at once, the number of features they take in at once has also increased. As such, they require exponentially more problem-samples for accurate prediction – each of which of course needs to be solved during training. Another reason for their poor performance may be that there is little additional benefit in seeing multiple samples at once for these simple problems: an item-sample with label 9 will always be the most valuable item, so you do not have to change its predicted value in the presence of other samples.

Conclusion 6

Predict-and-Optimize is still a relatively new field, and in this thesis we have attempted to aid development of this field by

- evaluating the effects of class imbalance, training duration increase and pre-training on the performance of the QPTL-approach relative to the baseline ML approach.
- suggesting how to adapt PnO approaches to a multi-class classification setting using neural networks.
- considering the effects of objective parameter structure on PnO approaches.
- showing the change performance in the problem-sample resampling setting.

In this chapter we list our answers to the research questions and point out research for the future.

6.1 Research Questions

RQ1: *What are the effects of class imbalance on the QPTL-framework when compared to conventional machine learning approaches?*

Resolving the class imbalance present in the Bipartite Matching experiment of [46] led to a performance increase of more than 35% in terms of solution quality for the 2L-ML network (after taking into account the performance increase obtained from the longer training duration). The reason is likely that the ML-network is able to better take into account the importance of positive samples; if no positive predictions are made in the problem-sample, no good solution value can be obtained, as the solver either selects item-samples randomly, or does not select any at all. For a severe class imbalance (in favour of samples with a negative label), the ML-network is more likely to label a particular (unknown) sample as negative, simply because it has seen more of those samples. This puts a bit of a damper on the results obtained by 2L-QPTL,as initially it more than doubled the performance of the 2L-ML network (7.40 versus 3.31), whereas now it increases performance by roughly 17% (7.40 versus 6.21). Wilder et al. [46] states that "no accuracy measure is well-correlated with solution quality". While that may be true for the other experiments performed, for the bipartite matching experiment there does seem to be a correlation between AUC and solution quality (Ferber et al. [21] also note this but do not seek a further explanation for the reason behind this). In fact, the QPTL-procedure in the bipartite matching setting could be to some extent considered as a differentiable proxy for AUC maximization, and it is likely that QPTL does not work as well on datasets that do not have a class imbalance present in the bipartite matching setting.

RQ2: *How does pre-training affect QPTL performance?*We found that pre-training causes worse performance than simply using QPTL alone. This is likely because the optimal weight configuration for an ML-network and for a QPTL-network are so different that pre-training merely manages to move the starting configuration (before training via the QPTL approach) of the network further away from the optimal QPTL-configuration. This is consistent with other findings; Ferber et al. [21] shows similarly (poor) results when it comes to pre-training and Mandi et al. [36] report no difference in performance.

RQ3: *How should one tackle multi-class classification datasets using PnO approaches, when using neural networks?* For the SPO approach it is clear from the experiments that NN-SPO-R is preferred over the other alternatives. For the QPTL-framework however, we find that all variants show similar performance. We therefore suggest to use NN-QPTL-B as default variant for QPTL; as it prevents integer overflows from occuring during solving, and is applicable to every single type of dataset (regression, binary classification, and multi-class classification). The exception is when labels of samples can be negative - because the sigmoid output activation function is constrained to the range (0,1), it might still cause the solver to pick items that worsen performance. In that case, we suggest the use of NN-QPTL-R.

RQ4: *How are PnO approaches affected when the objective parameters between problem-samples are identical versus when they are randomly selected?* We have evaluated the effects of the structure of the objective parameters on the performance of PnO networks compared to conventional ML approaches. We considered two settings, one where the objective parameters were identical across all problem-samples (setting S_I), and another were they were randomly selected (setting S_D). We considered NN-SPO-R and NN-QPTL-R as representatives of the PnO approach and NN-ML-C as representative of conventional ML approaches. We have shown that both the NN-QPTL-R network and NN-SPO-R network see significant differences in performance when comparing results from setting S_I with those from S_D. These results seem to indicate that NN-SPO-R prefers the objective parameters between problem-samples to be identical, while NN-QPTL-R sees a significant increase when objective parameters differ between problem-samples (likely because it is easier to obtain solutions of higher quality in S_D). However, the changes in performance are rather marginal.

RQ5: *How does problem-sample resampling affect the performance of PnO approaches? Does the use of a Multiple-Input Multiple-Output (MIMO) network further improve performance?*

Resampling problem-samples from the same pool of item-samples can positively affect the performance of PnO approaches compared to conventional ML approaches, especially when the number of samples is low. This is particularly prominent for NN-SPO-R, which tends to start off poorly, but sees large performance increases as the number of resamples increases, at its best seeing an increase of 13 percentage-points in the KP-domain and nearly 20 percentage-points in the BM-domain. Elmachtoub and Grigas [19] shows that SPO benefits more than competitors when more labeled training data becomes available, we show that a similar trend holds when constructing more problem-samples from the same pool of labeled training data. NN-QPTL-R sees less benefit from resampling in general, but performs best overall in the majority of experiments done. While PnO approaches that utilize the MIMO-network still see performance gains as the number of resamples increases, their overall performance is lackluster compared to the non-MIMO-networks. A reason for the poor performance of the MIMO-network could have been due to the curse of dimensionality. Because the MIMO networks predict multiple item-samples at once, the number of features they take in at once has also increased. As such, they require exponentially more problem-samples for accurate prediction – each of which of course needs to be solved during training. Another reason for their poor performance may be that there is little additional benefit in seeing multiple samples at once for these simple problems: an item-sample with label 9 will always be the most valuable item, so you do not have to change its predicted value in the presence of other samples.

6.2 Future Work

This section discusses potential areas of research for the future.

6.2.1 Reducing Training Time

The training time involved in PnO approaches setting is likely the single greatest hindrance of them all. One of the biggest reasons neural networks as an estimator started becoming popular is due to the introduction of better GPUs, allowing for more rapid training time. On the contrary, PnO approaches increase the training time by a large margin. As such, this is likely to hinder usage in real-life settings. This becomes a problem in particular when you consider an online learning setting, where new data flows in continuously, forcing the network to continuously keep training to stay up-to-date, rather than training once on a large dataset. More research like Mandi et al. [36] and Ferber et al. [21] (which considers training on smaller problem-samples and evaluating on larger problem-samples) should be performed on how this issue can be reduced.

6.2.2 Problem-sample resampling for more difficult problems

Further research can be done in the setting of Chapter 5, with more difficult problems. MIMO-networks may be more effective for such problems as well, as swapping one item-sample a contained in the problem-sample out for another item-sample b may change the desirability of all problem-samples by a huge degree. However, more difficult problems also take a longer time solve, which is difficult to combine with the fact that problem-sample resampling increases the number of problem-samples to solve.

6.2.3 Generalizability

Another interesting topic is that of the generalizability of PnO approaches. Currently a lot of research has been done where all problem-samples (both in the training as in the test set) are of the same size (i.e., they have the same number of item-samples present) and where all other parameters involved are the same. For example, Demirovic [12] explores various approaches in a PnO setting, including SPO and QPTL, using the knapsack problem as optimization problem. They explore the performance on the knapsack problem with three different capacities, one at a time.

In a more practical setting, however, we may either have knapsack problems with varying capacity in the test set or b) not know beforehand what the capacity will be of the problem-samples in the test set. Exploring how PnO approaches can be made more generalizable is likely an interesting area of research; one idea would be to simply add additional features that represent parameters of the optimization problem. However, that would imply creating problem-samples for each possible parameter value – each of which takes a long time to solve during training.

6.2.4 Prediction of constraint values

Prediction of constraint values rather than objective values has been suggested in Demirović et al. [13] and Elmachtoub and Grigas [19], but to our knowledge this has area seen no progress. As pointed out by Demirović et al. [13], this setting is more difficult, as induced solutions may no longer be feasible.

Appendix

7

7.1 Training runtime per iteration from Experiment in Chapter 4

Table 7.1: Training time per BM problem-sample listed as 'mean (standard deviation)'. Training time is averaged across entirety of training during a single experiment.

	Time/Iteration (s)
NN-ML-R	4.20E-3 Â± 7.16E-4
NN-ML-C	4.27E-3 Â± 3.52E-4
NN-QPTL-C	2.68E-2 Â± 5.11E-4
NN-QPTL-C-fixed	2.67E-2 Â± 1.05E-3
NN-QPTL-R	2.46E-2 Â± 9.49E-4
NN-QPTL-B	2.51E-2 Â± 9.12E-4
NN-SPO-C-fixed	1.60E-2 Â± 3.85E-4
NN-SPO-C	1.58E-2 Â± 4.74E-4
NN-SPO-B	1.62E-2 Â± 2.21E-3
NN-SPO-R	1.44E-2 Â± 1.29E-3

Table 7.2: Training time per KP problem-sample listed as 'mean (standard deviation)'. Training time is averaged across entirety of training during a single experiment.

	Time/Iteration (s)
NN-ML-R	3.99E-3 Â± 1.62E-3
NN-ML-C	2.67E-3 Â± 4.25E-4
NN-QPTL-C	9.79E-3 Â± 8.77E-4
NN-QPTL-C-fixed	9.78E-3 Â± 9.21E-4
NN-QPTL-R	8.15E-3 Â± 8.72E-4
NN-QPTL-B	9.67E-3 Â± 9.69E-4
NN-SPO-C-fixed	7.76E-3 Â± 8.78E-4
NN-SPO-C	7.61E-3 Â± 8.02E-4
NN-SPO-B	7.54E-3 Â± 9.12E-4
NN-SPO-R	7.48E-3 Â± 9.00E-4

Bibliography

[1]Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. „A Supervised Machine Learning Approach to Variable Branching in Branch-And-Bound". In: *Ecml* (2014) (cit. on p. 1).

[2]Brandon Amos and J Zico Kolter. „OptNet : Differentiable Optimization as a Layer in Neural Networks". In: *ICML*. Proceedings of Machine Learning Research. PMLR, 2017, pp. 136–145 (cit. on p. 13).

[3]Othman El Balghiti, Adam N. Elmachtoub, Paul Grigas, and Ambuj Tewari. „Generalization bounds in the predict-then-optimize framework". In: *Advances in Neural Information Processing Systems*. 2019. arXiv: `1905.11488` (cit. on p. 28).

[4]Richard Bellman. „Dynamic programming". In: *Science* (1966) (cit. on p. 38).

[5]Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. „Neural Combinatorial Optimization with Reinforcement Learning". In: *International Conference on Learning Representation* (2017) (cit. on p. 1).

[6]Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. „Machine Learning for Combinatorial Optimization : a Methodological Tour d ' Horizon". In: (), pp. 1–34. arXiv: `arXiv:1811.06128v1` (cit. on p. 1).

[7]Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. „A SAT-Based Version Space Algorithm for Acquiring Constraint Satisfaction Problems". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2005, pp. 23–34 (cit. on pp. 1, 10).

[8]Christian Bessiere, Abderrazak Daoudi, Emmanuel Hebrard, et al. „New Approaches to Constraint Acquisition". In: *Data Mining and Constraint Programming: foundations of a Cross-Disciplinary Approach*. Ed. by Christian Bessiere, Luc De Raedt, Lars Kotthoff, et al. Vol. 10101. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 51–76 (cit. on pp. 1, 10).

[9]Christian Bessiere, Remi Coletta, Barry O'Sullivan, and Mathias Paulin. „Query-driven constraint acquisition". In: *IJCAI International Joint Conference on Artificial Intelligence*. Hyderabad, 2007, pp. 44–49 (cit. on pp. 1, 10).

[10]Christian Bessiere, Luc De Raedt, Tias Guns, et al. „The Inductive Constraint Programming Loop". In: *IEEE Intelligent Systems* (2017) (cit. on p. 1).

[11]Christopher M. Bishop. *Pattern Recognition and Machine Learning*. 2006 (cit. on pp. 13, 18, 39).

[12]Emir Demirovic. „A Representation of the Solution Structure for Parameterised Combinatorial Optimisation and its Application to Predict + Optimise". In: () (cit. on p. 53).

[13]Emir Demirović, Peter J. Stuckey, James Bailey, et al. „An Investigation into Prediction + Optimisation for the Knapsack Problem“. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 11494 LNCS (2019), pp. 241–257 (cit. on pp. 1, 6, 7, 10, 21, 22, 30, 38, 54).

[14]Emir Demirović, Peter J Stuckey, James Bailey, et al. „Dynamic Programming for Predict + Optimise“. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. 2020 (cit. on p. 10).

[15]Emir Demirović, Peter J. Stuckey, James Bailey, et al. „Predict + Optimise with Ranking Objectives: Exhaustively Learning Linear Functions“. In: *IJCAI-19*. 2019, pp. 1078–1085 (cit. on pp. vii, 5, 6, 10, 38).

[16]Priya L. Donti, Brandon Amos, and J. Zico Kolter. „Task-based End-to-end Model Learning in Stochastic Optimization“. In: NIPS (2017) (cit. on pp. 6, 7).

[17]Paolo Dragone, Stefano Teso, and Andrea Passerini. „Constructive Preference Elicitation“. In: *Frontiers in Robotics and AI* 4.January (2018), pp. 1–16. arXiv: `1711.07875` (cit. on p. 11).

[18]Werner Dubitzky, Martin Granzow, and Daniel Berrar. *Fundamentals of data mining in genomics and proteomics*. 2007 (cit. on p. 25).

[19]Adam N. Elmachtoub and Paul Grigas. „Smart "Predict, then Optimize"“. In: (2017), pp. 1–38. arXiv: `1710.08005` (cit. on pp. 1, 6, 7, 9, 10, 21, 24, 27, 33, 37, 38, 49, 52, 54).

[20]Adam N. Elmachtoub, Jason Cheuk Nam Liang, and Ryan McNellis. „Decision Trees for Decision-Making under the Predict-then-Optimize Framework“. In: (2020). arXiv: `2003.00360` (cit. on p. 10).

[21]Aaron Ferber, Bryan Wilder, Bistra Dilkina, and Milind Tambe. „MIPaaL: Mixed Integer Program as a Layer“. In: (2019) (cit. on pp. 1, 8, 9, 14, 15, 19, 20, 22, 23, 51–53).

[22]Haibo He and Yunqian Ma. *Imbalanced learning: Foundations, algorithms, and applications*. 2013 (cit. on p. 15).

[23]Barry Hurley, Barry O'Sullivan, and Helmut Simonis. „Icon loop energy show case“. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2016 (cit. on p. 11).

[24]Barry Hurley, Lars Kotthoff, Barry O'Sullivan, and Helmut Simonis. „Icon loop health show case“. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2016 (cit. on p. 11).

[25]Aaron Courville Ian Goodfellow, Yoshua Bengio. „Deep Learning Book“. In: *Deep Learning* (2015). arXiv: `arXiv:1011.1669v3` (cit. on p. 21).

[26]Eric Jang, Shixiang Gu, and Ben Poole. „Categorical reparameterization with gumbel-softmax“. In: *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*. 2017 (cit. on pp. 22, 27, 28).

[27]David S. Johnson, Christos H. Papadimitriou, and Kenneth Steiglitz. „Combinatorial Optimization: Algorithms and Complexity.“ In: *The American Mathematical Monthly* (1984) (cit. on pp. 3, 4).

[28]Elias B. Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. „Learning to branch in mixed integer programming“. In: *30th AAAI Conference on Artificial Intelligence, AAAI 2016*. 2016 (cit. on p. 1).

[29]Jackson A. Killian, Bryan Wilder, Amit Sharma, et al. „Learning to Prescribe Interventions for Tuberculosis Patients Using Digital Adherence Data“. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. New York, NY, USA: ACM, 2019, pp. 2430–2438 (cit. on p. 8).

[30]Diederik P. Kingma and Jimmy Lei Ba. „Adam: A method for stochastic optimization“. In: *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*. 2015. arXiv: `1412.6980` (cit. on pp. 13, 40).

[31]Thomas N. Kipf and Max Welling. „Semi-supervised classification with graph convolutional networks“. In: *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*. 2017. arXiv: `1609.02907` (cit. on p. 38).

[32]Samuel Kolb, Sergey Paramonov, Tias Guns, and Luc De Raedt. „Learning constraints in spreadsheets and tabular data“. In: *Machine Learning* 106.9-10 (2017), pp. 1441–1468 (cit. on p. 10).

[33]Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. „Gradient-based learning applied to document recognition“. In: *Proceedings of the IEEE* (1998) (cit. on p. 23).

[34]Jing Liu, Fei Gao, and Jiang Zhang. „Gumbel-Softmax Optimization: A Simple General Framework for Combinatorial Optimization Problems on Graphs“. In: *Studies in Computational Intelligence*. 2020. arXiv: `1909.07018` (cit. on p. 27).

[35]Michele Lombardi, Michela Milano, and Andrea Bartolini. „Empirical decision model learning“. In: *Artificial Intelligence* 244 (2017), pp. 343–367 (cit. on p. 10).

[36]Jaynta Mandi, Emir Demirović, Peter. J Stuckey, and Tias Guns. „Smart Predict-and-Optimize for Hard Combinatorial Optimization Problems“. In: *AAAI* (2020). arXiv: `1911.10092` (cit. on pp. 1, 6, 7, 9, 15, 20, 22, 25, 37, 38, 52, 53).

[37]Mirco Nanni, Lars Kotthoff, Riccardo Guidotti, Barry O'Sullivan, and Dino Pedreschi. „ICON Loop Carpooling Show Case“. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2016, pp. 310–324 (cit. on p. 11).

[38]Prithviraj Sen, Galileo Mark Namata, Mustafa Bilgic, et al. „Collective classification in network data“. In: *AI Magazine* (2008) (cit. on pp. 13, 37).

[39]Pannaga Shivaswamy and Thorsten Joachims. „Coactive Learning“. In: *Journal of Artificial Intelligence Research* 53 (2015), pp. 1–40 (cit. on p. 11).

[40]Benoit Steiner, Zachary Devito, Soumith Chintala, et al. „PyTorch: An Imperative Style, High-Performance Deep Learning Library“. In: *Advances in Neural Information Processing Systems*. 2019 (cit. on p. 13).

[41]Stefano Teso, Andrea Passerini, and Paolo Viappiani. „Constructive Preference Elicitation by Setwise Max-margin Learning“. In: *IJCAI International Joint Conference on Artificial Intelligence* 2016-Janua (2016), pp. 2067–2073. arXiv: `1604.06020` (cit. on p. 11).

[42]Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. „Pointer networks“. In: *Advances in Neural Information Processing Systems*. 2015 (cit. on p. 1).

[43]Willem Waegeman, Krzysztof Dembczyński, and Eyke Hüllermeier. „Multi-target prediction: a unifying view on problems and methods“. In: *Data Mining and Knowledge Discovery* (2019) (cit. on p. 38).

[44]Sean Wei, Ermo; Wicke, Drew, Luke. „Hierarchical approaches for reinforcement learning in parameterized action space." In: *Proceedings of AAAI* (2018), pp. 3211–3218 (cit. on p. 27).

[45]Bryan Wilder, Eric Ewing, Bistra Dilkina, and Milind Tambe. „End to end learning and optimization on graphs". In: *Advances in Neural Information Processing Systems*. 2019. arXiv: `1905.13732` (cit. on pp. 37, 38, 41).

[46]Bryan Wilder, Bistra Dilkina, and Milind Tambe. „Melding the Data-Decisions Pipeline: Decision-Focused Learning for Combinatorial Optimization". In: *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*. 2019 (cit. on pp. vii, 1, 5, 7, 8, 10, 13, 14, 16, 18, 19, 21, 22, 24, 27, 37, 38, 51).

[47]Manzil Zaheer, Satwik Kottur, Siamak Rvanbakhsh, et al. „Deep Sets". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, {USA}*. 2017, pp. 3394–3404 (cit. on pp. 38, 40, 41).

List of Figures

List of Tables

Colophon

This thesis was typeset with $\LaTeX\,2_\varepsilon$. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.
Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.