Neural Column Generation for Capacitated Vehicle Routing

Behrouz Babaki [1, 2], **Sanjay Dominik Jena** [3, 4], **Laurent Charlin** [1, 2]

[1] HEC Montréal, [2] Mila, [3] Université de Québec à Montréal, [4] CIRRELT
behrouz.babaki@hec.ca, jena.sanjay-dominik@uqam.ca, laurent.charlin@hec.ca

Abstract

The column generation technique is essential for solving linear programs with an exponential number of variables. Many important applications such as the vehicle routing problem (VRP) now require it. However, in practice, getting column generation to converge is challenging. It often ends up adding too many columns. In this work, we frame the problem of selecting which columns to add as one of sequential decision-making. We propose a neural column generation architecture that iteratively selects columns to be added to the problem. The architecture, inspired by stabilization techniques, first predicts the optimal duals. These predictions are then used to obtain the columns to add. We show using VRP instances that in this setting several machine learning models yield good performance on the task and that our proposed architecture learned using imitation learning outperforms a modern stabilization technique.

1 Introduction

Column generation is widely regarded as an efficient technique for solving linear programming problems that have an exponential number of variables. This has allowed for solving many practically relevant large-scale linear integer programs, such as cutting stock, vehicle routing and crew scheduling, which require column generation for solving their linear relaxation.

Column generation is based on the observation that only a small subset of the variables are part of the optimal solution (i.e., have non-zero values). Therefore, in principle, it is possible to find the optimal solution using a small subset of columns. Column generation is an iterative procedure: starting with a subset of columns, at each iteration, the technique decides which column(s) to add to the problem. The procedure stops once the optimal solution is obtained.

Despite the theoretical guarantees of this procedure, in practice, it is often difficult to efficiently converge to the optimal solution. In particular, unless special care is taken, the vanilla version can add columns which do not serve the purpose of reaching the optimal solution.

Several classes of methods have been proposed for alleviating this issue, including interior-point stabilization (Rousseau, Gendreau, and Feillet 2007), which we are inspired by and compare to in our work.

We here take a different approach, which we call *neural column generation*. We formulate column generation as sequential decision making and propose a neural architecture to predict which column to add at each iteration. The model first encodes the problem (i.e., its variables and constraints) using a graphical neural network (Gasse et al. 2019). The learned representations (of the rows) are combined with instance-specific features and a differentiable optimization layer then predicts the optimal duals. This exact optimization layer guarantees the validity of the duals. In a final step, the predicted duals are decoded as a distribution over all candidate variables. The model is trained to imitate an expert that picks high-quality columns to add.

Trained on instances of the vehicle routing problem (VRP), we demonstrate that this approach outperforms a strong stabilization method (Rousseau, Gendreau, and Feillet 2007) in terms of number of columns added until convergence. Further, we validate our design choices using an ablation study and by comparing it to other neural architectures. The main contributions of this paper are:

- Formulating column selection as a sequential decision making problem, and proposing a neural architecture.
- The contributions of the architecture include a method for dealing with an exponential number of actions, ensuring the correctness of the learning-based method by plugging exact optimization layers into the deep network.
- An empirical validation of the method on VRP instances.

2 Column Generation

We will demonstrate the column generation framework using the *Capacitated Vehicle Routing Problem (CVRP)* as an example: Given the locations of N customers and one depot, the customer demands $d_i, i \in \{1, \dots, N\}$, and vehicle capacity c, find a set of routes starting and ending at the depot such that every customer is visited on some route, the sum of demands of customers on each route does not exceed the vehicle capacity, and the total travelled distance (i.e. the sum of lengths of the routes) is minimized.

This problem can be formulated as an integer linear program with an exponential number of variables (Desrochers,

Desrosiers, and Solomon 1992). Each variable in this formulation corresponds to a subset of customers $r \in \{1, \ldots, N\}$ that can form a valid route (i.e. $\sum_{i \in r} d_i \leq c$). We call each such subset a route, and denote the set of all routes by R. The cost of route r, denoted by c_r, is the length of the shortest tour that visits all customers in r and the depot. Let the binary variable x_r be 1 if route r is selected, and 0 otherwise. Further, for every route $r \in R$ and customer $i \in \{1, \ldots, N\}$, let constant e_{ir} be 1 if $i \in r$ and 0 otherwise. The CVRP can be formulated as:

$$(I): \quad \min \quad \sum_{r \in R} c_r x_r \tag{1}$$

$$\text{s.t.} \quad \sum_{r \in R} e_{ir} x_r \geq 1 \quad \forall i \in \{1, \ldots, N\} \tag{2}$$

$$\boldsymbol{x} \in \{0,1\}^{|R|}. \tag{3}$$

Constraints (2) ensure that every customer is visited at least once (even though in the optimal solution every customer is visited exactly once, this constraint is formulated as an inequality for practical reasons). In the rest of this paper, we will deal with the problem of solving the linear relaxation of this formulation (denoted by problem P) where constraint (3) is replaced by $\boldsymbol{x} \geq 0$. The integrality constraints can then be enforced through a branch-and-bound procedure.

The exponential size of R makes it impossible to explicitly represent this formulation beyond certain number of routes. Column generation is a procedure that iteratively adds promising columns (i.e. routes in CVRP) until the problem is solved to optimality. For ease of exposition, we will describe the working principle of column generation using the dual formulation of problem P:

$$(D): \quad \max \quad \sum_{i=1}^{N} \lambda_i \tag{4}$$

$$\text{s.t.} \quad \sum_{i=1}^{N} e_{ir} \lambda_i \leq c_r \quad \forall r \in R \tag{5}$$

$$\boldsymbol{\lambda} \geq 0. \tag{6}$$

In the dual formulation, variable λ_i represents the marginal cost of serving customer i. Constraints (5) ensure that for every route r, the sum of marginal costs of the customers in that route does not exceed the cost of that route. The column generation process is equivalent to generating lazy cuts in the dual formulation. At each iteration, only a subset of the constraints (5) (i.e. those corresponding to the routes $R' \in R$) is included in the dual formulation. After solving this restricted problem (which we will call problem D'), we verify if any of the excluded constrains are violated by the obtained solution. If this is the case, (some of) the violated constraints are added to the formulation and the process is repeated. Otherwise, the problem has been solved to optimality.

Finding the most violated constraint in the dual formulation requires exploring the exponential space of all possible routes. This can be formulated as a combinatorial optimization problem, called the *pricing* problem:

$$\hat{r} = \arg\min_{r \in R} (c_r - \sum_{i \in r} \hat{\lambda}_i). \tag{7}$$

where $\hat{\boldsymbol{\lambda}}$ is an optimal solution of D'. The degree of violation of the constraint corresponding to route r (i.e. $c_r - \sum_{i \in r} \hat{\lambda}_i$) in the dual formulation is equal to the reduced cost of the column corresponding to route r in the primal formulation. In other words, the pricing problem finds the column with the most negative reduced cost at each iteration of the column generation algorithm.

Problem D' can have more than one optimal solution, and the choice of this solution can affect the route obtained by the pricing problem. This, in turn, can significantly influence the speed at which the column generation algorithm converges. Consider the example presented in Figure 1, where $R' = \big\{\{1,2\},\{3,4\}\big\}$. Given the optimal solution $\boldsymbol{\lambda'} = (10, 20, 20, 10)$, the route $\{2,3\}$ has a reduced cost of -5 and might be added to D', even though it is not likely to be selected in the optimal solution of P. However, the alternative optimal solution $\boldsymbol{\lambda''} = (15, 15, 15, 15)$ yields a positive reduced cost of 5 for this route and does not add it to D'.

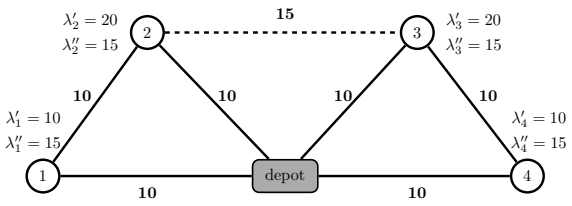

Figure 1: The route (*depot* $\rightarrow$ 2 $\rightarrow$ 3 $\rightarrow$ *depot*) has reduced costs -5 with $\boldsymbol{\lambda'}$ and 5 with $\boldsymbol{\lambda''}$. Figure adapted from Rousseau, Gendreau, and Feillet (2007).

2.1 Stabilization

The importance of choosing the right dual solution and its effect on the convergence of the column generation algorithm has been widely acknowledged, and different classes of techniques have been developed to address this problem. For a recent discussion on this topic, see Pessoa et al. (2018) and references therein.

Our work is particularly inspired by the stabilization method proposed by Rousseau, Gendreau, and Feillet (2007). The authors directly tackle the issue that the restricted dual problem tends to have several optimal solutions, each of which may distribute differently the marginal costs among the customers. Given that these extreme solutions are sparse by definition, some customers may have large dual values, while others may have zero dual values. This, in turn, may result in a bad estimation of the marginal costs of the customers. Instead of using a set of dual values proposed by one of the optimal solutions to the restricted dual problem, the stabilization method proposed by the authors uses an interior point of the convex hull of the extreme

solutions to the restricted dual by averaging over a few of them.

In this work, we follow the above intuition. However, instead of averaging over a set of randomly chosen extreme solutions, we aim at learning how to identify the extreme point (represented by its simplex tableau) that provides optimal convergence within the column generation procedure.

3 Neural Column Generation

The column generation procedure can be framed as a sequential decision making problem. At each iteration, we are faced with the task of choosing one of the columns to be included in the restricted problem. Using a collection of problem instances, we aim at learning a *policy* that dictates which column to pick. We formulate this problem as a *Contextual Markov Decision Process*.

A *Markov Decision Process (MDP)* is a tuple $\langle \mathcal{S}, \mathcal{A}, p(s' \mid s, a), r(s, a), p_0 \rangle$ where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $p(s' \mid s, a)$ is the transition probability ($s', s \in \mathcal{S}, a \in \mathcal{A}$), $r(s, a)$ is the reward function, and p_0 is the initial state distribution.

A contextual Markov decision process is a family of MDPs parameterized by a context z, where the state space, action space, reward function and transition probabilities depend on z (Sonnerat et al. 2021). We define z as the properties of a CVRP instance (vehicle capacity, demands and locations of customers, etc.). The state s_t at iteration t of the column generation procedure consists of all the information presented in the final simplex tableau in that iteration[1]. The most important components of the state are the columns (routes), available within the restricted problem, and the dual values (i.e. the optimal solution of the restricted dual problem). The action a_t is the next column to add to the restricted problem. The transition function $p(s' \mid s, a)$ is deterministic and is evaluated by adding the column corresponding to action a to the restricted problem encoded in s and solving it. Since our goal is to improve the convergence rate, we consider a constant reward of -1. The process terminates when there is no column with a negative reduced cost.

A policy $\pi_\theta(a_t \mid s_t, z)$ is a probability distribution over actions, conditioned on the state and context, and parameterized by a vector θ. Our goal is to learn this distribution, that is, find the value for θ that maximizes the expected reward across different contexts. We learn this policy by *imitating* an *expert*. This expert is assumed to know how to select columns in a way that leads to fast convergence of the column generation algorithm, but is potentially too expensive to be used directly. We record the actions of the expert by solving a collection of N problem instances using the expert policy. For every instance i, we record the context $z^{(i)}$. Moreover, at every iteration t, we record the state ${s_t}^{(i)}$ and the action $a_t^{*(i)}$ advised by the expert. We then learn the policy by minimizing the cross-entropy loss:

$$\mathcal{L}(\theta) = -\sum_{i=1}^{N} \sum_{t=1}^{T_i} \log \pi_\theta(a_t^{*(i)} \mid {s_t}^{(i)}, z^{(i)}) \tag{8}$$

Minimizing this loss function is equivalent to maximizing the likelihood of the expert actions.

3.1 Expert Policy

Our proposed method requires an expert that can choose the right columns to make the column generation procedure converge quickly. We will now present such an expert policy. Recall that at each iteration, the vector of dual values $\boldsymbol{\lambda'}$ (the optimal solution of D' obtained by the simplex algorithm) is an *estimate* of $\boldsymbol{\lambda}^*$, the optimal solution of D. Motivated by this observation, from the space of optimal solutions of D' we pick the $\boldsymbol{\lambda'}$ that is closest to $\boldsymbol{\lambda}^*$:

$$\min_{\boldsymbol{\lambda'}} \quad \| \boldsymbol{\lambda'} - \boldsymbol{\lambda}^* \|_2^2 \tag{9}$$

$$\text{s.t.} \quad \sum_{i=1}^{N} \lambda'_i = \Lambda \tag{10}$$

$$\sum_{i=1}^{N} \lambda'_i e_{ir} \leq c_i \quad \forall r \in R' \tag{11}$$

$$\boldsymbol{\lambda'} \geq 0 \tag{12}$$

where Λ is the optimal objective value of D', and constraint (10) enforces the optimality of the solution. After solving this problem, we provide these *adjusted* duals to the pricer, and obtain a route a^* which is used as the expert action at this state.

Note that the expert requires access to $\boldsymbol{\lambda}^*$, which means that we needs to first solve the instance to optimality. After solving a collection of instances offline and collecting the expert actions, we can train the policy function $\pi_\theta(a_t \mid s_t, z)$ which is expressed as a deep neural network.

3.2 Policy Network

The policy function takes the context (a CVRP instance) and the state (the final simplex tableau in an iteration) as input, and returns a distribution over all columns (all feasible routes). Expressing this function as a deep network raises a number of challenges. First, this distribution is defined over an exponential number of columns which can be enumerated only for small instances. We should be able to use this distribution without the need to explicitly represent it. Second, the size of the context and state varies by instance and iteration. Hence this function should accept variable-sized inputs. Finally, this function should be aware of certain symmetries in the input. For example, permuting the columns of the simplex tableau should not change the output, and after swapping two constraints in the tableau the corresponding output probabilities should be swapped.

A Distribution over Actions We address the first challenge by learning a mapping from s_t and z to a vector of adjusted duals that is optimal wrt the restricted dual corresponding to s_t. After learning this mapping, we use these

[1] Alternatively, we can formulate the task as an MDP where both pieces of information are encoded in the state

adjusted duals in the column generation procedure. At each iteration, we provide s_t and z to this mapping and obtain the adjusted duals. the pricer then takes the adjusted duals as input and returns a column. The procedure terminates when the reduced cost of the returned column is non-negative. Note that the optimality of adjusted duals is essential for the soundness of this approach.

Let us assume for now that there is a function f_θ that maps s_t and z to the optimal duals $\boldsymbol{\lambda}$. We can turn this function into a distribution over all columns (and the termination action) and train it using the loss function of Equation 8. We do this using instances for which the columns can be enumerated. For an instance with n columns, let us represent column i and its cost by $\boldsymbol{e_i}$ and c_i, respectively. Given the adjusted duals $\boldsymbol{\lambda}$, the reduced cost of column i is $c_i - \boldsymbol{\lambda e_i}$.

We denote the action of choosing column i by $\boldsymbol{e_i}$ $(i = 1, \ldots, n)$, and choosing no column (i.e. terminating the procedure) by a_0. Our goal is to define a probability distribution over $a_0, a_1, \ldots, a_n$ that depends on $\boldsymbol{\lambda}$ and gives higher probability to columns with smaller reduced cost. More formally, we require that for every pair of columns i and j, $P(a_i) \geq P(a_j)$ iff $c_i - \boldsymbol{\lambda e_i} \leq c_j - \boldsymbol{\lambda e_j}$. Moreover, the action a_0 should have the highest probability only if there is no column with a negative reduced cost. We enforce this by requiring that for every column i, $P(a_0) \geq P(a_i)$ iff $c_i - \boldsymbol{\lambda e_i} \geq 0$. These requirements are met by the following distribution:

$$P(a_0) = \frac{1}{1 + \sum_{j=1}^{n} \exp(\boldsymbol{\lambda e_j} - c_j)} \tag{13}$$

$$P(a_i) = \frac{\exp(\boldsymbol{\lambda e_i} - c_i)}{1 + \sum_{j=1}^{n} \exp(\boldsymbol{\lambda e_j} - c_j)} \tag{14}$$

This distribution is obtained by applying the softmax function to the vector $(0, \boldsymbol{\lambda e_1} - c_1, \ldots, \boldsymbol{\lambda e_n} - c_n)$. Figure 2 depicts the mapping from s_t and z to this distribution. Next, we will describe the architecture of f_θ, the deep network that maps s_t and z to the optimal duals $\boldsymbol{\lambda}$.

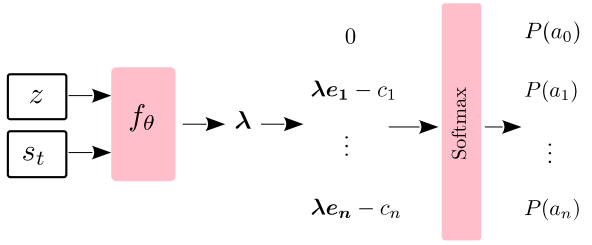

Figure 2: The deep network f_θ maps the context z and state s_t to a vector of optimal duals $\boldsymbol{\lambda}$ which is then used for defining the policy $\pi_\theta(a_t \mid s_t, z)$.

Representing the State We represent the state s_t as a bipartite graph $G = (\mathcal{C}, \mathcal{V}, \mathcal{E})$ where $\mathcal{C}$ corresponds to constraints (customers), $\mathcal{V}$ corresponds to variables/columns (routes), and $(c, v) \in E$ iff variable v has a nonzero coefficient in constraint c. Other state information correspond either to variables (e.g. cost, reduced cost) or constraints (e.g. slack, dual value) and can be represented as features of the nodes in $\mathcal{V}$ and $\mathcal{C}$.

After encoding the state as a bipartite graph $G = (\mathcal{C}, \mathcal{V}, \mathcal{E})$ with node features $c_i \in \mathbb{R}^{d_\mathcal{C}}, i \in \{1, \ldots, |\mathcal{C}|\}$ and $v_i \in \mathbb{R}^{d_\mathcal{V}}, i \in \{1, \ldots, |\mathcal{V}|\}$, we can apply the graph convolutional layers on this graph and obtain the node embeddings $c'_i \in \mathbb{R}^{d'_\mathcal{C}}, i \in \{1, \ldots, |\mathcal{C}|\}$ and $v_i \in \mathbb{R}^{d'_\mathcal{V}}, i \in \{1, \ldots, |\mathcal{V}|\}$. We use a graph convolutional layer similar to the one introduced by Gasse et al. (2019) to process this bipartite graph:

$$c'_i = f_\mathcal{C}\big(c_i \parallel \sum_{j:(i,j)\in E} g_\mathcal{C}(c_i \parallel v_j)\big) \tag{15}$$

$$v'_i = f_\mathcal{V}\big(v_j \parallel \sum_{i:(i,j)\in E} g_\mathcal{V}(c'_i \parallel v_j)\big) \tag{16}$$

where $\parallel$ is the concatenation operator and $f_\mathcal{C}$, $f_\mathcal{V}$, $g_\mathcal{C}$ and $g_\mathcal{V}$ are 2-layer perceptrons with ReLU activation functions.

By representing the state as a graph, states with different sizes can be processed by the same model. Moreover, the desired permutation invariance and equivariance properties are maintained by the model.

Representing the Context We include the context z in the model by adding extra information to the constraint embeddings c'_i. Let us represent the properties of customer i (e.g. location and demand) by the vector $\hat{c}_i$. After concatenating c'_i and $\hat{c}_i$ we obtain the feature vector c''_i. We can also include global information (e.g. the vehicle capacity) in this vector. We will then apply the set encoder layers to the set of constraint features and obtain another set of features $\bar{c}_i, i \in \{1, \ldots, |\mathcal{C}|\}$.

Self-Attention (Vaswani et al. 2017) is a mechanism that maintains permutation equivariance, and has been used in models that operate on sets (Lee et al. 2019) or graphs (Velickovic et al. 2018). An attention layer takes n elements $x_i \in \mathbb{R}^{d_x}, i \in \{1, \ldots, n\}$ and outputs $z_i \in \mathbb{R}^{d_z}, i \in \{1, \ldots, n\}$, where each output is a weighted sum of linear transformations of the input elements:

$$z_i = \sum_{j=1}^{n} \alpha_{ij}(x_j W^V) \tag{17}$$

where the weights are computed in terms of compatibility scores e_{ij}:

$$\alpha_{ij} = \frac{\exp e_{ij}}{\sum_{k=1}^{n} \exp e_{ik}} \tag{18}$$

and the compatibility scores compare two input elements using a scaled dot product function:

$$e_{ij} = \frac{(x_i W^Q)(x_j W^K)^T}{\sqrt{d_z}} \tag{19}$$

$W^Q, W^K, W^V \in \mathbb{R}^{d_x \times d_z}$ are learnable parameters. Similar to Transformers, we create encoder layers which consist of a self-attention and a position-wise feedforward layer, where each of these sublayers is accompanied by residual connections and is followed by layer normalization.

Generating the Optimal Duals Finally, we need to map the feature vectors $\bar{c}_i, i \in \{1, \ldots, |\mathcal{C}|\}$ to dual values λ_i that are optimal wrt the restricted dual. The space of optimal duals can be represented by a set of linear constraints consisting of the restricted dual and an optimality constraint, similar to constraints (10-12).

We create a mapping from $\bar{c}_i$ vectors to an objective function over these constraints. By solving the resulting optimization problem we obtain a vector of duals which is a function of $\bar{c}_i$ vectors, and also optimal wrt the restricted dual problem. Assuming that $\bar{c}_i \in \mathbb{R}^k$ and denoting the jth element of $\bar{c}_i$ by $\bar{c}_{ij}$, we construct the vector $\boldsymbol{q}$ and matrix $\boldsymbol{Q}$ as follows:

$$\boldsymbol{q} = (\bar{c}_{11}, \ldots, \bar{c}_{|\mathcal{C}|1}) \tag{20}$$

$$\boldsymbol{P} = \begin{bmatrix} \bar{c}_{12} & \ldots & \bar{c}_{|\mathcal{C}|2} \\ \ldots & \ldots & \ldots \\ \bar{c}_{1k} & \ldots & \bar{c}_{|\mathcal{C}|k} \end{bmatrix} \tag{21}$$

$$\boldsymbol{Q} = \boldsymbol{P}^T \boldsymbol{P} \tag{22}$$

We then define $\frac{1}{2}\boldsymbol{\lambda}^T \boldsymbol{Q} \boldsymbol{\lambda} + \boldsymbol{q}^T \boldsymbol{\lambda}$ as the objective function over the space of optimal duals, and create a quadratic optimization problem. This optimization problem is differentiable wrt $\boldsymbol{q}$ and $\boldsymbol{Q}$, and can be included as a layer in a deep network.

Differentiable optimization layers are computational units that not only solve an optimization problem, but also calculate the gradient of its solution with respect to the problem parameters. Such layers make it possible to include an optimization step as part of a deep learning pipeline. We will use the OptNet architecture of Amos and Kolter (2017) which solves Quadratic Programming (QP) problems of the form:

$$\min_z \quad \frac{1}{2} z^T Q z + q^T z \tag{23}$$

$$\text{s.t.} \quad Az = b \tag{24}$$

$$Gz \leq h \tag{25}$$

Assume that in the backward pass of the backpropagation algorithm, we receive the vector $\frac{\partial \ell}{\partial z^*} \in \mathbb{R}^n$, $D(.)$ creates a diagonal matrix from a vector, and z^*, ν^* and λ^* are the optimal primal and dual variables. OptNet first calculates:

$$\begin{bmatrix} d_z \\ d_\lambda \\ d_\nu \end{bmatrix} = - \begin{bmatrix} Q & G^T D(\lambda^*) & A^T \\ G & D(Gz^* - h) & 0 \\ A & 0 & 0 \end{bmatrix}^{-1} \begin{bmatrix} (\frac{\partial \ell}{\partial z^*})^T \\ 0 \\ 0 \end{bmatrix} \tag{26}$$

using which then the gradients with respect to all problem parameters are calculated. In this work, we will use the gradients with respect to the coefficients of the objective function (i.e. Q and q):

$$\nabla_Q \ell = \frac{1}{2}(d_z z^T + z d_z^T) \qquad \nabla_q \ell = d_z \tag{27}$$

The coefficient matrices G in our examples are not always full-rank, and this makes it impossible to perform the matrix inversion in Equation 26. Hence we replace the inversion opertion with pseudo-inversion. Figure 3 summarizes the architecture of the policy network.

4 Experiments

In this section we show the effectiveness of the proposed approach by performing an empirical evaluation. We investigate four research questions in our experiments: **Q1** To what extent the learned models improve the convergence of the column generation algorithm?, **Q2** How do different architectures compare in terms of loss and accuracy?, and **Q3** To what extent removing the differentiable optimization layer affects the performance of the learned models?

We train and evaluate our approach using randomly-generated CVRP instances. We generate the instances using the method proposed in (Uchoa et al. 2017) with 21 customers, a *central* depot, *clustered* customer positioning, and a value of 6 for the r parameter (the desired average number of customers in a route). In training, we need all valid routes for an instance. Inspired by the dynamic programming algorithm for solving the TSP, we developed algorithm 1 for generating all valid routes for a CVRP instance.

Algorithm 1: Generating all routes for CVRP

```
function GENERATEROUTES(n, distances, demands, capacity)
    for i ← 1, ..., n do
        path_lenght[{i}, i] ← distances[0, i]
        Q.push[{i}]
        routes ← routes ∪ {i}
    while !Q.empty() do
        path ← Q.pop()
        d ← Σ_{i∈path} demands[i]
        for i ← max(path) + 1 ... n do
            if demands[i] + d ≤ capacity then
                next_path ← path \ {i}
                Q.push(next_path)
                routes ← routes ∪ {next_path}
                for u ∈ next_path do
                    prefix ← next_path \ {u}
                    path_length[next_path, u] ←
                      min_{v∈prefix} (path_length[prefix, v] +
                      distances[v, u])
    for path ∈ routes do
        route_lengths[path] ←
          min_{v∈path} path_lengths[path, v] + distances[v, 0])
    return route_lenghts
```

We solve 350 training and 350 validation CVRPs using the expert policy described in section 3.1. Each training/validation instance corresponds to one iteration executed according to the expert policy. For testing, we solve 1500 CVRP instances using different policies and recorded the number of iterations. Since the IPS method is randomized and the number of iterations can vary significantly, for this method we take the average over 20 runs.

We train all models using Adam (Kingma and Ba 2015) with an initial learning rate of $3 * 1e-4$ and minibatches of

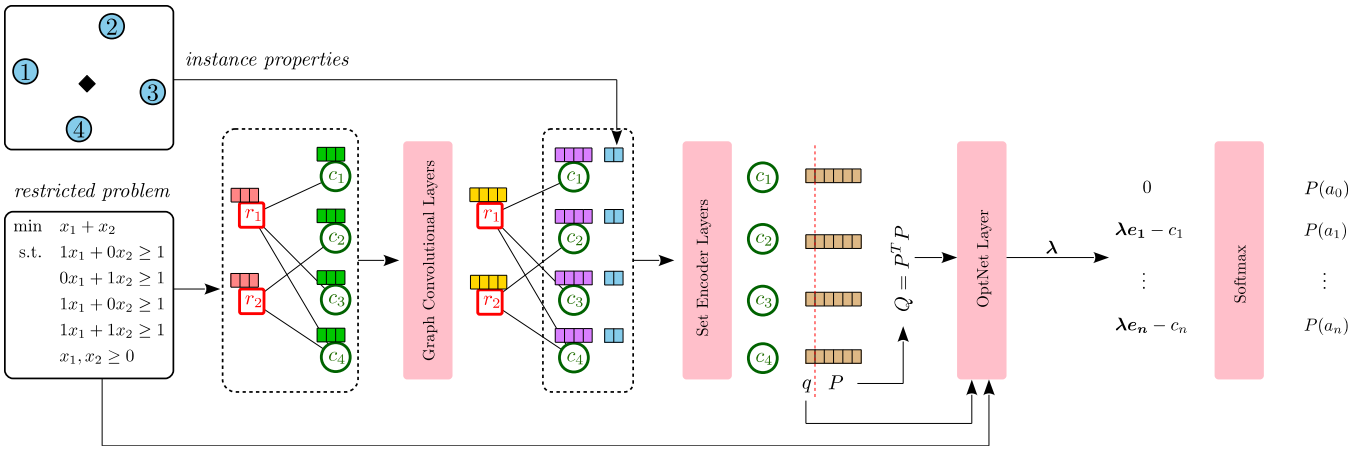

Figure 3: The architecture of policy network in neural column generation. The nodes r_i and c_j correspond to the routes and customers (i.e. the columns and rows in the restricted problem), respectively.

size 32. After 10 epochs with no validation error improvement the rate is divided by 5. The training stops after 20 such epochs or after 12 hours of training, whichever happens first. We train the models using a single thread of CPU using machines with Intel 6148 2.4 GHz processors with a memory limit of 32 GB. The code and data used in these experiments is publicly available online[2].

To answer **Q1**, we compare the convergence of several learned models with the Interior-point Stabilization (**IPS**) method of Rousseau, Gendreau, and Feillet (2007). In order to simplify the training, we assume that $\boldsymbol{Q}$ in the OptNet objective is fixed to $0.001\boldsymbol{I}$ and only learn the linear coefficients $\boldsymbol{q}$. These coefficients are then fed into OptNet which produces the probability distribution over the columns (the last part of Figure 3). We compare the following methods for learning $\boldsymbol{q}$ in our empirical evaluation:

- **Baseline**: learning a fixed $\boldsymbol{q}$ for all training instances, discarding the state and context information.
- **MLP**: a model with a row-wise two-layer feed-forward network, applied independently to each customer i with q_i as the output.
- **SE**: a set encoder with customer information as input and $\boldsymbol{q}$ as output.
- **GCN**: a graph neural network with state information as input, and $\boldsymbol{q}$ as output.
- **GCN+SE**: using both graph convolutional and set encoder layers, as depicted in Figure 3.

For the route nodes (used in the models **GCN** and **GCN+SE**), we used a single feature, namely the cost of the route. Initially, the customer nodes have a single feature in the **GCN+SE** model, which is the original dual value given by the simplex tableau. After receiving the embeddings of customer nodes from the graph convolutional layers, they are concatenated with the demands and locations of customers, and the vehicle capacity (appended to the feature vectors of all customers). In models **Baseline**, **MLP**, **SE** and **GCN**, all customer features are provided in one feature vector.

[2]https://github.com/Behrouz-Babaki/NCG4CVRP

	IPS	*Baseline*	*MLP*	*SE*	*GCN*	*GCN+SE*
#Wins	1	278	296	248	436	**695**
Ratio	1.581	1.239	1.211	1.232	1.17	**1.12**

Table 1: Comparing different policies in terms of number of wins and the average ratio of number of iterations with respect to the expert policy.

Table 1 compares the convergence of different methods on 1500 test instances. The first row (wins) shows the number of times that each method has the smallest number of iterations. Moreover, for each instance and method, we divide the number of iterations by that of the expert policy. The second row shows the averages of these ratios over all test instances. The results indicate that all learned models outperform **IPS**. Among the learned models, **GCN+SE** significantly dominates the other ones.

In order to answer **Q2**, in Table 2 we present the cross-entropy loss of different models on the validation set. We also present the top-k accuracy for different values of k, which is the percentage of the times that the target column appears in the k columns with the highest probability in the distribution generated by each model. The learning curves of different models are presented in Figure 4. These results clearly indicate the advantage of the model **GCN+SE** over the alternatives.

Finally, we answer **Q3** by creating models which directly predict the adjusted duals used by the expert policy. Both of these models have architectures similar to **GCN+SE**, except that the layers after set encoders are removed. In the **MSE** model, the output of set encoders is used as predicted adjusted duals. This model is trained by minimizing the Mean Squared Error (MSE) loss between the predicted and target adjusted duals. Note that the dual vector predicted by this model is not necessarily optimal. We address this problem in

	Loss	*Top-k Accuracy*			
		1	*10*	*100*	*1000*
Baseline	41.192	0.460	0.774	0.934	0.983
MLP	32.243	0.492	0.816	0.952	0.988
SE	28.271	0.519	0.840	0.961	0.989
GCN	28.007	0.521	0839	0.959	0.989
GCN+SE	**21.955**	**0.575**	**0.881**	**0.969**	**0.990**

Table 2: Cross-entropy loss and top-k accuracy for different models.

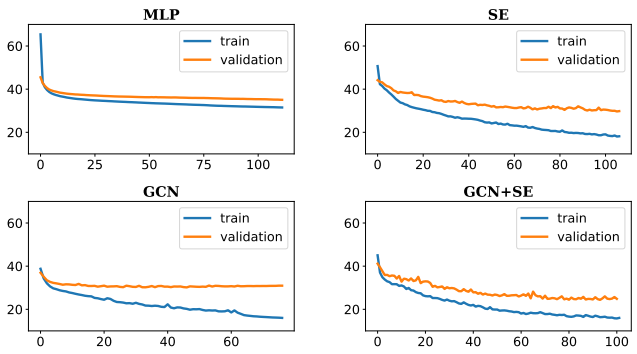

Figure 4: Loss curves for different architectures. The vertical and horizontal axes represent the cross-entropy loss and number of iterations, respectively.

the **KLD** model, which learns to distribute the optimal objective among the customers. This distribution is obtained by applying the softmax function to the output of set encoders. This model is trained by minimizing the KL-divergence between the predicted distribution and the actual distribution of total value among the target adjusted duals.

Table 3 shows the cross-entropy loss and top-k accuracy for these two models. Comparing these values with those in Table 2 shows that including the OptNet layer is essential for obtaining a reasonable performance.

5 Conclusion and Future Work

In this paper we proposed and explored several architectures for improving the convergence of the column generation algorithm using deep learning. The empirical evaluation demonstrates the advantage of encoding the instance information (using a set encoder), the routes currently included in the problem (using a bipartite graph convolution layer), and end-to-end learning (using a differentiable optimization layer).

Despite the advantages of the OptNet layer, it also turns into a bottleneck in the learning pipeline and limits the amount of training data that can be processed. First, it has to solve a QP for every data instance at every forward pass. Second, our implementation of OptNet can only use one cpu thread, which severely harms the scalability of our approach. An interesting direction for future work is removing the OptNet layer and train networks with larger capacity using more data.

In theory, column generation is presented as a method that adds one column at each iteration. In practice, the pricer returns a set of columns with small reduced cost (including the smallest) and some or all of these columns are added to the problem. Generalizing this work to such a procedure is another direction for future work.

The convergence of a column generation algorithm depends not only on the number of iterations, but also on the time taken for solving the pricing problem at each iteration. Incorporating the latter is another venue for future research. Finally, it is know that algorithmic reasoning using graph neural networks often does not generalize well to unseen graph sizes. Studying and addressing this phenomenon is another important direction for future study.

We presented neural column generation using VRP as an application. However, the main principles of our approach remain the same across many classes of problems. In order to present this approach as a generic method that is applicable to all these problem classes, we need to introduce a generic method for representing the instance properties, wich opens another direction for follow-up work.

	Loss	*Top-k Accuracy*			
		1	*10*	*100*	*1000*
MSE	139.549	0.083	0.297	0.651	0.910
KLD	167.104	0.061	0.236	0.595	0.898

Table 3: The effect of removing the differentiable optimization layer on cross-entropy loss and top-k accuracy.

Acknowledgments

We are thankful to Loius-Matrin Rosseau, Maxime Gasse, Prateek Gupta, Seyed Mehran Kazemi, Thibaut Vidal and Giulia Zarpellon for enlightening discussions. We appreciate the helpful comments of the anonymous reviewers. This research was partially supported by IVADO, FRQNT, NSERC, the CIFAR AI Chairs program, Calcul Québec and Compute Canada.

References

Amos, B.; and Kolter, J. Z. 2017. OptNet: Differentiable Optimization as a Layer in Neural Networks. In *ICML*, volume 70 of *Proceedings of Machine Learning Research*, 136–145. PMLR.

Desrochers, M.; Desrosiers, J.; and Solomon, M. M. 1992. A New Optimization Algorithm for the Vehicle Routing Problem with Time Windows. *Oper. Res.*, 40(2): 342–354.

Gasse, M.; Chételat, D.; Ferroni, N.; Charlin, L.; and Lodi, A. 2019. Exact Combinatorial Optimization with Graph Convolutional Neural Networks. In *NeurIPS*, 15554–15566.

Kingma, D. P.; and Ba, J. 2015. Adam: A Method for Stochastic Optimization. In *ICLR (Poster)*.

Lee, J.; Lee, Y.; Kim, J.; Kosiorek, A. R.; Choi, S.; and Teh, Y. W. 2019. Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks. In *ICML*,

volume 97 of *Proceedings of Machine Learning Research*, 3744–3753. PMLR.

Pessoa, A. A.; Sadykov, R.; Uchoa, E.; and Vanderbeck, F. 2018. Automation and Combination of Linear-Programming Based Stabilization Techniques in Column Generation. *INFORMS J. Comput.*, 30(2): 339–360.

Rousseau, L.; Gendreau, M.; and Feillet, D. 2007. Interior point stabilization for column generation. *Oper. Res. Lett.*, 35(5): 660–668.

Sonnerat, N.; Wang, P.; Ktena, I.; Bartunov, S.; and Nair, V. 2021. Learning a Large Neighborhood Search Algorithm for Mixed Integer Programs. *CoRR*, abs/2107.10201.

Uchoa, E.; Pecin, D.; Pessoa, A. A.; Poggi, M.; Vidal, T.; and Subramanian, A. 2017. New benchmark instances for the Capacitated Vehicle Routing Problem. *Eur. J. Oper. Res.*, 257(3): 845–858.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2017. Attention is All you Need. In *NIPS*, 5998–6008.

Velickovic, P.; Cucurull, G.; Casanova, A.; Romero, A.; Liò, P.; and Bengio, Y. 2018. Graph Attention Networks. In *ICLR (Poster)*. OpenReview.net.