Cognitive Robotics Project (097244)
Spatial planning in a dynamic environment using convolutional neural networks

Yoav Meiri, Ido Yutko

March 2023

Abstract

We consider the problem of spatial path planning, in an environment containing moving obstacles. In contrast to the classical solutions, in this work we learn a planner from the data (in a supervised learning manner) that allows us to leverage statistical regularities from past data. For that purpose we will use a state of the art machine learning model. In the setting where the ground truth map (real distance from the goal from each tile) is not known to the agent, we leverage pre-trained deep learning models (which in our case will contain convolutional neural networks) in an end-to-end framework that has the structure of mapper and planner built into it which allows seamless generalization to new maps and goals.

1 Overview

Path planning in an environment with obstacles is the task of finding a collision-free path between a starting point and a goal point in a complex environment. This task is particularly important in various fields, such as robotics, autonomous vehicles, and video games, where it is necessary to navigate through obstacles while avoiding collisions. The objective of path planning is to find the shortest path possible while considering the constraints of the environment, such as the size and shape of the obstacles.

There are several common ways to tackle the problem of path planning in an environment with obstacles. One approach is to use search algorithms, such as A* or Dijkstra's algorithm, which search through a graph of possible paths and select the shortest one that avoids obstacles. These algorithms use heuristics to guide the search towards the goal and avoid exploring paths that are unlikely to lead to a solution. A* algorithm, for example, combines the cost of the path with a heuristic estimate of the remaining cost to the goal to determine which path to explore next.

Machine learning techniques, such as neural networks and reinforcement learning, are also being explored for path planning tasks, and this will be the focus of this work. In these approaches, the algorithm learns from experience, either through supervised learning or by trial and error. Reinforcement learning, in particular, has shown promising results in complex environments with a large number of possible paths.

On top of that, looking at a frequent task in a similar environment (in a manner that it is reasonable to assume the existence of a distribution over the states of the problem), it is only logical to exploit the statistical characteristics of the environment. Finding a way to represent and use statistical properties of the environment (for e.g., walls are mostly parallel or perpendicular to each other) can allow us to plan more easily in future instances of the same problem.

Obstacles moving in a random manner require us to find a new plan each time the environment changes. Starting the planning process each time from scratch might not be the most efficient thing to do. Exploiting the previously discussed statistical properties of the problem may lead to a more efficient and faster planning for new instances of the problem.

Us humans, we can easily see a path in a relatively simple environment even if it contains obstacles. After a quick search we found out the name of this phenomena is called "perceptional pop-up". It was stated before that neural networks in general, and convolutional neural networks in particular demonstrate human-like perception skills. Under this assumption we wanted to measure how well a CNN (convolutional neural network) model can assess distance from the goal from each point in the environment given an image of the environment as input.

1.1 Problem Definition

We are given a $n \times n$ matrix in which free spaces are denoted by 0 and obstacles are denoted by 1, a starting tile and a goal tile. The agent's task is to reach the goal state from his starting location, while able to move only up, down left or right. On top of that, **the obstacles are moving** in a random (yet not arbitrary) way between time points (also up, down left or right), and the agent mustn't collide with any of them or go out of bounds. **The environment is fully observable to the agent**.

Due to the moving obstacles this problem to be non trivial. The obstacles movement is not given as input or has a simple traceable pattern. Therefore, in each time point the agent needs to reprocess the new environment and act accordingly.

Let us notice that the problem as described is not a classical supervised learning problem. This setup will later on be translated to a machine learning problem we will use CNN's to solve.

2 Data

As described earlier, the main component of the data was a binary map describing the environment. These are the components of each data point in out dataset:

1. *Map* - $n \times n$ binary matrix where 0 are free spaces and 1 are obstacles.
2. *Start* - $n \times n$ binary matrix filled with 0 except for a 1 in the starting location.
3. *Goal* - $n \times n$ binary matrix filled with 0 except for a 1 in the goal location.
4. *Distances* - $n \times n$ matrix where each non-obstacle location contains it's real distance (in steps) from the goal while taking he obstacles into account. The distances correspond with the input MAP.

2.1 Data Generating Process

We manually generated the data. We enforced the obstacles to have shapes and sizes that will make sense in a real world environment, and we located them randomly in the map. Also, we made sure that in the initial state the agent has a possible path to reach the goal. We generated around 30,000 training samples and other 7,500 test samples. Theoretically we could have generated an infinite amount of data, yet we needed our model training process to end in a reasonable time. In figure 1 we can see some samples from the generated dataset.

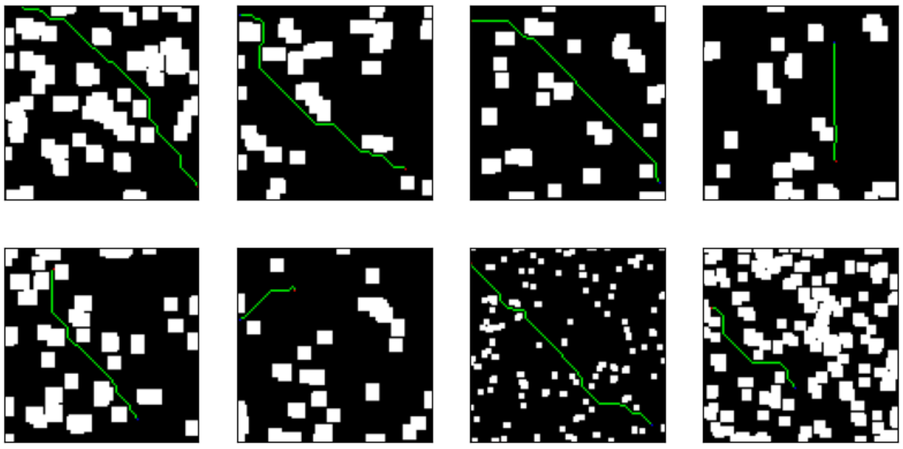

Figure 1: Example of data sample maps. In green we can see the path between the start and the goal tile (which is highlighted in red

3 Simulation

We built our simulation completely from scratch. In practice, the working horse of our modeling was the simple Numpy ndarray. In the array each numerical value is associated with an entity (obstacle, agent) or state of a free tile (simple empty tile, goal tile). We built the simulator using numpy and other python OOP tools (for entity modeling).

We decided not to use any premade simulator and build our own because we needed both good compatibility with the dataset format and ability to control the obstacle movement. We couldn't find any suitable premade simulator. On top of that, this formatting allowed us to feed the simulator states to a pytorch[1] model without any third party scripts or significant processing.

[1]python library for deep learning models

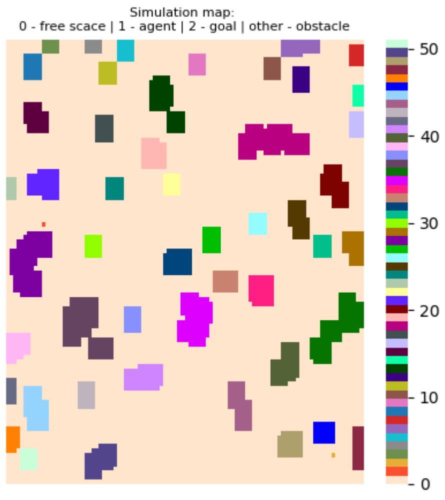

Figure 2: Example of a snapshot of a simulated environment. The small red dot is the agent and the yellow dot is the goal location. Each obstacle has a different color which corresponds to it's unique numerical value in the environment state matrix.

3.1 Obstacles

As mentioned, in order to make this work non trivial and representative of the real world we needed to model and add obstacles to our environment. As can be seen above, the shapes, sizes and spread of the obstacles make the environment challenging and realistic. In order to harden the problem, we decided to add the obstacles the ability to move between time points. We decided to make the movement of the obstacles random. At first, each obstacle in each time point moved in a single random direction. This lead to a oscillation like movement that was both not very realistic and was not very challenging for the agent to deal with (because for the most part the obstacles remained in a small area around their initial location).

In order to solve this issue and make the movement more realistic we decided to use a **UCB (upper confidence bound) technique** in which the obstacles have a certain momentum to their moving that caused them to maintain a single direction for longer time periods. On top of that, the UCB mechanism leads to a "explore - exploit" type behavior, which prevents the obstacles to stick to a single direction through the whole run and change direction when following a single direction for a long time.

The movement is still random, but we artificially changed the distribution in order to make the movement more realistic.

3.2 Agent

In our environment, the agent size is a single tile. He can move only in the 4 traditional directions and 1 tiles in each turn. His goal is of course to reach the goal tile in the least number of turns possible. He must avoid collisions with obstacles, otherwise the game ends and his utility is 0 (in contrast to the 1 he gets when he reaches the goal). In each turn (including the initial state), he sees the map and the location of each obstacle (which tiles are free and which are not).

As mentioned before, the moving obstacles require the agent to recalculate his path in each turn. For this purpose he uses planning tools which can get only what the agent sees (map boundaries and obstacles location).

4 Proposed Method

4.1 Background & Motivation

A convolutional neural network (CNN) is a type of artificial neural network that can analyze visual imagery and other types of data that have a spatial structure. A CNN uses a shared-weight architecture of convolution kernels or filters that slide along input features and provide feature maps. A CNN typically consists of three types of layers: convolutional layers, pooling layers, and fully-connected layers. The convolutional layers perform local feature extraction by applying filters to small regions of the input. The pooling layers perform downsampling by reducing the size and dimensionality of the feature maps. The fully-connected layers perform classification by combining the features learned by previous layers into a final output vector.

Our objective is to develop methods that can learn to plan from data. However, a natural question is
why do we need learning for a problem which has stable classical solutions?
One key reason, is that classical methods do not capture statistical regularities present in the natural world, (as stated in the introduction part), because they optimize a plan from scratch for each new setup. This also makes analytical planning methods to be often slow at inference time which is an issue in dynamic scenarios where a more reactive policy might be required for fast adaptation from failures. A learned planner represented via a neural network can not only capture regularities but is also efficient at inference as the plan is just a result of forward-pass through the network (in our case it includes matrix multiplication and convolution operation).

4.2 Using the model for planning

Let's take a look at a snapshot of the environment in some time point. The agent sees the goal location and the locations of all of the obstacles, and now he needs to decide which tile to go to. He needs to assess the distance to the goal from each possible next tile. For that purpose he can use our method or the Dijkstra baseline.

As mentioned before, after using the Dijkstra method the distance matrix contains the **real** distance to the goal from each possible next tile.

If the agent uses our method, first he passes the binary obstacle map and the goal matrix through the model and gets a predicted distance matrix. He uses this matrix to continue to the tile which is closest to the goal.

4.3 Model Architecture

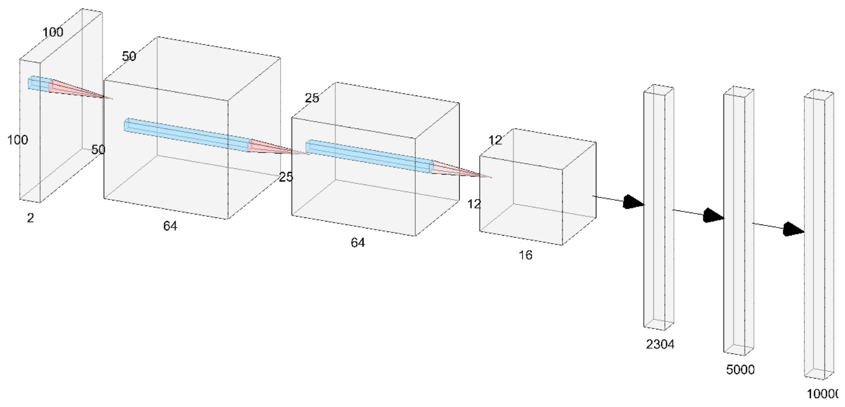

Figure 3: A visualization of the CNN model we used. The activation functions, pooling layers and batch normalization layers do not appear here.

Our main goal is to minimize the difference between the model predictions for the distance from the goal to each tile, to the real distances (which appear in the data). Our model

Other than that, make the network as small and compact as possible. Small networks with fewer parameters will be lighter to store, and will result in faster forward passes and training time.

The **first layer** of the CNN is a convolutional layer that applies 64 filters (or kernels) to the input image. Each filter has a small spatial size (3 by 3) but extends through the full depth of the input image (2 channels). The convolutional layer performs a dot product between each filter and a small region of the input image, producing a 2D activation map for each filter. The activation maps are then stacked along the depth dimension, forming the output volume of the convolutional layer. The output volume has a size of W_{out} by H_{out} by 64, where W_{out} and H_{out} depend on the filter size, the stride (the number of pixels the filter moves at each step), and the padding (the number of pixels added to the border of the input image).

The **second layer** of the CNN is a max pooling layer that reduces the spatial size of the output volume from the previous layer. Max pooling is a form of downsampling that applies a max operation to non-overlapping regions of the input. This reduces the width and height of the output by half, while preserving the depth. The max pooling layer has no parameters to learn; it only helps to reduce computation and prevent overfitting.

After these first 2 layers there are 2 more convolutional layers and max pooling layers.

The **sixth layer** of the CNN is a fully connected layer that takes the output volume from the previous layer and flattens it into a vector. A fully connected layer connects every neuron in one layer to every neuron in another layer. It is in principle the same as a standard neural network layer. The fully connected layer has parameters that are learned during training.

The **seventh layer** of the CNN is another fully connected layer that takes the output vector from the previous layer and applies a nonlinear activation function, such as ReLU, to it. ReLU stands for rectified linear unit, and it is defined as f(x) = max(0,x). It introduces nonlinearity to the network, which enables it to learn more complex functions.

The **eighth and final layer** of the CNN is another fully connected layer that takes the output vector from the previous layer and transforms it into a 100 by 100 matrix, which represents the predicted distance map. The distance map indicates how far each pixel in the input image is from the goal.

On top of that, after every pooling layer we added a batch normalization layer. It works by normalizing the inputs to each layer so that they have zero mean and unit variance. This helps to reduce the internal covariate shift problem, which is when the distribution of inputs to each layer changes during training. By reducing this problem, batch normalization can help to improve the stability and performance of neural networks

4.4 Training process

The training process of our CNN involved the following steps:

1. Initialize the weights and biases of the CNN layers randomly.

2. Divide the training data into batches of size 50, which means each batch contains 50 samples of input and output data.

3. For each batch, feed the input data to the CNN and compute the output using a **forward pass**. The output is a prediction of the desired output for each sample in the batch.

4. Compare the prediction with the actual output using a loss function, which in our case was MSE (Mean Squared Error), which measures the average squared difference between the prediction and the actual distance matrix. The loss indicates how well the CNN performs on the batch.

5. Compute the gradient of the loss with respect to the weights and biases of the CNN layers using a backward pass. The gradient is a vector that points in the direction of steepest increase of the loss function.

6. Update the weights and biases of the CNN layers using the ADAM (Adaptive Moment Estimation) optimizer, which adjusts the learning rate and momentum based on the gradient and previous updates. The update reduces the loss for the batch and improves the performance of the CNN.

7. Repeat steps 3 to 6 for all batches until one epoch is completed. An epoch is one full pass through the entire training data.

8. Repeat steps 2 to 7 for a predefined number of epochs or until some convergence to a minimal loss.

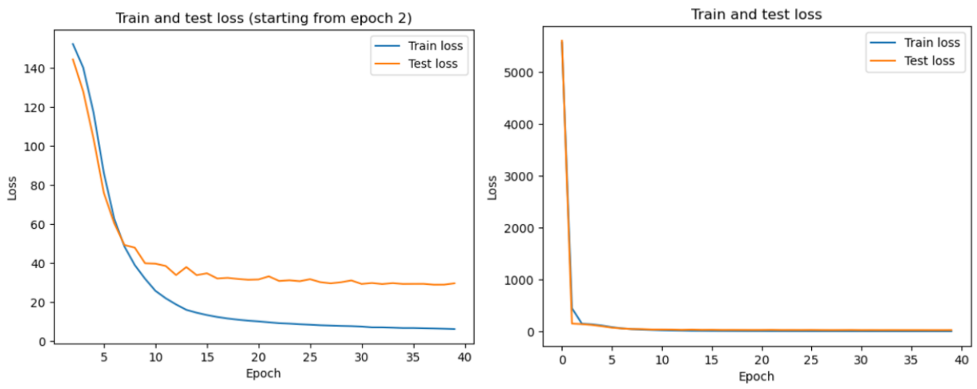

Figure 4: The loss function throughout the training process for both the test set and training set. Both graphs describe the same function, yet the right one is starting from 4 so that we can see the loss decreasing between epochs 2 and 40

5 Experiments & Results

5.1 Preliminary Experiments

At first, we tried using a CNN that contains only convolutional and pooling layers (**without the fully-connected head**). In order to maintain the spatial dimensions of the output we used a U-net type CNN [1]. Using a fully connected head significantly increases the number of model parameters. A more complex model (in terms of parameter number) is more likely to lead to overfitting, will require more time for each forward pass and will be heavier (in terms of storage requirements).These are all things we wish to avoid, so it was only logical to not include the fully-connected head at first. Unfortunately, this type of models does not excel in regression task (they are more suitable for classification and segmentation task) and the model performed poorly. In figure 5 there is an example for a prediction done by the best trained model we obtained without using fully-connected head. As we can see, although there is a basic ability to give smaller values to tiles closer to the goal there is no way this result can be accounted on for planning.

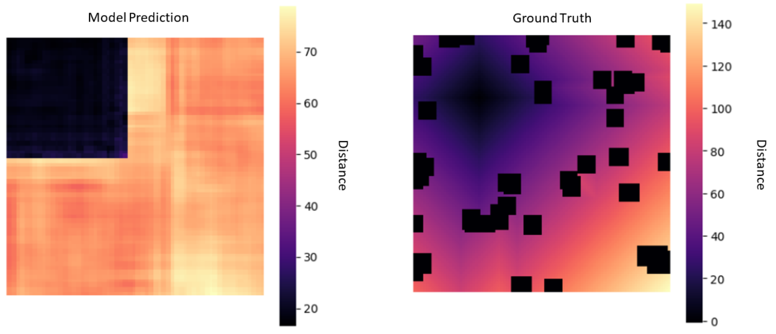

Figure 5: An example for a predicted distance matrix done by a U-net like CNN (no fully-connected head) after the training process reached convergence. The obstacles are in black.

Our main conclusion from this experiment is that **a fully-connected head is vital for reasonable performance in this task**, even if it means that a forward pass will take more time and the model will be heavier to store.

After this experiment we removed the "expanding" half of the U-net and after the first half added the fully-connected layers. This was the final architecture we described in section 4.3.

The whole training process took about 40 minutes and the results were significantly better compared to those of the model with no fully-connected head.

5.2 Smoothing the result

After our model's performance was sufficient and converged to a very small loss value, we faced another big challenge. Although the model performance was good enough, is was not suitable for the greedy behavior we set for the agent. We decided to solve this issue using a convolution filter full of 1's (which we divided by the kernel size squared) with kernel size 5 we applied across the image. The result was that every entry in the matrix contained more global knowledge about the entries around it. This practically solved this issue and the processed result performed well in the planning part.

5.3 Results

For result production we run our simulator multiple times in multiple maps and documented both the number of steps needed to reach the goal and the average time needed to compute each step.
Model Performance - In figure 4 we saw that our model reached convergence. In the examples below we see that the model predicts the distances to a good level. It is noticeable that the model learned also to take the obstacles into account. Under the assumption that the agent knows the location of the obstacles, it doesn't matter which values the model predicts for the obstacle tiles (the agent will not consider those in the first place). Therefore, we can assess the model considering only the obstacle masked version.

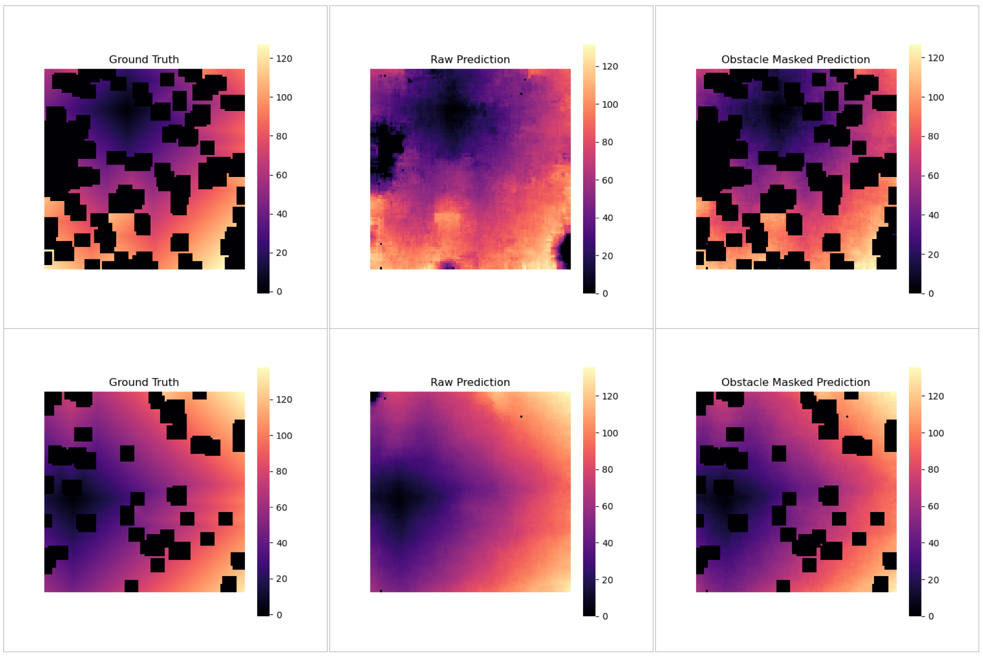

Figure 6: 2 Examples for prediction made by the model. Each row describes a single example

Path Length - For the most part, there was no significant difference in the number of moves between the model and the baseline. We stated before that our Dijkstra baseline produces the **real** distances from the goal in each time point, thus produces the best greedy movement possible. There is no optimality guarantee in our proposed method, yet the similar number of moves indicates that our model is close to being optimal. In figure 7 we present the distribution of number of moves, where we ran the simulation from the same initial state 100 times using the model and 100 times using the baseline. We can see significant overlap between the 2 histograms that support our claim that our model is close to being optimal.

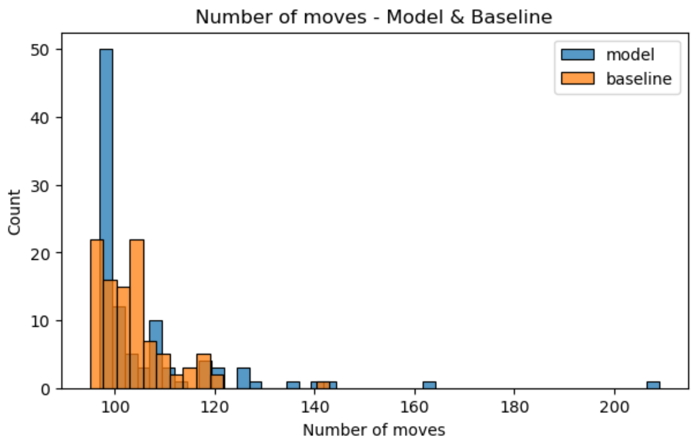

Figure 7: A histogram of all path lengths over 100 simulations in a random map for both the baseline and the model.

Planning Time - As discussed in section 4.1, we wished for our model to enable faster planning compared to the baseline we used. For the most part, our model indeed enabled faster planning. In figure 8 we present the average time per move we documented when we ran our simulation 100 times with the same initial state for both the model and the baseline. We see that our model planned faster than the baseline as expected.

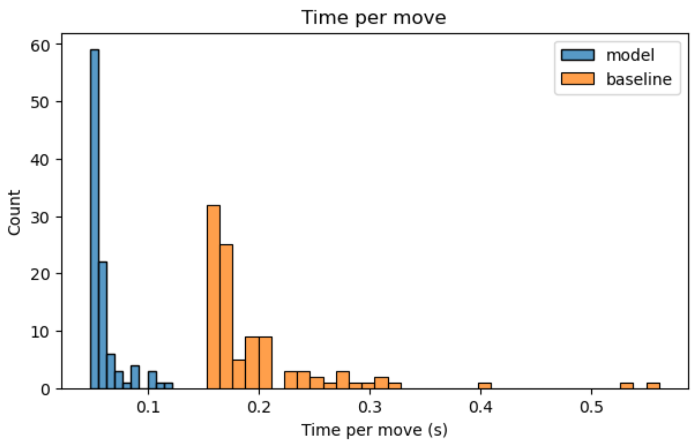

Figure 8: A histogram of all average time per move values over 100 simulations in a random map for both the baseline and the model.

6 Discussion

In this work, we presented a novel CNN-based approach for spatial planning task, which aims to find the optimal layout of a set of moving objects in a given space. We compared our model with a baseline method that uses Dijkstra's algorithm to search for the shortest path to the goal. Our experimental results showed that our model can achieve faster and close to optimal solutions than the baseline method, while also being able to handle complex and dynamic scenarios. We attribute the success of our model to its ability to learn high-level features and spatial relations from the input images, and to generate feasible and diverse layouts using a decoder network (fully-connected head). Our model can be applied to various domains that require spatial planning, such as urban design, interior design, robotics, and game development.

7 Reflection

We learned a lot from this work, both in terms of technical skills and research insights. We gained experience in building a simulator from scratch, using various tools and libraries such as NumPy, and OpenCV. We also learned how to design and implement CNNs using pyTorch, and how to optimize them using techniques such as data augmentation, dropout, batch normalization, and learning rate decay. We also learned how to analyze and interpret the results of our experiments, and how to communicate them effectively in a report.

We faced many challenges and difficulties along the way, while devoting an enormous amount of time and effort to deal with them, but we also enjoyed the process of solving these problems and finding creative solutions. We are proud of our achievements and hope some day to expand our work in this field.

8 Code & Video

All of our code appears in this GitHub repository. Please contact us if you are having any trouble finding it or using it. A YouTube video in which we explain our work and demonstrate it can be found in this link.

References

[1] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.