Fantastically Eclectic Set Cover Extreme Solver

Rabo Birch, Daniel Firebanks, Emily Hamlin, Chris Ikeokwu

December 21, 2019

Abstract

We develop a deep optimization technique that utilizes a 2D CNN to take an instance of set cover and predicts the approximation algorithm that would perform best on it. Our model was trained on a combination of preexisting datasets and instances that we randomly generated. We used four approximation algorithms as our labels: greedy, deterministic rounding, dual-rounding, and primal-dual. (Sth about training set percentages maybe).We demonstrate that a CNN combined with node embeddings to represent our instance of set cover is effective in solving this classification problem. Our model was able to get to very decent accuracy of around 60% and as high as 90% on some data sets. We found the most important factor for the performance was the resolution of images we create from our node embedding and future work would be to increase this to find the optimal.

1 Introduction

Recent advances in machine learning give the impression that its applications are limitless. As the computational resources available expand, increasingly complex machine learning models can be effectively trained to learn difficult problems. However, one area of problem solving that has seen little attention from machine learning research is NP-hard problems. Typically, these problems' intractable nature leaves them discarded and thought to be best taken care of by traditional approximation algorithms.

However, our research suggests that one should not be too quick to make such a conclusion. Rather than try to entirely oust the place these algorithms have, though, our approach seeks to augment them using machine learning techniques. A key weakness in approximation algorithms is that since they do not produce perfect solutions, there can be great variation in how optimal the solutions they produce are. To combat this, we wish to use machine learning to allow a meta-algorithm to select which approximation algorithm is likely to produce the best results for a given instance of an NP-hard problem. In particular, we have selected the weighted set cover problem as a classical example of an NP-hard problem with well researched approximation algorithms as well as a plethora of real-world applications such as routing and scheduling problems.

Our research seeks to show that machine learning models can be used to increase accuracy of polynomial set cover solutions by dynamically choosing from popular and well-researched approximation algorithms the optimal for a given instance. To do this, we began by synthesizing a dataset that will generalize to applications outside of research environments. This consists of both externally sourced datasets of set cover instances and randomly generated instances that we have produced. We then ran deterministic rounding, dual rounding, primal-dual, and greedy approximation algorithms on this data so that we could label each instance with the algorithm that best solved it. Finally, we trained a convolutional neural network (CNN) on these instances to select the approximation algorithm most likely to produce a more optimal solution.

2 Background & Related Work

2.1 Set Cover Description

The set cover problem is a common problem in the study of algorithms. There are two inputs to the set cover problem: the universe, $E = e_1, \ldots, e_n$, and a list $(S_1, S_2, \ldots, S_n)$ of subsets whose union equals the universe. Each subset S_j has a corresponding weight $w_j \geq 0$. The goal of the set cover problem is

to choose a collection of subsets that covers the universe with minimum total cost. In other words, the goal is to find a list of subsets $S_{i_1}, S_{i_2}, \ldots, S_{i_M}$ such that

$$\bigcup_{k=1}^{M} S_{i_k} = E \ ,$$

$$\sum_{k=1}^{M} w_{i_k}.$$

2.2 Linear Programming

Linear programming is a method of finding the optimal solution to a mathematical problem in the form of an objective function and a set of constraints over a set of *decision variables*. The objective function is a maximization/minimization, and each constraint is a linear equality or inequality. Decision variables generally represent a choice or decision.

Integer programming (IP) is a type of linear programming where the decision variables are constrained to integer values, often 0 and 1. Unlike general linear programming, which has polynomial runtime, integer programming is NP-hard and thus takes exponential time. Because of this, we can use integer programming to solve set cover, which is also NP-hard. The IP for set cover is

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{m} w_j x_j \\ \text{subject to} \quad & \sum_{j:e_i \in S_j} x_j \geq 1, \qquad \forall i \in \{1, \ldots, n\} \\ & x_j \in \{0, 1\}, \qquad \forall j \in \{1, \ldots, m\}, \end{aligned}$$

where w_j is the weight of subset S_j, x_j is a decision variable whose value is 1 S_j is included in the set cover and 0 otherwise, n is the number of elements in the universe, and m is the number of subsets (Williamson and Shmoys, 2011). The objective function represents our goal of minimizing the sum of costs of each subset in our final set cover (notice that if a decision variable is 0, the term containing it will not contribute to the sum). The first constraint makes sure our result is *feasible* by requiring that for each element in the universe, at least one set is chosen that covers that element. Without this, the program would simply set all decision variables to 0 to minimize the objective function, which would output a set cover with no sets. The second constraint requires that each decision variable is set to either 0 or 1 – this is the integer programming part.

2.3 Related Work

The general idea for our project was inspired by the Newman et al. paper "Deep Optimization for Spectrum Repacking" (2017). In this paper, the authors framed the problem of transferring radio spectrum from television to internet as a satisfiability problem. They then created a solver which uses 'deep optimization' to find a solution quickly and accurately. The deep optimization approach involves creating multiple customized algorithms and running them in parallel, which is more complex than what we plan to do.

Our project is more similar to the paper "SATzilla: Portfolio-based Algorithm Selection for SAT" in which the authors "use machine learning techniques to build an empirical hardness model, a computationally inexpensive predictor of an algorithms runtime on a given problem instance based on features of the instance and the algorithm's past performance" (Xu et al., 2011). The authors used the difference in runtime between the optimal and predicted algorithm as the error metric. The approach used in this paper differs somewhat from ours because they used ridge regression for their classifier, whereas we use a deep learning algorithm. In addition, our error metric is based on solution accuracy rather than runtime.

In the paper "Choosing the Efficient Algorithm for Vertex Cover Problem" by K.V.R. Kumar, the author describes and gives detailed pseudo-code for several algorithms that solve set and vertex cover and compares and analyzes their performances on several example graphs (2009). Algorithms provided in this paper include greedy algorithms, linear programming formulations and even genetic algorithms.

In the paper "Neural Combinatorial Optimization with Reinforcement Learning", the authors trained a neural net to come up with optimal solutions for TSP problems rather than using a neural net to pick

between several classifiers (Bello et al., 2016). However, they discovered that even using labelings of optimal solutions the algorithm wasn't able to generalize well. Instead by letting the algorithm explore different paths on its own as a reinforcement learning agent it was better able to understand what made solutions optimal and how to solve them.

In "Combinatorial Optimization with Graph Convolutional Networks and Guided Tree Search", the authors rather take a different approach than the previously mentioned papers, as they give the task of solving certain NP-hard problems to a network instead, and search for optimal solutions using a tree search (Li et al., 2018). A key distinction here is that there is not a space of algorithms previously defined to guide the search, and the novelty lies in the Graph Convolutional Neural Network, accounting for diverse solutions to a problem using a randomized tree search.

Finally, in "Learning Combinatorial Optimization Algorithms over Graphs", the authors combine reinforcement learning and graph embedding (Dai et al., 2017). Specifically, they focus on differentiating the usefulness of every node in a graph for a particular solution of an NP-hard problem, using a vectorized representation of nodes as features, that are later fed into a Q-learning algorithm to find the optimal solution for the problem. The end goal is to develop heuristics that allow the model to adapt to different instances of a graph, using the general knowledge developed during training. This framework can be used to solve many variations of graphs as well as variations of NP-hard problems.

3 Problem

In this project we are generally interested in the problem of accurately classifying instances of the weighted set cover problem that may appear in real-world applications by the polynomial-time approximation that will give the best solution.

While finding the *most* optimal solution is NP-hard, there are a multitude of approximation algorithms which produce solutions within some factor of the most optimal. Given a particular set cover instance I, and some collection of approximation algorithms, some particular algorithm will give the most optimal out of the group. As such, we label each instance I in our dataset with a label y that represents which approximation algorithm this is among those that we have selected.

Our goal is to train a machine learning model to classify instances I into their appropriate labels y. However, we must also discuss an implicit secondary problem as well, which is acquiring data that leads to a trained model which generalizes to new data as well. While our interest in the set cover problem partially lies in its open-ended nature as a problem that models a wide variety of scenarios, this also means that the instances of the problem we may encounter in both real world and theoretical settings vary greatly. Despite this, we have produced a dataset, which we will now discuss further. We reserve a discussion of the creation of this dataset for the Solution section below.

3.1 Data Sets

We used two datasets as inputs to our CNN. The first is a group of 40 set cover benchmark instances (Xu, 2003a). The set cover benchmark instances were generated from a transformed from a group of independent set benchmark instances, which were in turn transformed from a method for producing CSP benchmarks (Xu, 2003b). The CSP benchmarks were produced using Model RB, a type of random CSP model proposed by Ke Xu and Wei Li in their paper Exact Phase Transitions in Random Constraint Satisfaction Problems (2000). The first (non-comment) line of each file is the range of elements in the universe, and each remaining line is a particular subset in the set cover. The model is designed to create difficult but satisfiable CSP and SAT instances. More information on the set cover benchmark instances is provided in Table 1.

The second dataset is from the operations research library for set cover and includes 87 data files (Beasley). We used a group of 50 data files which came from the paper "An Algorithm for Set Covering Problems" (Beasley, 1987). All of the Beasley instance files are stored in a matrix representation: the first line is the number of rows and columns in the matrix, and each following line lists the cost of the column, the number of rows it covers, and a list of the rows it covers.

Our input to the CNN is made up of the 90 instances from preexisting datasets, along with 2692 instances that we randomly generated. See Table 2 for the parameters we used for random generation, as well as the characteristics of each set. We were interested to see if there would be any differences between performance of approximation algorithms on set cover instances where the size of the universe is less than, equal to, or greater than the number of subsets in the problem. Label frequencies for each

Table 1: Set cover benchmark stats (Xu, 2003a).

filename	# of subsets	# of instances	size of min set cover
frb30-15-msc.tar.gz	450	5	420
frb35-17-msc.tar.gz	595	5	560
frb40-19-msc.tar.gz	760	5	720
frb45-21-msc.tar.gz	945	5	900
frb50-23-msc.tar.gz	1150	5	1100
frb53-24-msc.tar.gz	1272	5	1219
frb56-25-msc.tar.gz	1400	5	1344
frb59-26-msc.tar.gz	1534	5	1475

Table 2: Characteristics of our randomly generated instances. m is the range of numbers in the universe, m is the size of the universe, l is the number of subsets, and w is the range of weights.

SC instances	n	m	l	w	Properties
0-499	1000	(50, 100)	(50, 400, 5)	(10, 250)	N/A
500-999	500	(100, 400)	(50, 100, 5)	(3, 50)	$m > l$
1000-1074	5000	(333, 666, 7)	(666, 999, 9)	(66, 969, 3)	$m < l$
1075-1599	5000	(50, 100)	(100, 400, 5)	(3, 50)	$m < l$
1600-2000	6666	(100, 400)	(100, 400)	(69, 420)	$m = l$
2000-2691	1000	(100, 1000)	(50, 1000, 5)	(3, 50)	N/A

dataset are provided in Figure 1 and in Table 3. In all datasets, the most common label by far was greedy. The explanation for this is that the greedy algorithm is a closer approximation than the others. The next most common algorithm was deterministic rounding, followed by primal-dual and finally dual rounding. The fact that all the set cover benchmark instances were labeled greedy is interesting. A possible explanation for this is that he goal of these problems is to be difficult, whereas our random generation had no such goal. A possible explanation for why dual rounding had so few labels is that it's generally the slowest – therefore, in the case of ties, one of the other three approximations that did equally well (especially greedy and primal-dual, which are far faster), would be chosen.

4 Solution

Our solution is to train a convolutional neural network (CNN) on a combination of existing set cover datasets and randomly generated set cover instances in order to predict which approximation algorithm will lead to the smallest cost set cover. We create each training instance using the following steps:

1. Read in or randomly generate a set cover instance.
2. Run the set cover instance on each of our approximation algorithms and set the label of our instance to the algorithm that returned the lowest-cost set cover. Break any ties by shortest runtime.
3. Convert the set cover instance to a biadjacency matrix to input into our neural network.

Table 3: Label Percentages.

SC Instances	Greedy	Deterministic Rounding	Dual Rounding	Primal Dual
0-499	58%	23%	22%	17%
500-999	71%	16%	2%	12%
1000-1074	75%	15%	1%	9%
1075-1599	92%	4%	1%	3%
1600-2000	46%	29%	2%	22%
2000-2691	80%	9.8%	1%	9%
Benchmarks	100%	0%	0%	0%
OR-Library	71%	29%	0%	0%
Total	72%	15%	1%	12%

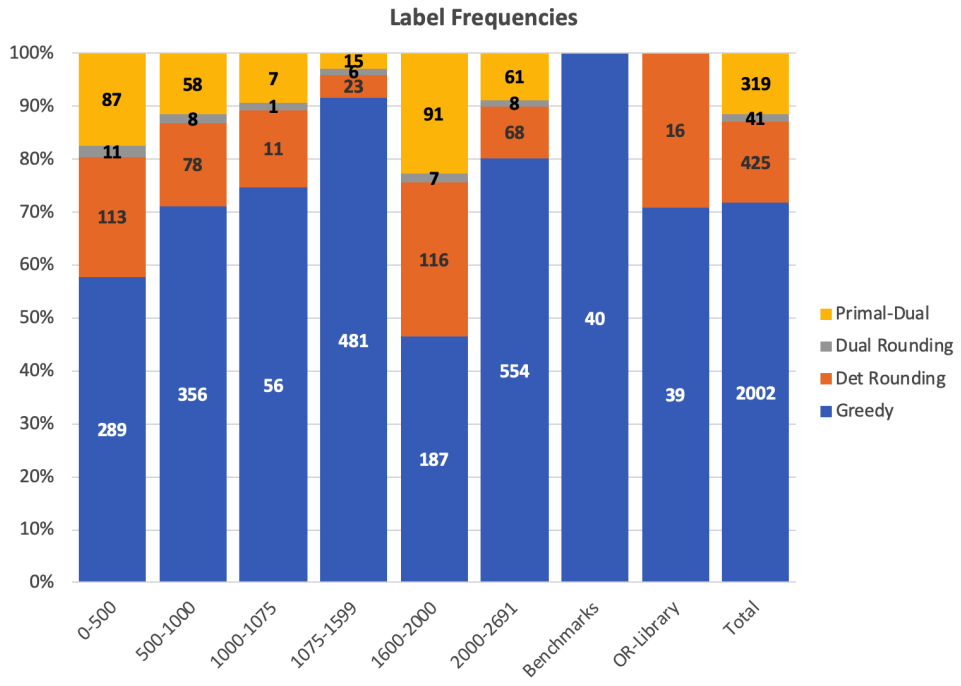

Figure 1: Label Frequencies for our random and preexisting datasets. 'Benchmark' refers to the set cover benchmark instances, and 'OR-Library' refers to the operations research library instances. The numbers on the graph refer to the number of instances with each label.

4.1 Approximation Algorithms

We implemented four different approximation algorithms: deterministic rounding, dual rounding, primal-dual, and a simple greedy algorithm. Each of these algorithms is discussed in depth in *The Design of Approximation Algorithms* by Williamson and Shmoys (2011).

Deterministic rounding refers to a way of rounding the *LP relaxation* of the set cover IP. An LP relaxation is a program where the integer constraint is replaced with a continuous constraint. The set cover LP relaxation is

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{m} w_j x_j \\ \text{subject to} \quad & \sum_{j:e_i \in S_j} x_j \geq 1, \qquad \forall i \in \{1, \ldots, n\} \\ & x_j \geq 0, \qquad \forall j \in \{1, \ldots, m\}, \end{aligned}$$

where the x_js can now be at any value greater than 0. As is, the LP relaxation doesn't give us a feasible set cover – after all, we can't add some fraction of a subset to the cover, we either add it or don't. This is where rounding comes in. We will add a subset S_j to the set cover if $x_j \geq 1/f$, where f is the maximum number of subsets in which any element appears. This algorithm is an f-approximation, meaning that it is guaranteed to produce a solution that is within f times the cost of the optimal solution.

Dual rounding utilizes the *dual* LP for set cover. The dual of a linear program can be systematically constructed from the original (called the primal), can be thought of as its opposite. All solutions to the dual are bounded by the optimal solution to the primal (all higher or all lower, depending on whether the primal is a minimization or maximization problem), and the optimal solution to the primal is equal to the optimal solution to the dual. These two principles are known as weak and strong duality. The

dual problem for set cover is

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} y_i \\ \text{subject to} \quad & \sum_{i:e_i \in S_j} y_j \leq w_j, \qquad \forall j \in \{1, \ldots, m\} \\ & y_i \geq 0, \qquad\qquad \forall i \in \{1, \ldots, n\}, \end{aligned}$$

where y_i is the dual decision variable. For our dual rounding algorithm, we first solve the dual, and then add a subset S_j to the set cover if the dual constraint containing w_j is *tight* – that is, if the inequality in the constraint is a strict equality ($\sum_{i:e_i \in S_j} y_j = w_j$). Dual rounding is also an f-approximation.

The primal-dual approximation algorithm utilizes the relationship between primal and dual LPs in order to create a feasible set cover without ever actually having to solve the LP or the dual. This means that primal-dual is faster than the two rounding schemes. For primal-dual, we start with *any* (not necessarily optimal) solution to the dual. The easiest solution to pick is one where all of the y_is are 0. This is a feasible solution to the dual (it's just a very bad one since the dual is a maximization, but this doesn't matter for our purposes). While the subsets do not cover the universe, pick an element not yet in the set cover and increase its dual variable until some dual constraint goes tight. Add the subset corresponding to this constraint to the set cover and continue. A more detailed algorithm and explanation are provided in Williamson and Schmoys. Once again, this algorithm is an f-approximation.

Our fourth method, a greedy algorithm, is the simplest to understand and also the one with the best approximation guarantee. Greedy algorithms involve choosing some metric and choosing items in order ("greedily") based on how well they do on this metric until we have produced a feasible solution. For our greedy set cover algorithm, simply greedily add subsets to the set cover that minimize the ratio $w_j/|\hat{S}_j|$, where $\hat{S}_j$ is the number of uncovered elements that S_j contains, until we have a full set cover. We used code from the post Greedy Set Cover in Python as a starting point for our implementation, and modified it to work with weighted set cover (Broadhurst, 2017).

4.2 Biadjacency Matrix Representation

We decided to represent each set cover instance as a bipartite graph. A bipartite graph can be written in terms of a biadjacency matrix, where we define one type of nodes n_u to reprsent the elements in the universe and another type n_s to represent the subsets in the instance. We connect a universe node n_{u_i} and subset node n_{s_j} if the element i from the universe is contained or "covered" by the subset j. For example, take the instance:

$$\begin{aligned} \text{Universe:} \quad & E = \{1, 2, 3, 4\} \\ \text{Subsets \& Weights:} \quad & S_1 = \{1, 2, 3\}, \quad w_1 = 2 \\ & S_2 = \{1, 3, 4\}, \quad w_2 = 3 \\ & S_3 = \{1, 2\}, \quad\ \ w_3 = 1 \\ \text{Optimal Solution:} \quad & S_2, S_3 \quad \text{(total weight = 4)} \end{aligned}$$

We first map each element in the universe to an index i in the range of the length of the universe n.

$$Idx(E) = \{(E_1 : i_1), (E_2 : i_2), \ldots, (E_n : i_n)\}$$
$$n = len(E), \quad i \in [0, n]$$

For our example, the resulting map would be:

$$Idx(E) = \{(1 : 0), (2 : 1), (3 : 2), (4 : 3)\}$$

Then, we use those indices in the map as to be the row indices of the biadjacency matrix, while the subsets themselves end up being the column indices. The values of each element in the matrix represents the weight of the that column's subset:

$$
\mathrm{M} = \begin{array}{c} \\ Idx[E_1] \\ Idx[E_2] \\ Idx[E_3] \\ Idx[E_4] \end{array}
\begin{array}{c}
\begin{array}{ccc} S_1 & S_2 & S_3 \end{array} \\
\begin{bmatrix} 2 & 3 & 1 \\ 2 & 0 & 1 \\ 2 & 3 & 0 \\ 0 & 3 & 0 \end{bmatrix}
\end{array}
$$

4.3 Graph Transformation

We considered feeding the adjacency or bi-adjacency matrix representations of our graph directly into a CNN. However, CNNs depend on local correlations between pixels to classify images. If you feed in adjacency matrices directly as an image there is no locality between "pixels". However, in the paper "Graph Classification with 2D Convolutional Neural Networks" (Tixier et al., 2017), the authors describe a method of converting graphs into images where the pixels correspond to properties of the actual graph and have the desired locality property. We took their process as a starting point and modified it to work on weighted graphs. We give a more elaborate version of the description of the process below.

1. *Graph Node Embeddings*: Node embeddings are created by taking random walks from a starting node in the graph and, for every other node we come across, keeping track of the number of steps from the start it took to get there. Nodes that are reached in fewer steps are transformed into vectors such that the distance between two vectors are inversely correlated with the number of steps it took to get between the nodes they represent.

2. *Alignment and compression with PCA*: As the node embeddings are stochastic, dimensions are recycled from run to run, which means that a given dimension will not be associated with the same latent concepts across graphs, or across several runs of the same graph. Therefore, to ensure that the embeddings of all the graphs in the collection are comparable and because often times graphs are of different starting dimensions, we apply PCA and retain the first $d << D$ principal components (where D is the dimensionality of the original node embedding space). PCA also serves an information maximization (compression) purpose. Compression is desirable in terms of complexity, as it greatly reduces the shape of the tensors fed to the CNN at the expense of a negligible information loss.

3. *2D Histograms*: We finally repeatedly extract 2D slices from the d-dimensional PCA node embedding space, and turn those planes into regular grids by discretizing them into a finite, fixed number of equally-sized bins, where the value associated with each bin is the count of the number of nodes falling into that bin. In other words, we represent a graph as a stack of $d/2$ 2D histograms of its (compressed) node embeddings.

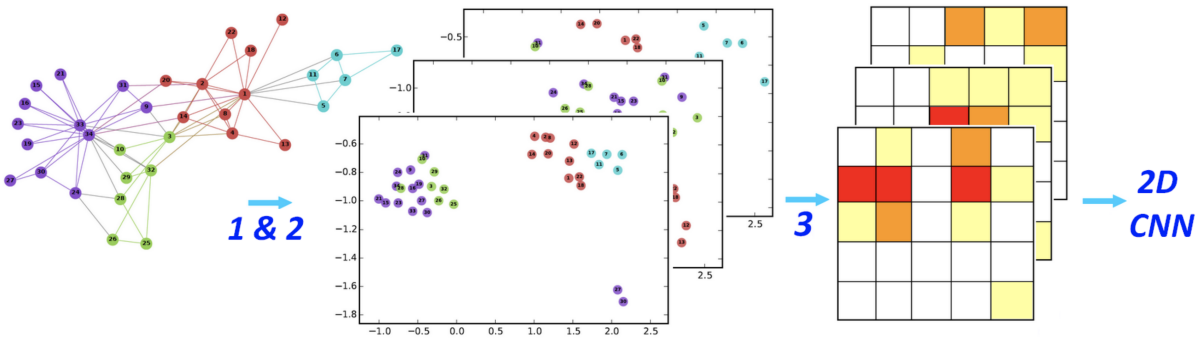

Figure 2: Transformation of graph to 2D histograms (Tixier et al., 2017)

5 Experimental Setup

5.1 Performance Measures

In this report, we plan to evaluate our models using 5-fold cross validation and accuracy. Accuracy is a good performance measure when we are interested in the percentage of correct classification our network would enable us to get more optimal solutions on as opposed to using the approximation algorithms by themselves. We decided to use 5 folds because this would correspond to roughly a training set of 80% of the total and validation of 20%. We decided to use cross validation since the greedy label dominated, we had limited data, and we wanted to train our model in a way that emphasizes generalizability.

5.2 Histogram Computation

We start by using the graph embedding library 'node2vec' developed by Stanford in "node2vec: Scalable Feature Learning for Networks" (Grover and Leskovec, 2016) but modified their python version to take weighted graphs. Specifically, we weigh the transition probabilities in the random walk by the edge between nodes. We chose to create nodes of 128 dimensions. We considered 512 and 1024 dimensions but found that this didn't improve the perfomance of our CNN. For PCA, we experimented but decided to keep the 20 first principal dimension as this explained $90 - 99\%$ of the variance for most of our instances. We varied the number of bins but found the larger the number bins for our histogram, the better the model performed. This is largely due to resolution because when we reduce the bins, nodes that might have been in different 2D slices of the PCA embedding space would now be in the same one and some information would be lost. However, we were limited by how large we could make them and run in a reasonable amount of memory on our machine but found that the more bins the better the performance of the CNN

5.3 Network Architecture

We implemented a simple CNN architecture that deploys four convolutional-pooling layers (each repeated twice) in parallel, with respective region sizes of 3, 4, 5 and 6, followed by two fully connected layers. We employed dropout for regularization at every hidden layer and ultimately picked 0.5. We originally started at 0.3 but found our model was prone to overfitting at this level. The activations are ReLU functions except for the ultimate layer, which uses a softmax to output a probability distribution over the approximation algorithms. For the convolution-pooling block, we employ 64 filters at the first level, and as the signal is halved through the (2,2) max pooling layer, the number of filters in the subsequent convolutional layer is increased to 96 to compensate for the loss in resolution. We trained our model using Adam optimizer with a learning rate of 0.0001 and our loss function was categorical crossentropy because this is a multiclass classification problem.

6 Results

Initially, we achieved 90% accuracy on our test set after training on our dataset of 1000 instances, while using the tesla01 computing cluster and a resolution of 400 and dropout of 0.3. During our first experiments with this configuration, our model started overfitting after about 15 epochs. While training accuracy went up our validation accuracy started dropping below 90%, but the validation accuracy was still high around 83%. This could be due to the large input dimension of the images, which caused our CNN to have a lot of neurons for very few data points and thus allowed the model to memorize the instances rather than finding patterns after a certain point.

Later on, we reduced the size of our training set, increased the number of channels to 10, decreased the learning rate and increased our dropout to 0.5.

Due to memory issues, we were not able to experiment with images of size bigger than 100 pixels, and we realized that low-resolution images would give the worst results. A possible explanation is the fact that high-resolution images were able to provide more information to the model per histogram, so if we had been able to experiment with more memory we could have probably achieved a better result.

The results on each dataset were quite similar, with the exception of 500-999, which had the highest accuracy. There doesn't appear to be any strong connection between the distribution/parameters of each dataset and the performance of the model. Overall, the lower end of the accuracies ranged from $0.47 - 0.56$, whereas the outlier high scores were of 0.6 and 0.7 on ***0-2125*** and ***599-900***, respectively.

Table 4: Results

Dataset	**Channels**	**Resolution**	**Learning rate**	**Dropout**	**Accuracy**
500-999	10	80	0.001	0.5	0.5
500-999	10	80	0.0001	0.5	0.56
1600-2000	10	80	0.001	0.5	0.47
500-999	10	100	0.00005	0.5	**0.7**
1600-2000	5	100	0.00001	0.5	0.53
0-1000	5	400	0.001	0.3	**0.9**
0-2152	5	165	0.0001	0.5	0.68

However, these are not significant enough across our experiments to make generalized claims on the model performance. Then, we can evaluate the performance of our model across all datasets equally.

First of all, unsurprisingly, the experiments with less data yielded low results, as the model got stuck on a particular loss after the first 10 epochs and we couldn't see much of a change afterwards. This probably happened because the samples we fed were not enough for the model to learn useful patterns. A second conclusion we reached was that if the image representation of our graph had high resolution, we were able to increase our accuracy no matter the distribution of the initial instances. An interesting corollary is that, we know that this architecture was designed to tackle the MNIST challenge of small sized images during the late 90s/early 2000's, but we also speculate that modern models could perform better in our bigger sized images. Finally, a smaller learning rate performed better than the default 0.01, and our results had more strength against overfitting once we increased the dropout.

7 Conclusions & Future Work

It seems that the most important factor in how well our modeled performed was the resolution of the histograms. Since we were coming from a very higher dimensional space in the first place, it seems that we would need very high definition images to be able to correctly classify our approximation algorithms.

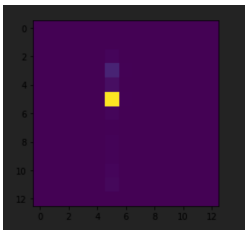

Figure 3: Histogram with small resolution (12)

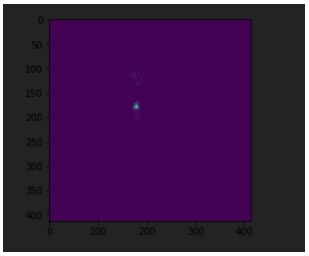

Figure 4: Histogram with large resolution (400)

Thus to improve our model going forward we would hope to have access to a computer with bigger memory to handle larger-resolution images. Additionally, to address our CNN getting stuck at a particular accuracy, we would want to experiment with more varied architectures or with some of the graph-based neural net architecture.

We also hope to train our models on bigger datasets and train them on some of the more real world data sets that we obtained. Node2vec also lets us vary whether we used random walks that are breadth first or depth first. Right now we did half of the walks with each. Breadth first helps learn more about the larger macro structures of the graph while depth first helps us learn more about the local structures. We didn't have enough time to experiment about whether the local or macro structures of the graph are more important to the approximation algorithms and playing with these parameters might be able to help with that.

References

J E Beasley. Or-library. URL http://people.brunel.ac.uk/~mastjjb/jeb/orlib/scpinfo.html.

J.E. Beasley. An algorithm for set covering problem. *European Journal of Operational Research*, 31(1): 8593, 1987. doi: 10.1016/0377-2217(87)90141-x.

Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016. URL http://arxiv.org/abs/1611.09940.

Martin Broadhurst. Greedy set cover in python, Feb 2017. URL http://www.martinbroadhurst.com/greedy-set-cover-in-python.html.

Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *CoRR*, abs/1704.01665, 2017. URL http://arxiv.org/abs/1704.01665.

Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks, 2016.

K.V.R. Kumar. Choosing the efficient algorithm for vertex cover problem, Jun 2009. URL http://gdeepak.com/thesisme/thesis-Choosing%20the%20Efficient%20Algorithm%20for%20Vertex%20Cover%20problem.pdf.

Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. *CoRR*, abs/1810.10659, 2018. URL http://arxiv.org/abs/1810.10659.

Neil Newman, Alexandre Fréchette, and Kevin Leyton-Brown. Deep optimization for spectrum repacking. *CoRR*, abs/1706.03304, 2017. URL http://arxiv.org/abs/1706.03304.

Antoine Jean-Pierre Tixier, Giannis Nikolentzos, Polykarpos Meladianos, and Michalis Vazirgiannis. Graph classification with 2d convolutional neural networks, 2017.

David P. Williamson and David Bernard. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.

Ke Xu. Benchmarks with hidden optimum solutions for set covering, set packing and winner determination, Oct 2003a. URL http://sites.nlsde.buaa.edu.cn/~kexu/benchmarks/set-benchmarks.htm.

Ke Xu. Forced satisfiable csp and sat benchmarks of model rb - a simple way for generating hard satisfiable csp and sat formulas, Oct 2003b. URL http://sites.nlsde.buaa.edu.cn/~kexu/benchmarks/benchmarks.htm.

Ke Xu and Wei Li. Exact phase transitions in random constraint satisfaction problems. *CoRR*, cs.AI/0004005, 2000. URL http://arxiv.org/abs/cs.AI/0004005.

Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for SAT. *CoRR*, abs/1111.2249, 2011. URL http://arxiv.org/abs/1111.2249.