LangProp: A code optimization framework using Large Language Models applied to driving

Shu Ishida[1,2]*, **Gianluca Corrado**[1], **George Fedoseev**[1], **Hudson Yeo**[1], **Lloyd Russell**[1], **Jamie Shotton**[1], **João F. Henriques**[2] & **Anthony Hu**[1]
[1]Wayve Technologies [2]Visual Geometry Group, University of Oxford
research@wayve.ai, ishida@robots.ox.ac.uk

Abstract

We propose LangProp, a framework for iteratively optimizing code generated by large language models (LLMs), in both supervised and reinforcement learning settings. While LLMs can generate sensible coding solutions zero-shot, they are often sub-optimal. Especially for code generation tasks, it is likely that the initial code will fail on certain edge cases. LangProp automatically evaluates the code performance on a dataset of input-output pairs, catches any exceptions, and feeds the results back to the LLM in the training loop, so that the LLM can iteratively improve the code it generates. By adopting a metric- and data-driven training paradigm for this code optimization procedure, one could easily adapt findings from traditional machine learning techniques such as imitation learning, DAgger, and reinforcement learning. We show LangProp's applicability to general domains such as Sudoku and CartPole, as well as demonstrate the first proof of concept of automated code optimization for autonomous driving in CARLA. We show that LangProp can generate interpretable and transparent policies that can be verified and improved in a metric- and data-driven way. Our code is available at `https://github.com/shuishida/LangProp`.

1 Introduction

Building systems that can self-improve with data is at the core of the machine learning paradigm. By leveraging vast amounts of data and having an automated feedback loop to update models according to an objective function, machine learning methods can directly optimize the metrics of interest, thus outperforming systems that are handcrafted by experts. In the early history of artificial intelligence (AI), Symbolic AI, e.g. rule-based expert systems (Hayes-Roth, 1985; Jackson, 1986), was a dominant and perhaps a more intuitive and explainable approach to solving tasks in an automated way, and is still widely used in fields such as medicine (Abu-Nasser, 2017) and autonomous driving (Badue et al., 2021). However, there have been numerous successes in recent decades in machine learning, e.g. deep neural networks, that demonstrate the advantage of data-driven learning.

Advances in Large Language Models (LLMs) (Brown et al., 2020; OpenAI, 2023; Touvron et al., 2023) were enabled by neural networks. Trained on both natural language and code, they can translate human intent and logic into executable code and back, expanding the boundaries of applying logic and reasoning. Unlike other machine learning techniques, LLMs have an affinity with Symbolic AI since they operate in discrete symbolic input-output spaces. The generated outputs are interpretable, even though the internal representation of these tokens is in a continuous embedding space. This observation led us to question if it is possible to have the best of both worlds – having an interpretable and transparent system, characteristic of Symbolic AI, which can self-improve in a data-driven manner, following the machine learning paradigm. We believe that LLMs provide the missing piece of the puzzle; the optimization mechanism.

Our insight is that we can draw a direct analogy from training neural networks, and *train* symbolic systems by leveraging the power of LLMs to interpret and generate scripts. Using this analogy, an LLM can be considered as an *optimizer* equivalent to stochastic gradient descent or Adam. The actual *model* in our paradigm is an object that handles the initialization and updates of *parameters*

[*] Work done during an internship at Wayve

as well as the forward pass logic, where the *parameters* are a collection of symbolic scripts that the LLM generates. At every iteration, we perform a forward pass through the model, compare it against the ground truth in the dataset, and pass the scores and feedback into the LLM which interprets the results and updates the scripts in a way that fixes the issues raised.

While many methods use LLMs for code generation, and systems such as Auto-GPT (Richards, 2023) iteratively query LLMs to execute tasks in an agent-like manner, as far as we know, we are the first to completely translate and apply the training paradigm used in machine learning for iterative code generation. We draw inspiration from VOYAGER (Wang et al., 2023), which introduced the idea that a collection of LLM-generated code (skill library) can be considered as sharable and fine-tunable *checkpoints*. However, VOYAGER's method is specific to Minecraft, and additional work is needed to apply its approach to other domains. We propose LangProp, a code optimization framework that is easily adaptable to many application domains.

Autonomous driving is a key area in which model interpretability and transparency are critical. We consider LangProp to be a valuable proof of concept for building interpretable and language-instructable systems in a more automated and learnable way. We tested our hypotheses that (a) LangProp can generate interpretable code that learns to control a vehicle, (b) LangProp can improve driving performance with more training data in comparison to zero-shot code generation, and (c) we can easily transfer training paradigms from machine learning to LangProp such as imitation learning (IL), reinforcement learning (RL) (Sutton & Barto, 2018) and DAgger (Ross et al., 2011).

2 RELATED WORK

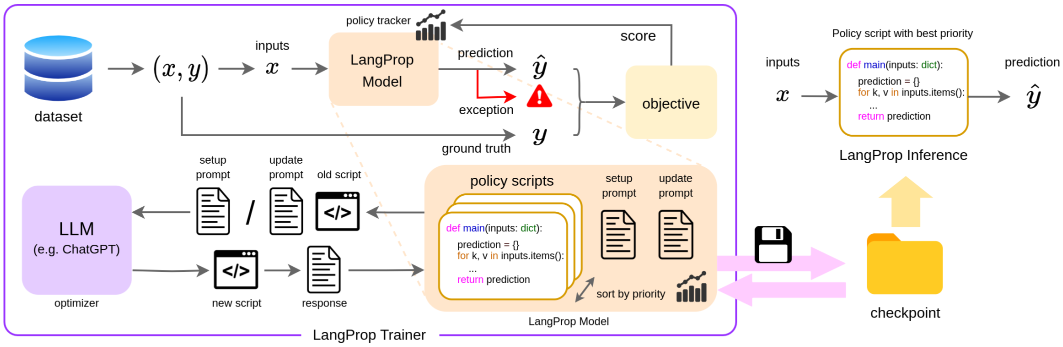
2.1 LLMS FOR CODE GENERATION

Transformers (Vaswani et al., 2017) have shown outstanding performance in code generation tasks (Chen et al., 2021; Li et al., 2022; Xu et al., 2022; Nijkamp et al., 2023; Fried et al., 2023). In particular, general purpose LLMs (Ouyang et al., 2022; OpenAI, 2023) have shown remarkable capabilities of translating between natural language and code. However, there is no guarantee that the generated code is error-free. Benchmarks have been suggested to evaluate LLMs on the code generation quality (Chen et al., 2021; Liu et al., 2023). Code generation with execution is highly relevant to our work. Cobbe et al. (2021) and Li et al. (2022) used majority voting on the execution results to select code from a pool of candidates. but this is prone to favoring common wrong solutions over correct solutions. Ni et al. (2023) suggested a ranking mechanism using a learned verifier to assess code correctness. CLAIRIFY (Skreta et al., 2023) implemented automatic iterative prompting that catches errors and provides feedback to the LLM until all issues are resolved.

Tangentially related fields are Automated Program Repair (Xia & Zhang, 2022; Xia et al., 2022), unit test generation (Roziere et al., 2022), and planning for code generation (Le et al., 2022; Zhang et al., 2023). While orthogonal to our approach of iteratively generating code using a pre-trained general-purpose LLM as an optimizer, findings from these fields may be compatible with LangProp.

2.2 LLMS FOR AUTOMATING COMPOSITIONAL TASKS

LLM-powered agents have demonstrated sophisticated planning capabilities. Sequential prompting with the history of observation, action, and the reason for the action was proposed by ReAct (Yao et al., 2023) as an improvement to Chain-of-Thought prompting (Wei et al., 2022). Auto-GPT (Richards, 2023) automated tasks by iteratively generating a sequence of subtasks in finer detail until they are executable. SayCan (Ahn et al., 2022) used LLMs to generate candidate subgoals and assessed their affordances with a value function given visual observations to ground the agent's behavior. VIMA (Jiang et al., 2023) and PaLM-E (Driess et al., 2023) demonstrated profound reasoning and execution capabilities on multi-modal tasks such as Visual Q&A and robotics by fine-tuning LLMs to allow multi-modal prompting. Inner Monologue (Huang et al., 2023) used environment and user feedback to replan for embodied tasks. Unlike our method, the above methods require an LLM in the loop during inference, whereas our method only requires access to an LLM during the code optimization stage. Liang et al. (2023) and Singh et al. (2023) used LLMs to directly generate code for robotics, while ViperGPT (Dídac et al., 2023) and VisProg (Gupta & Kembhavi, 2023) composed pre-trained vision-and-language models to solve challenging vision tasks which require reasoning and domain knowledge. However, none of the above methods implement code optimization via iterative prompting.



```
def main(inputs: dict):
    prediction = {}
    for k, v in inputs.items():
        ...
    return prediction
```

```
def main(inputs: dict):
    prediction = {}
    for k, v in inputs.items():
        ...
    return prediction
```

Figure 1: An overview of the LangProp framework, which consists of a LangProp model, an LLM optimizer, and a LangProp trainer. During training, the LLM generates and updates the policy scripts which are evaluated against a training objective. The performances of the policies are monitored and aggregated over time by a policy tracker as *priorities*, which is then used to rerank the policies. Policies with higher priorities are selected for updates, and the best policy is used for inference.

Our method is inspired by VOYAGER (Wang et al., 2023), which integrates environment feedback, execution errors, and self-verification into an iterative prompting mechanism for embodied control in Minecraft. VOYAGER maintains a *skill library*, a collection of verified reusable code, which can be considered as *checkpoints*. However, there is no mechanism to optimize or remove a sub-optimal skill in the skill library. We address this limitation and present a more general code optimization framework that can be applied to a variety of domains, including autonomous driving.

3 THE LANGPROP FRAMEWORK

The LangProp framework, shown in Figure 1, addresses a general task of optimizing code on a given metric of success in a data-driven way, similar to how a neural network is optimized on an objective function. LangProp performs iterative prompting to improve code performance, using the inputs, outputs, exceptions, metric scores, and any environmental feedback to inform the LLM upon updates. The updates in LangProp are performed using a form of an evolutionary algorithm (Bäck & Schwefel, 1993). The following sections describe the key concepts in LangProp in more detail.

3.1 MODEL DEFINITION

The LangProp model consists of a setup prompt, an update prompt, and a collection of executable code generated by the LLM, which we refer to as *policies*. While neural models are parameterized by floating-point weights, the *parameters* of a LangProp model is the set of policies. Each policy is associated with an executable *script* as well as a statistics tracker, which updates the *priority*, an aggregate measure of the policy's performance with respect to the training objective. The priority is used to rerank policies so that the best-performing policies are used for updates and inference.

3.1.1 POLICY SETUP

The initialization of the policies is done similarly to zero-shot code generation. The definition and specification of the requested function are given as a docstring of the function, including the names and types of the inputs and outputs, what the function is supposed to achieve, and a template for the function. We also adopt Chain-of-Thought prompting (Wei et al., 2022). Examples of setup prompts can be found in Appendix A.1. Responses from the LLM are parsed to extract the solution code snippets. Multiple responses are collected to ensure the diversity of the initial policies.

3.1.2 TRAINING OBJECTIVE

The difference of LangProp over typical usage of LLMs for code generation is that it performs code optimization in a metric- and data-driven manner. In many tasks, it is easier to provide a dataset of inputs and ground truth corresponding outputs rather than to accurately specify the requirements for a valid solution or write comprehensive unit tests. Similar to how neural networks are trained, the user defines an objective function that measures how accurate the policy prediction is against the ground truth, e.g. L1 or L2 loss. A penalty is given if the policy raises an exception.

3.1.3 FORWARD-PASS AND FEEDBACK

Similar to training neural networks, LangProp assumes a dataset of inputs and associated ground truth labels for supervised learning (or rewards/returns for reinforcement learning, discussed in Section 4.3.3). For every batch update, the inputs are fed into all the policies currently in the LangProp model to make predictions, equivalent to a *forward-pass*. For each policy, the prediction is evaluated by the objective function which returns a *score*. If an exception is raised during execution of a policy script, it is caught by the model and an exception penalty is returned as a score instead.

The execution results, which include the score, exception trace, and any printed messages from the execution, are fed back into the model and are recorded by the policy tracker. This is analogous to how neural network parameters are assigned gradients during back-propagation (see Appendix A.9). This information stored by the tracker is used in the policy update step in Section 3.1.5.

3.1.4 PRIORITY

The priority is, simply put, an average of scores with respect to the training objective. In case a small batch size is required for faster computation, a running average of the scores is used as the priority rather than ranking the policies' performance based on scores from the current batch alone, which may result in highly stochastic results. This is sufficient for supervised learning with a fixed-size dataset. As discussed later in Section 4.3.3, however, a more complex training method such as reinforcement learning or DAgger (Ross et al., 2011) has a non-stationary training distribution. Therefore, we use exponential averaging with a discount factor of $\gamma \in (0, 1]$ following Equation (1).

$$P_{i,k} = \left(\sum_{j=1}^{N_k^B} s_{i,j,k} + W_{i,k-1} P_{i,k-1} \right) / \left(N_k^B + W_{i,k-1} \right), \quad W_{i,k} = \gamma (N_k^B + W_{i,k-1}) \tag{1}$$

Here, N_k^B, $P_{i,k}$ and $W_{i,k}$ are the batch size, priority, and priority weighting of the k-th batch for the i-th policy, respectively, and $s_{i,j,k}$ is the objective score of the i-th policy for the j-th element in the k-th batch. Initial conditions are $P_{i,0} = 0$ and $W_{i,0} = 0$. By weighting recent scores higher, we ensure policies with higher priorities have high performance on the most up-to-date dataset.

3.1.5 POLICY RERANKING AND UPDATE

This step updates the model based on the most recent forward-backward pass and updated priorities. This corresponds to the optimization step in neural network training, where parameters are updated based on gradients computed on the most recent batch. First, the policies are reranked by the priorities and the top N^K number of policies are kept, out of which the top N^U policies are selected for updates. For each of these policies, the policy tracker is queried for the worst-case input-output pairs in the training batch, namely that with the lowest objective score. The tracker returns the corresponding input, output and score, along with any exception or print messages during the execution. This information, together with the old policy script, is embedded into the update prompt by a prompt template engine (Section 3.2). The update prompt is passed to the LLM, which returns N^R responses containing new policy scripts for each of the N^U policies chosen for updates.

After the update, there are $N^U \times N^R$ new policies and up to N^K old policies. To initialize the new policies with sensible priorities, objective scores for the new policies are evaluated by performing the forward-backward pass, using the same training samples as the current update. Finally, all the policies are sorted by their priorities, ready for inference or training on a new batch.

3.2 PROMPT TEMPLATE ENGINE

During the policy update stage, we require a dynamic prompting mechanism to embed information about the input, predicted output, ground truth, exception, print messages, and the policy script to be revised. The logic to generate these prompts is sometimes complex, for example, predictions are only made when there are no exceptions. To enable flexible prompt generation while avoiding any hardcoding of the prompts in the codebase, we developed a simple yet powerful prompt template that can parse variables, execute Python code embedded within the prompt, and import sub-prompts from other files, and will be included in our open-sourced solution. The update prompt examples shown in Appendix A.2 make extensive use of the policy template engine's capabilities.

3.3 Training paradigm

LangProp mirrors the code abstraction of PyTorch (Paszke et al., 2019) and PyTorch Lightning (Falcon, 2019) for the module and trainer interfaces. This allows LangProp to be task-agnostic, making it easily applicable to a range of domains and use cases. Moreover, it helps highlight the similarities between neural network optimization and code optimization using LangProp and facilitates a smooth integration of other neural network training paradigms.

Importantly, LangProp's internal implementation does not depend on PyTorch or PyTorch Lightning. LangProp supports PyTorch datasets and data loaders, as well as any iterable dataset object for training and validation. Listing 1 shows an example of a standard LangProp training script.

```python
train_loader = DataLoader(train_data, batch_size, shuffle=True, collate_fn=lambda x: x)
val_loader = DataLoader(val_data, batch_size, shuffle=True, collate_fn=lambda x: x)
model = LPModule.from_template(name, root)
trainer = LPTrainer(model, RunConfig(run_name))
trainer.fit(train_loader, val_loader, epochs=epochs)
```

Listing 1: Training a LangProp model with a trainer. The model can be instantiated from a path to the setup and update prompts that specify the task to be learned.

After every training step on a mini-batch, the trainer saves a *checkpoint*, which consists of the setup prompt, update prompt template, the policy scripts (maximum of $N^K + N^U \times N^R$), and the statistics monitored by the policy tracker (priorities P and priority weights W). Since these can be stored as text or JSON files, the size of a checkpoint is in the order of a few hundred kilobytes. Checkpoints can be used to resume training, fine-tune the model, or for inference.

```python
model = LPModule.from_checkpoint(checkpoint)
model.setup(config=RunConfig())
prediction = model(*input_args, **input_kwargs)
```

Listing 2: Inference with a LangProp model checkpoint.

Listing 2 shows how a LangProp checkpoint can be loaded and used for inference. The policy with the highest priority is used. Since policies are *parameterized* as executable code, the use of an LLM is only required during training, not during inference. Since querying LLMs is both expensive and slow, this is a key advantage of the LangProp approach, which makes integration of LLMs more feasible for real-time applications, such as robotics and autonomous driving.

4 Experiments

We demonstrated LangProp's code optimization capability in three domains with increasing complexity. For the LLM, we used GPT 3.5 Turbo 16k model (OpenAI, 2022).

4.1 Generalized Sudoku

A generalized Sudoku puzzle consists of $W \times H$ subblocks, each with $H \times W$ elements, where H and W represent height and width, respectively. A valid solution places numbers from 1 to WH in each cell, such that each row, column and subblock contains no repeated numbers. We trained LangProp on this problem given 100 samples of unsolved Sudoku puzzles as input and corresponding solutions as output. We define the training objective to be the correctness of the arrived solution, i.e. whether the puzzle completed by a LangProp-learned policy is a valid Sudoku puzzle solution. The setup and update prompts are in Appendix A. Due to the complexity of the task specification, we found that the LLM queried zero-shot occasionally failed on the first attempt, confusing the task with a standard 3×3 Sudoku. LangProp filtered out incorrect results during training and identified a fully working solution. Samples of an incorrect zero-shot solution and a correct solution after LangProp training can be found in Appendix F.1.

4.2 CartPole

CartPole Brockman et al. (2016) is a widely used environment for RL. To make it feasible for LangProp to solve this task, we provided the observation and action specifications, available in the Gymnasium documentation for CartPole-v1. The setup and update prompts are in Appendix A. Queried zero-shot, the LLM generated a solution that is simplistic and does not balance the CartPole, achieving a score of 9.9 out of 500. With a simple Monte-Carlo method of optimizing the policy for the total rewards, we obtained improved policies using LangProp, achieving the maximum score of 500.0. Interestingly, LangProp learned a policy that implemented a PID controller to solve the task.

Figure 2 shows learning curves of the LangProp policy for 10 different seeds. We chose training hyperparameters to be $N^U = N^R = N^K = 3$. Out of 10 seeds, 9 converged to an optimal solution within 10 LangProp updates, and within $10k$ total steps in the CartPole environment. For comparison, we also plotted learning curves of PPO Schulman et al. (2017), a widely used reinforcement learning algorithm, which converges at around $80k$ environment steps. This shows that certain tasks may be more sample-efficient to solve with LangProp. While it is infeasible to arrive at a correct solution zero-shot, the LangProp optimization loop allows the LLM to discover a correct solution.

Sample results can be found in appendix F.2. Implementations, prompts, checkpoints, and examples of zero-shot and trained policies are available in the open-sourced repository.

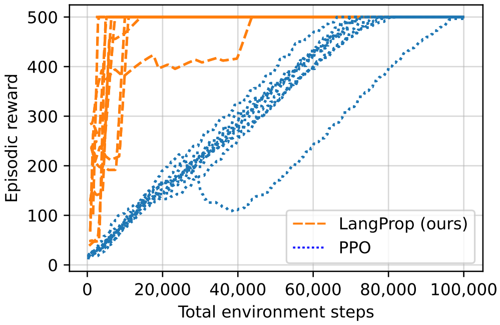


Figure 2: The total number of *environment* steps required to learn CartPole-v1 (10 seeds per method) in comparison to a RL method (PPO). Most seeds converged to an optimal solution within 10 LangProp updates.

4.3 Driving in CARLA

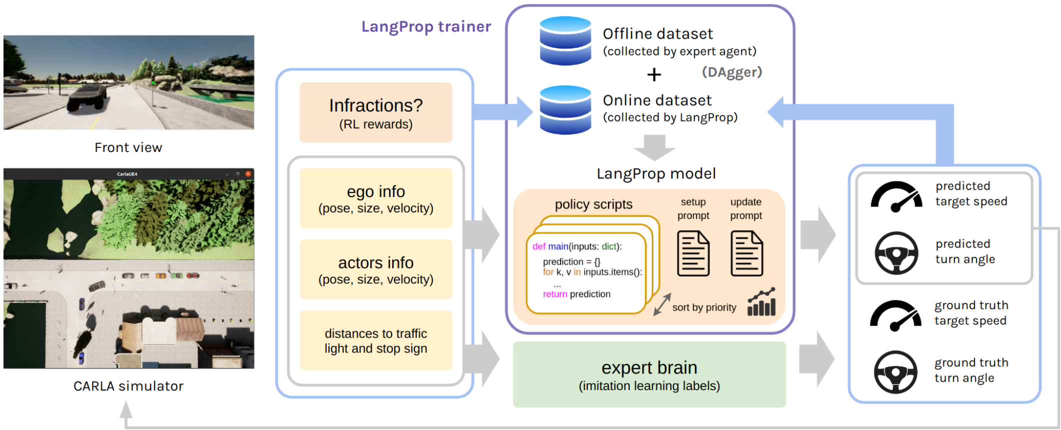
In this section, we describe how the LangProp framework can be used in the context of autonomous driving in CARLA. CARLA (Dosovitskiy et al., 2017) is a widely used open-sourced 3D simulator for autonomous driving research, and many prior works on CARLA have open-sourced their expert agents. We chose CARLA as a benchmark since (a) autonomous driving requires interpretable driving policies, (b) CARLA has a rich collection of human-implemented expert agents to compare against, and (c) a metric-driven learnable approach would be beneficial since driving decisions are challenging planning problems, and even human-implemented experts have sub-optimal performance. Appendix B discusses related work on autonomous driving.

4.3.1 Expert design

We implemented our expert agent for data collection and to provide action labels to train the LangProp agent with IL. While TransFuser (Chitta et al., 2022) and TF++ (Jaeger et al., 2023) use a computationally expensive 3D bounding box collision detection algorithm, and InterFuser (Shao et al., 2023) uses line collision which is faster but less accurate, we use an efficient polygon collision detection algorithm between ground-projected bounding boxes. Safety margins to pedestrians and vehicles are calculated by extrapolating the motion of those actors into the future and checking for any polygon intersections. The target speed is determined to give a 2 s margin to any actors in collision course, traffic light and/or stop sign. Steering is evaluated by calculating the angle to a waypoint 4 m ahead of the ego vehicle. A PID controller is used for low-level control to convert the target speed and angle to throttle, brake, and steering. For more details, see Appendix C.2.

4.3.2 LangProp agent

Similarly to all the baseline experts, we provide privileged information from the CARLA simulator to the agent. Unlike the baseline experts where post-processing is manually implemented, we let LangProp decide how the information should be handled (e.g. converting world coordinates into the ego-centric frame). For the ego vehicle, as well as for all vehicles and pedestrians within a 50 m radius, we provide the location (in world coordinates), orientation, speed, length, and width of the actor. Importantly, we do not filter out actors even if they are irrelevant to the driving agent. We also provide the target waypoint (4 m ahead, used by other baseline experts) and the distances to a red traffic light and stop sign along the current lane if they exist. Given this information, the LangProp policy is expected to return a desired speed level ("MOVE": 6 m/s, "SLOW": 1 m/s, "STOP":



```
def main(inputs: dict):
    prediction = {}
    for k, v in inputs.items():
        ...
    return prediction
```

Figure 3: An overview of the LangProp agent training pipeline. The LangProp model is updated on a dataset that includes both offline expert data as well as online LangProp data annotated with expert actions, similar to DAgger. The agent is given negative rewards upon infraction.

0 m/s)[1] and a turning angle for the ego vehicle. These are passed to an external PID controller to convert them into throttle, brake, and steering. The function specification in the setup prompt is given in Listing A.5 as a docstring. Given this function definition, an LLM generates policy script candidates that satisfy the specification and updates them following the procedures in Section 3.

4.3.3 Imitation Learning, DAgger, and RL

We explore three major training paradigms often used to train embodied agents - IL, DAgger (Ross et al., 2011), and RL. In IL, the accuracy of the policy outputs is measured against ground truth expert actions for a pre-collected dataset. IL is known to have issues with out-of-distribution inputs at inference time, since the expert's policy is used to collect the training data, while the learned policy is used for rollouts at inference time. DAgger addresses this issue by labeling newly collected *online* data with expert actions, and adding them to the expert-collected *offline* data to form an aggregate replay buffer. CARLA runs at a frame rate of 20 Hz. LangProp adds training samples to the replay buffer every 10 frames, and a batch update is performed after every 100 new samples.

While DAgger solves the issue of distribution mismatch, the performance of the learned policy is still upper-bounded by the accuracy of the expert. It also does not take into account that certain inaccuracies are more critical than others. In the context of autonomous driving, actions that result in infractions such as collisions should be heavily penalized. Reinforcement learning offers a way of training a policy from reward signals from the environment, which is convenient since we can directly assign penalties upon any infractions according to the CARLA leaderboard (CARLA, 2020). While RL typically optimizes for maximum returns (discounted sum of future rewards), we simplify the setting by assigning an infraction penalty if there is an infraction in the next 2 s window. The agent monitors infractions every 10 frames, and triggers an update upon infractions.

Since infraction penalties are sparse signals, and will become rarer as the policies improve, we adopt two strategies; (a) we combine RL with IL to provide denser signals, and (b) we sample training data with infractions with 100 times higher sampling probability. The expert is only imitated upon no infractions, or if the expert disagrees with the behavior policy which incurred the infraction. An infraction cost is only given when the current policy takes the same action as the behavioral policy that caused the infraction and disagrees with the expert. Details on the objective are in Appendix D.2.

4.3.4 Baselines

We compared the LangProp agent against RL agents with privileged information (Roach (Zhang et al., 2021), TCP (Wu et al., 2022)) as well as human-written experts (TransFuser (Chitta et al., 2022), InterFuser (Shao et al., 2023), TF++ (Jaeger et al., 2023), ours). We used the official training

[1] While it is straightforward for the policy to directly predict the speed or acceleration as numeric values, this makes the task of designing a suitable loss function for IL more challenging and open-ended. Therefore, we opted for a categorical output which simplifies the scoring function.

Table 1: Driving performance of expert drivers in CARLA. The driving score is a product of the route completion percentage $\bar{R}$ and infraction factor $\bar{I}$. DAgger uses both online and offline data.

Method	**Training routes**			**Testing routes**			**Longest6**		
	Score ↑	$\bar{R}$ ↑	$\bar{I}$ ↑	**Score** ↑	$\bar{R}$ ↑	$\bar{I}$ ↑	**Score** ↑	$\bar{R}$ ↑	$\bar{I}$ ↑
Roach expert	57.8	95.9	0.61	63.4	98.8	0.64	54.9	81.7	0.67
TCP expert	64.3	92.3	0.71	72.9	93.2	0.77	46.9	63.1	0.76
TransFuser expert	69.8	94.5	0.74	73.1	91.3	0.80	70.8	81.2	0.88
InterFuser expert	69.6	83.1	0.86	78.6	81.7	0.97	48.0	56.0	0.89
TF++ expert	**90.8**	95.9	0.94	86.1	91.5	0.94	**76.4**	84.4	0.90
Our expert	88.9	92.8	0.95	**95.2**	98.3	0.97	72.7	78.6	0.92
LangProp: Offline IL	0.07	0.37	0.97	0.00	0.00	1.00	0.00	0.00	1.00
LangProp: DAgger IL	36.2	94.5	0.40	41.3	95.3	0.44	22.6	87.4	0.30
LangProp: DAgger IL/RL	64.2	90.0	0.72	61.2	95.2	0.64	43.7	71.1	0.65
LangProp: Online IL/RL	**70.3**	90.5	0.78	**80.9**	92.0	0.89	**55.0**	75.7	0.73

and testing routes provided by the CARLA leaderboard (CARLA, 2020), as well as the Longest6 benchmark (Chitta et al., 2022) that has longer routes with denser traffic. For the LangProp agent, only the training routes are used for imitation/reinforcement learning at training time, and the saved checkpoints are used for inference during evaluation runs on different routes. See Appendix E.1 for more details on the benchmark and the routes and towns used. The results are shown in Table 1.

4.3.5 Results

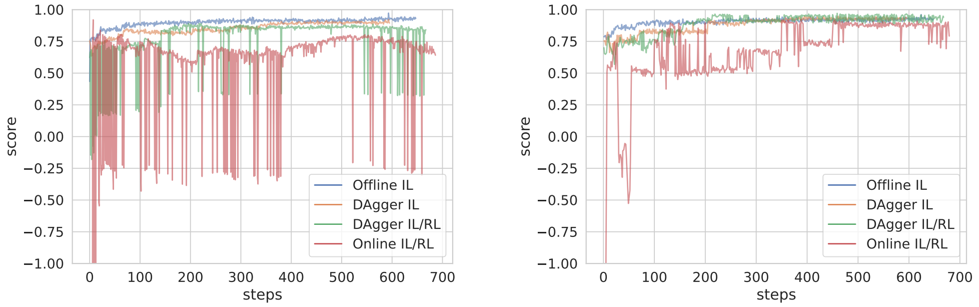
Our expert and the TF++ expert significantly outperformed all other expert agents in all routes, and our expert outperformed TF++ by a margin on the test routes. The core collision avoidance logic is just 100 lines of code, with additional preprocessing and tooling for data collection. From the breakdown of the scores, our expert seems to prioritize safer driving with fewer infractions (higher infraction factor $\bar{I}$) by trading off route completion compared to TF++ in the Longest6 benchmark.

For the LangProp agent, we observe that training using offline samples, DAgger, and online samples improves performance in this order. Adding the infraction penalties as an additional reinforcement learning objective further improved the performance. The best-performing agent, LangProp trained on online data with IL and RL, achieved better performance than the Roach expert (trained with PPO) as well as the TransFuser and InterFuser experts (both written by researchers) on all benchmarks apart from TransFuser on the Longest6 benchmark. Note that TransFuser has an advantage over the Longest6 benchmark since LangProp has never seen this benchmark during training. The driving policy generated using LangProp is shown in appendix F.3.

The result has two important implications. Firstly, the code selection metric (the training objective) plays a large role in the ultimate performance of the code. This is an important finding since prior work on code generation mostly focused on error correction given exceptions. Our results demonstrate that for complex tasks, it is important to treat code generation as an iterative optimization process rather than a zero-shot task. Secondly, training using LangProp exhibits similar characteristics as training in deep learning; in deep learning, it is a well-studied problem that policies trained with IL on offline datasets do not generalize to out-of-distribution online data. DAgger and reinforcement learning are two of the common ways of addressing this problem. Our results show that these training paradigms can also be effective when used in LangProp.

4.3.6 Analysis of training methods

A common failure mode of offline trained models was that the agent remained stationary indefinitely until the timeout was reached. Upon inspection of the policy code that was generated, we were able to identify the failure to be a phenomenon known as causal confusion in IL (De Haan et al., 2019). A snippet of code responsible for such failure in one of the runs is shown in Listing 3.



(a) training scores on the replay buffer

(b) validation scores on the offline dataset

Figure 4: Training curves for the different training methods of the LangProp agent. The training scores are evaluated on 1000 samples from the offline training dataset and/or online replay buffer, and the validation scores are evaluated on 1000 samples from the offline validation dataset. Updates are performed every 1000 frames of agent driving, and upon infractions in the RL setting. The score is in the range of $[-10, 1]$ due to exception penalties. We limit the axis to $[-1, 1]$ in the plots.

This exemplifies the interpretability of LangProp models, allowing us to directly assess the source of failure. The code predicts 0 speed when the agent's current speed is already close to 0. Note that this is not a failure of LangProp, but due to such a policy maximizing the IL objective on an offline dataset, bypassing the need to learn a more complex policy. This phenomenon is also common in the context of *deep* IL, and can be avoided by employing training on online data, e.g. DAgger or RL. We believe our work to be the first to report a similar phenomenon using LLMs for policy optimization.

```python
# General rule: if the ego vehicle is stopped or moving very slowly, set the speed level to "STOP"
if np.abs(scene_info["ego_forward_speed"]) < DELTA_V_THRESHOLD:
    speed_level = "STOP"
```

Listing 3: Causal confusion in offline-trained policy

The use of online training samples alleviated the issue of causal confusion, leading to selecting policies where the agent has a sensible driving performance. This is because if the agent remains stationary, those samples will accumulate in the replay buffer, resulting in a lower priority for the causally confused policy. Comparing the results in Table 1 and the validation scores in Figure 4b, it seems that the scores on the offline dataset are not indicative of the agent's driving performance. From the training scores on the replay buffer and/or offline dataset in Figure 4a, we see that the agents trained with RL on infractions have spikes corresponding to infractions. This is due to oversampling infractions when they occur, allowing the policy update to immediately address the issue. DAgger has a milder response compared to training just on online data because the offline dataset does not include on-policy infractions. The higher rate of infractions in the training distribution may be why the online trained agent has a lower training score but has a higher driving performance.

5 Conclusion

We presented LangProp, a framework that uses LLMs for data-driven code optimization, and demonstrated its capability of generating and improving policies in the domains of Sudoku, CartPole and CARLA. In particular, LangProp generated driving policies in CARLA that outperform those that existed when the backbone GPT 3.5 was trained. We showed that classical training paradigms such as IL, DAgger, and RL directly translate to training with LangProp, and the choices of the objective function and the training data distribution can be used to guide which policies are selected. Automatically optimizing the code to maximize a given performance metric has been a key missing feature in few-shot code generation. The LangProp framework provides this feature by reformulating the machine learning training paradigm in the context of using LLMs as code optimizers and treating policy code as parameters of the model. We believe that the LangProp paradigm opens up many possibilities for data-driven machine learning with more interpretability and transparency.

Acknowledgments

The authors would like to thank Oleg Sinavski, Long Chen, Benoît Hanotte, Ana-Maria Marcu, Jan Hünermann and Corina Gurau for insightful research discussions and assistance with compute resources.

References

Bassem Abu-Nasser. Medical expert systems survey. *International Journal of Engineering and Information Systems (IJEAIS)*, 1(7):218–224, 2017.

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation*, 1(1):1–23, 1993.

Claudine Badue, Rânik Guidolini, Raphael Vivacqua Carneiro, Pedro Azevedo, Vinicius B Cardoso, Avelino Forechi, Luan Jesus, Rodrigo Berriel, Thiago M Paixao, Filipe Mutz, et al. Self-driving cars: A survey. *Expert Systems with Applications*, 165:113816, 2021.

Mayank Bansal, Alex Krizhevsky, and Abhijit Ogale. Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst. *arXiv preprint arXiv:1812.03079*, 2018.

Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars, 2016.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020.

CARLA. Carla autonomous driving leaderboard. `https://leaderboard.carla.org/`, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

Kashyap Chitta, Aditya Prakash, Bernhard Jaeger, Zehao Yu, Katrin Renz, and Andreas Geiger. Transfuser: Imitation with transformer-based sensor fusion for autonomous driving. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Pim De Haan, Dinesh Jayaraman, and Sergey Levine. Causal confusion in imitation learning. *Advances in Neural Information Processing Systems*, 32, 2019.

Surís Dídac, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. *arXiv preprint arXiv:2303.08128*, 2023.

Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In Sergey Levine, Vincent Vanhoucke, and Ken Goldberg (eds.), *Proceedings of the 1st Annual Conference on Robot Learning*, volume 78 of *Proceedings of Machine Learning Research*, pp. 1–16. PMLR, 13–15 Nov 2017.

Danny Driess, Fei Xia, Mehdi S. M. Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, Wenlong Huang, Yevgen Chebotar, Pierre Sermanet, Daniel Duckworth, Sergey Levine, Vincent Vanhoucke, Karol Hausman, Marc Toussaint, Klaus Greff, Andy Zeng, Igor Mordatch, and Pete Florence. Palm-e: An embodied multimodal language model, 2023.

William A Falcon. Pytorch lightning. `https://github.com/Lightning-AI/lightning`, 2019.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*, 2023.

Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14953–14962, 2023.

Frederick Hayes-Roth. Rule-based systems. *Communications of the ACM*, 28(9):921–932, 1985.

Anthony Hu, Gianluca Corrado, Nicolas Griffiths, Zachary Murez, Corina Gurau, Hudson Yeo, Alex Kendall, Roberto Cipolla, and Jamie Shotton. Model-based imitation learning for urban driving. *Advances in Neural Information Processing Systems*, 35:20703–20716, 2022.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. In *Conference on Robot Learning*, pp. 1769–1782. PMLR, 2023.

Peter Jackson. Introduction to expert systems. 1986.

Bernhard Jaeger, Kashyap Chitta, and Andreas Geiger. Hidden biases of end-to-end driving models. *arXiv preprint arXiv:2306.07957*, 2023.

Yunfan Jiang, Agrim Gupta, Zichen Zhang, Guanzhi Wang, Yongqiang Dou, Yanjun Chen, Li Fei-Fei, Anima Anandkumar, Yuke Zhu, and Linxi Fan. Vima: General robot manipulation with multimodal prompts. In *Fortieth International Conference on Machine Learning*, 2023.

Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, Alex Bewley, and Amar Shah. Learning to drive in a day. In *2019 International Conference on Robotics and Automation (ICRA)*, pp. 8248–8254. IEEE, 2019.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

Jesse Levinson, Jake Askeland, Jan Becker, Jennifer Dolson, David Held, Soeren Kammel, J Zico Kolter, Dirk Langer, Oliver Pink, Vaughan Pratt, et al. Towards fully autonomous driving: Systems and algorithms. In *2011 IEEE intelligent vehicles symposium (IV)*, pp. 163–168. IEEE, 2011.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9493–9500. IEEE, 2023.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, 2023.

Yiren Lu, Justin Fu, George Tucker, Xinlei Pan, Eli Bronstein, Becca Roelofs, Benjamin Sapp, Brandyn White, Aleksandra Faust, Shimon Whiteson, et al. Imitation is not enough: Robustifying imitation with reinforcement learning for challenging driving scenarios. *arXiv preprint arXiv:2212.11419*, 2022.

Will Maddern, Geoffrey Pascoe, Chris Linegar, and Paul Newman. 1 year, 1000 km: The oxford robotcar dataset. *The International Journal of Robotics Research*, 36(1):3–15, 2017.

Rowan McAllister, Yarin Gal, Alex Kendall, Mark Van Der Wilk, Amar Shah, Roberto Cipolla, and Adrian Weller. Concrete problems for autonomous vehicle safety: advantages of bayesian deep learning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pp. 4745–4753, 2017.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-Tau Yih, Sida Wang, and Xi Victoria Lin. LEVER: Learning to verify language-to-code generation with execution. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 26106–26128. PMLR, 23–29 Jul 2023.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2023.

OpenAI. Chatgpt. `https://openai.com/blog/chatgpt`, 2022.

OpenAI. Gpt-4 technical report, 2023.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 27730–27744. Curran Associates, Inc., 2022.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Toran Bruce Richards. Auto-gpt. `https://github.com/Significant-Gravitas/Auto-GPT`, 2023.

Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 627–635. JMLR Workshop and Conference Proceedings, 2011.

Baptiste Roziere, Jie Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. In *International Conference on Learning Representations*, 2022.

Ahmad EL Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging*, 29:70–76, 2017.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Hao Shao, Letian Wang, Ruobing Chen, Hongsheng Li, and Yu Liu. Safety-enhanced autonomous driving using interpretable sensor fusion transformer. In *Conference on Robot Learning*, pp. 726–737. PMLR, 2023.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 11523–11530. IEEE, 2023.

Marta Skreta, Naruki Yoshikawa, Sebastian Arellano-Rubach, Zhi Ji, Lasse Bjørn Kristensen, Kourosh Darvish, Alán Aspuru-Guzik, Florian Shkurti, and Animesh Garg. Errors are useful prompts: Instruction guided task programming with verifier-assisted iterative prompting. *arXiv preprint arXiv:2303.14100*, 2023.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2018.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, MN Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, et al. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of field Robotics*, 25(8):425–466, 2008.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

Junqing Wei, Jarrod M Snider, Junsung Kim, John M Dolan, Raj Rajkumar, and Bakhtiar Litkouhi. Towards a viable autonomous driving research platform. In *2013 IEEE Intelligent Vehicles Symposium (IV)*, pp. 763–770. IEEE, 2013.

Penghao Wu, Xiaosong Jia, Li Chen, Junchi Yan, Hongyang Li, and Yu Qiao. Trajectory-guided control prediction for end-to-end autonomous driving: A simple yet strong baseline. *Advances in Neural Information Processing Systems*, 35:6119–6132, 2022.

Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 959–971, 2022.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Practical program repair in the era of large pre-trained language models. *arXiv preprint arXiv:2210.14179*, 2022.

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, pp. 1–10, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392730. doi: 10.1145/3520312.3534862.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A survey of autonomous driving: Common practices and emerging technologies. *IEEE access*, 8:58443–58469, 2020.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. Planning with large language models for code generation. In *The Eleventh International Conference on Learning Representations*, 2023.

Zhejun Zhang, Alexander Liniger, Dengxin Dai, Fisher Yu, and Luc Van Gool. End-to-end urban driving by imitating a reinforcement learning coach. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 15222–15232, 2021.

Appendices

REPRODUCIBILITY STATEMENT

We open-source the code used in this paper, including code for the general LangProp framework, applying LangProp to tasks such as Sudoku and CartPole, and training and evaluating the LangProp agent in CARLA. More details of the implementation and design decisions can be found in the appendices.

Supplementary materials can be found at `https://github.com/shuishida/LangProp/`, which includes the code, pre-trained checkpoints using LangProp, and videos of sample runs by the LangProp agent.

A LANGPROP MODEL AND PROMPT DEFINITIONS

LangProp as a framework can be used to optimize a diverse range of code optimization problems. The functionality of the model is determined by the choices in the setup prompt, the update prompt, and the dataset that the LangProp model is trained on.

A.1 POLICY SETUP PROMPT EXAMPLES

We provide simple examples of learning a Sudoku algorithm, and learning a policy that plays CartPole-v1, a widely used reinforcement learning environment in Brockman et al. (2016), to show the generality of the framework. The setup prompt should include the specification of the function's inputs and outputs and their types in the form of a docstring.

~~~
I am developing code to solve a sudoku puzzle. Please write a function which takes a numpy array of an unsolved sudoku puzzle and return a complete solution.

Here is the definition of the function.

```
Given a numpy array of non-negative integers as a starting condition of a sudoku puzzle, the function returns a complete solution, also as a numpy array.
The inputs to the function are the incomplete sudoku numpy array, the width of the sudoku, and the height of the sudoku. Note that the overall numpy array has a shape of (height x width, width x height).
For example, if we are solving a conventional 3x3 sudoku, the width is 3, the height is 3, and the numpy array is 9x9. The constraint of a sudoku puzzle is that, every row, every column, and every block of dimensions (height, width) should contain unique values of 1 to height x width, one each. The unfinished sudoku puzzle is given as a numpy array of integers where some of the values are filled, and other values which are unsolved are 0. The function returns a complete sudoku puzzle as an numpy array of integers.

Args:
    - sudoku: np.ndarray      # shape of (height x width, width x height)
    - width: int
    - height: int

Returns:
    - solution: np.ndarray    # shape of (height x width, width x height)
```


This is a template of the code.

~~~

~~~
```python
def {{ function_name }}(sudoku: np.ndarray, width: int, height: int) -> np.ndarray:
    # Write code here
    return solution
~~~

Please do the following:
Step 1. Describe step by step what the code should do in order to achieve its task.
Step 2. Provide a python code solution that implements your strategy, including all necessary import statements.
```

Listing A.1: Setup prompt template to learn an algorithm to solve generalized Sudoku

```
I am developing code to solve CartPole. Please write a function which takes the position and velocity of the cart, and the angle and angular velocity of the pole, and return the action that the policy should take.

Here is the definition of the function.

```
Given the position and velocity of the cart, and the angle and angular velocity of the pole, return the action that the policy should take to balance the pole on the cart.

Args:
    - cart_position: float          # range of -4.8 to 4.8 [m]
    - cart_velocity: float          # range of -inf to +inf [m/s]
    - pole_angle: float             # range of -0.418 to 0.418 [radian]
    - pole_angular_velocity: float  # range of -inf to +inf [radian/s]

Returns:
    - action: int    # 0 if the cart should be pushed to the left (negative direction), 1 if it should be pushed to the right (positive direction)
```

This is a template of the code.

```python
def {{ function_name }}(cart_position, cart_velocity, pole_angle, pole_angular_velocity) -> int:
    # Write code here
    return action
```

Please do the following:
Step 1. Describe step by step what the code should do in order to achieve its task.
Step 2. Provide a python code solution that implements your strategy, including all necessary import statements.
```

Listing A.2: Setup prompt template to learn an agent policy to play CartPole

A.2 POLICY UPDATE PROMPT EXAMPLE

The prompt used to update the policy contains the same information as the setup prompt, but in addition, has example inputs and outputs where the code had failed to produce a valid prediction. If there was an exception or printed messages during the execution of the code, this will also be provided as feedback. The LLM is asked to identify the source of the sub-optimal performance and rewrite the code to achieve a higher score.

```
I am developing code to solve a sudoku puzzle. Please write a function which takes a numpy array of an unsolved sudoku puzzle and return a complete solution.

Here is the definition of the function.

```
Given a numpy array of non-negative integers as a starting condition of a sudoku puzzle, the function returns a complete solution, also as a numpy array.
The inputs to the function are the incomplete sudoku numpy array, the width of the sudoku, and the height of the sudoku.
Note that the overall numpy array has a shape of (height x width, width x height).
For example, if we are solving a conventional 3x3 sudoku, the width is 3, the height is 3, and the numpy array is 9x9.
```

~~~
The constraint of a sudoku puzzle is that, every row, every column, and every block of dimensions (height, width) should contain unique values of 1 to height x width, one each.
The unfinished sudoku puzzle is given as a numpy array of integers where some of the values are filled, and other values which are unsolved are 0.
The function returns a complete sudoku puzzle as an numpy array of integers.

Args:
    - sudoku: np.ndarray      # shape of (height x width, width x height)
    - width: int
    - height: int

Returns:
    - solution: np.ndarray    # shape of (height x width, width x height)
```


Here is an example code that I have written. However, it is not working as expected.

```python
{{ code }}
```

I executed the code, and got an accuracy of {{ int(avg_score * 100) }}%.

$begin
if printed:
    print("There was a print message saying: {{ printed }}")
if exception:
    print("""The code failed to run because there was an exception. The exception message was as follows: {{ exception }}""")
    print("Resolving this exception is the top priority.")
else:
    if {{ same_outputs }}:
        print("""It also seems that the code produces the same output of ```{{ outputs }}``` for all inputs, which is not the behaviour we want.""")

    print("""

The code produced incorrect results for the following inputs. The prediction, ground truth label and score were as follows.

Inputs: sudoku = {{ args[0] }}, width = {{ args[1] }}, height = {{ args[2] }}
Incorrect prediction: solution = {{ outputs }}""")
    if {{ label is not None }}:
        print("""Ground truth label: solution = {{ label[0] }}""")
    print("Score: {{ int(score * 100) }}%")
$end

$begin
if feedback:
    print("""{{ feedback }}""")
$end

Please do the following:

$begin
if exception:
    print("Step 1. Look at the error message carefully and identify the reason why the code failed, and how it can be corrected.")
else:
    print("Step 1. Given the example input and output, identify the reason why the code made a wrong prediction, and how it can be corrected to achieve a good score.")
$end

Step 2. Describe step by step what the code should do in order to achieve its task.
Step 3. Please rewrite the python function `{{ function_name }}` to achieve a higher score, including all necessary import statements.

~~~

Listing A.3: Update prompt template to learn an algorithm to solve generalized Sudoku

```
I am developing code to solve CartPole. Please write a function which takes the position and velocity of the cart, and the angle and angular velocity of the pole, and return the action that the policy should take.

```

```
Here is the definition of the function.

```
Given the position and velocity of the cart, and the angle and angular velocity of the pole, return the action that the policy should take to balance the pole on the cart.

Args:
    - cart_position: float   # range of -4.8 to 4.8 [m]
    - cart_velocity: float   # range of -inf to +inf [m/s]
    - pole_angle: float   # range of -0.418 to 0.418 [radian]
    - pole_angular_velocity: float # range of -inf to +inf [radian/s]

Returns:
    - action: int   # 0 if the cart should be pushed to the left (negative direction), 1 if it should be pushed to the right (positive direction)
~~~

Here is an example code that I have written. However, it is not working as expected.

```python
{{ code }}
~~~

I executed the code, but the performance was not very high. I got a score of {{ int(avg_score) }} where a good score is 500.

$begin
if printed:
    print("There was a print message saying: {{ printed }}")
if exception:
    print("""The code failed to run because there was an exception. The exception message was as follows: {{ exception }}""")
    print("Resolving this exception is the top priority.")
else:
    if {{ same_outputs }}:
        print("It also seems that the code produces the same output of ```{{ outputs }}``` for all inputs, which is not the behaviour we want.")
$end

$begin
if feedback:
    print("""{{ feedback }}""")
$end

Please do the following:

$begin
if exception:
    print("Step 1. Look at the error message carefully and identify the reason why the code failed, and how it can be corrected.")
else:
    print("Step 1. Given the example input and output, identify the reason why the code made a wrong prediction, and how it can be corrected to achieve a good score.")
$end

Step 2. Describe step by step what the code should do in order to achieve its task.
Step 3. Please rewrite the python function `{{ function_name }}` to achieve a higher score, including all necessary import statements.

Listing A.4: Update prompt template to learn an agent policy to play CartPole

### A.3 MODEL FORWARD PASS DEFINITION

The LangProp module captures printed outputs and exceptions and stores them in the policy tracker along with the corresponding inputs during a forward pass. The Python code snippet extracted from the LLM's response and saved as a text string is executed using the `exec` function in Python. The local scope variables can be accessed via `locals`.

```python
class LPModule:
    ...

    def __call__(self, *args, **kwargs) -> Any:
        if not self.training:
            return self.forward(self.script_records[0].script, *args, **kwargs)
```

```

        inputs = (args, kwargs)
        script = self.run_config.active_tracker.record.script
        with CapturePrint() as p:
            try:
                output = self.forward(script, *args, **kwargs)
                self.run_config.active_tracker.forward(inputs, output, "\n".join(p))
            except KeyboardInterrupt as e:
                raise e
            except Exception as e:
                trace = "\n".join(traceback.format_exc().split('\n')[-3:])
                detail = f"""{type(e).__name__}: {trace}"""
                self.run_config.active_tracker.store_exception(inputs, e, detail,
                ↪  "\n".join(p))
                raise e
        return output

    def forward(self, script, *args, **kwargs):
        exec(script, locals(), locals())
        output = locals()[self.name](*deepcopy(args), **deepcopy(kwargs))
        return output
```

Listing A.5: Forward passing mechanism of the LangProp module (extract)

A.4 Trainer forward-backward definition

The trainer has a similar abstraction to deep learning training. At every step, it triggers a forward method that calls the policy and stores the inputs, the policy's prediction, and the expected output, and a backward method that updates the policy tracker with the scores, exceptions, or any feedback.

```
class LPTrainer:
    ...

    def step(self, tracker: RecordTracker, func_args, func_kwargs, label, feedback=""):
        with self.run_config.activate(tracker):
            score, exception_detail = self.forward(func_args, func_kwargs, label)
            tracker.backward(score, label, feedback + exception_detail)

    def forward(self, func_args, func_kwargs, label):
        try:
            with set_timeout(self.run_config.forward_timeout):
                output = self.module(*func_args, **func_kwargs)
            self.test_output(output, func_args, func_kwargs, label)
            score = self.score(output, label)
            exception_detail = ""
        except KeyboardInterrupt as e:
            raise e
        except Exception as e:
            score = self.run_config.exception_score
            trace = "\n".join(traceback.format_exc().split('\n')[-3:])
            exception_detail = f"""\nThere was an exception of the
            ↪  following:\n{type(e).__name__}: {trace}"""
        return score, exception_detail
```

Listing A.6: Forward-backward pass in the LangProp Trainer (extract)

A.5 Policy definition for the LangProp driving agent in CARLA

The driving policy is given the location, orientation, speed, length, and width of the ego vehicle, other vehicles and pedestrians in the scene, the distances to the next red traffic light and stop sign, and the target waypoint ($4\ m$ ahead, used by other baseline experts), all in absolute world coordinates.

```
```
Args:
    - scene_info: dict
        Contains the following information:
        {
            "ego_location_world_coord": np.ndarray,   # numpy array of shape (2,)
            ↪ which contains (x, y) of the center location of the ego vehicle in world
            ↪ coordinates given in [m]
            "ego_target_location_world_coord": np.ndarray, # numpy array of shape (2,)
            ↪ which contains (x, y) of the target location of the ego vehicle in world
            ↪ coordinates given in [m]
```
```

~~~
            "ego_orientation_unit_vector": np.ndarray,      # numpy array of shape (2,)
            ↪  which contains (x, y) of unit vector orientation of the ego vehicle in
            ↪  world coordinates. The vehicle moves in the direction of the orientation.
            "ego_forward_speed": float,                      # the speed of the ego vehicle
            ↪  given in [m/s].
            "ego_length": float,                             # length of the ego vehicle in
            ↪  the orientation direction, given in [m/s].
            "ego_width": float,                              # width of the ego vehicle
            ↪  perpendicular to the orientation direction, given in [m].
            "distance_to_red_light": Union[float, None],     # distance to red light given
            ↪  in [m]. None if no traffic lights are affecting the ego vehicle
            "distance_to_stop_sign": Union[float, None],     # distance to stop sign given
            ↪  in [m]. None if no stop signs are affecting the ego vehicle
            "vehicles": {                       # dictionary of nearby vehicles
                <vehicle_id: int>:  {
                    "location_world_coord": np.ndarray,      # numpy array of shape (2,)
                    ↪  which contains (x, y) of the center location of vehicle
                    ↪  <vehicle_id> in world coordinates given in [m]
                    "orientation_unit_vector": np.ndarray,   # numpy array of shape (2,)
                    ↪  which contains (x, y) of unit vector orientation of vehicle
                    ↪  <vehicle_id> in world coordinates. The vehicle moves in the
                    ↪  direction of the orientation.
                    "forward_speed": float,                  # speed of vehicle <vehicle_id>
                    ↪  given in [m/s].
                    "forward_length": float,                 # length of the vehicle
                    ↪  <vehicle_id> along the orientation direction, given in [m].
                    "sideways_width": float,                 # width of the vehicle
                    ↪  <vehicle_id> perpendicular to the orientation direction, given in
                    ↪  [m].
                },
            },
            "pedestrians": {                    # dictionary of nearby pedestrians
                <pedestrian_id: int>:  {
                    "location_world_coord": np.ndarray,      # numpy array of shape (2,)
                    ↪  which contains (x, y) of the center location of pedestrian
                    ↪  <pedestrian_id> in world coordinates given in [m]
                    "orientation_unit_vector": np.ndarray,   # numpy array of shape (2,)
                    ↪  which contains (x, y) of unit vector orientation of pedestrian
                    ↪  <pedestrian_id> in world coordinates. The vehicle moves in the
                    ↪  direction of the orientation.
                    "forward_speed": float,                  # speed of pedestrian
                    ↪  <pedestrian_id> relative to the orientation given in [m/s].
                    "forward_length": float,                 # length of the pedestrian
                    ↪  <pedestrian_id> along the orientation direction, given in [m].
                    "sideways_width": float,                 # width of the pedestrian
                    ↪  <pedestrian_id> perpendicular to the orientation direction, given
                    ↪  in [m].
                },
            }
        }

Returns:
    - speed_level: str          # Choose from ("MOVE", "SLOW", "STOP").
    - turn_angle: float         # Predicted turn angle of the ego vehicle to reach the
    ↪  target waypoint in [degrees]. The range should be between -180 to 180 degrees
```
~~~

Listing A.7: Docstring given as part of the setup prompt for the LangProp agent

A.6 Notes on specifying the policy

One of the challenges in the early stages of the project was in specifying the inputs and outputs of the function. Most of the failures in learning a policy were due to misspecification of the inputs, rather than a fundamental problem with the LLM or with LangProp. For instance, we found that it is crucial to specify the units of the input values, e.g. m/s, which allowed the LLM to choose sensible values for some internal parameters. It was also important to name input variables explicitly such that it is clear whether the coordinates are given as absolute world coordinates or coordinates relative to the ego vehicle. A useful property of LangProp is that because the LLM has some understanding of the world from natural language, it can easily incorporate this knowledge when generating the code, constraining the search space of feasible code. We can further guide the LLM to generate policies with certain characteristics, e.g. having a larger safety margin, by expressing our preferences in the prompts. This adds to the benefits of the LangProp approach, where it is easier to encourage policies to exhibit certain behaviors.

A.7 Details of the prompt template engine

In the template engine, every line that begins with “#" is treated as comments. Every line that begins with “$ " or line blocks in between “$begin" and “$end" are treated as executable Python code, as well as everything surrounded by {{ }} in a single line. If a “print" function is used within the prompt template, it will execute the Python code inside the print function and render the resulting string as a part of the prompt. Variables can be passed to the prompt template engine, and are made accessible in the local scope of the prompt template.

As an example, consider the following prompt template.

```
{{" and ".join(p for p in people)}} {{"are" if len(people) > 1 else "is"}} work here.
$begin
for i, p in enumerate(people):
    print(f"{p} is employee No. {i + 1}.")
$end
```

Listing A.8: Example prompt template

If the prompt template engine is called with the arguments `read_template("example", people=["Tom", "Jerry"])`, this resolves to: “Tom and Jerry work here.\nTom is employee No. 1.\nJerry is employee No. 2.".

A.8 How to choose the priority discount factor

How the priorities of the policies are calculated has a large effect on the final performance of the trained LangProp model. For a stationary training distribution (e.g. supervised learning on a fixed offline dataset), whether one uses the immediate average score, a running average, or an exponential average does not make a difference except that just using the immediate average score results in a more stochastic result due to fewer numbers of samples. If the computational resources and time are not constrained, one could increase the batch size and just use the immediate average score. If these are constrained, one may adopt a running average with smaller batch sizes. This works when the training distribution is stationary and there are no other changing components other than the policy currently training.

If the training distribution changes or the policy consists of multiple chained modules, each with a learnable sub-policy, we can no longer use a simple running average but have to use either the scores evaluated on a single large batch or the exponential averaging scheme. The current implementation of LangProp does not support multiple chained modules, but is a foreseeable and natural extension to the framework. Changes in the training distribution are expected in DAgger or reinforcement learning. For training our LangProp agent in Section 4.3.3, we used a discount factor $\gamma = 0$, effectively only using the immediate average scores evaluated on a freshly sampled batch. This is because forward passes through the LangProp driving policies are fast due to not having any complex components so we could afford to have a large batch size. However, in applications where forward passes are expensive and the batch size must be small, using exponential averaging with a non-zero discount factor γ is recommended.

A.9 Use of the term “back-propagation"

The current LangProp implementation is limited to an update of a single module, i.e. it does not yet accommodate for chaining of modules. We have explored this path by making the LLM generate docstrings of helper functions so that submodules can be instantiated, and track priorities also for submodules. However, version tracking of submodules and the mechanism of providing feedback for submodule updates were substantial challenges. LangProp v1 does not implement the full back-propagation algorithm, but we refer to a single-layer feedback operation as *back-prop* to highlight the similarities and encourage future research in this area.

B Autonomous Driving and the CARLA benchmark

Approaches to Autonomous Driving can be broadly classified into modular systems and end-to-end systems (Yurtsever et al., 2020). Most systems take a modular approach (Urmson et al., 2008;

Levinson et al., 2011; Wei et al., 2013; Maddern et al., 2017), which has human-defined rules that orchestrate separately engineered components for localization and mapping, object detection, tracking, behavior prediction, planning, and vehicle control. Such systems allow compartmentalization and better interpretability, but can be complex and require domain knowledge to maintain and update. Another challenge is error propagation (McAllister et al., 2017), i.e. the upstream outputs can be erroneous and must be corrected downstream. Recent work has harnessed end-to-end learning to address these issues. Imitation learning (IL) (Bojarski et al., 2016; Bansal et al., 2018) optimizes the policy to match actions taken by experts, and is the most widely used approach. However, its performance is upper-bounded by the expert. Deep reinforcement learning has also shown successes in simulation (Sallab et al., 2017), on the road (Kendall et al., 2019), and in combination with IL (Lu et al., 2022). Our work combines both the benefit of interpretability of expert systems while also taking a data-driven approach, exposing the system to potential failure modes and adverse scenarios during training time and iteratively optimizing the system towards a well-defined driving metric so that the resulting system is robust to adverse events and potential errors in intermediate components.

CARLA (Dosovitskiy et al., 2017) is a widely used open-sourced 3D simulator for autonomous driving research, and provides a benchmark (CARLA, 2020) to evaluate driving performances of both privileged expert agents and sensor-only agents. Many prior works on CARLA have open-sourced their expert agents. Roach (Zhang et al., 2021) trained a PPO agent (Schulman et al., 2017) on handcrafted reward signals with privileged information. The heavy lifting is done at the reward shaping level, where hazardous agents are identified and the desired speed and pose are computed. Roach expert is also used in MILE (Hu et al., 2022) and TCP (Wu et al., 2022), where TCP has an additional emergency braking upon detecting potential collisions. TransFuser (Chitta et al., 2022), InterFuser (Shao et al., 2023) and TF++ (Jaeger et al., 2023) implement their handcrafted expert systems, either using cuboid intersections or line intersections for hazard detection. TransFuser also introduced the Longest6 benchmark, which consists of longer routes compared to the official CARLA benchmark and is less saturated.

C Data collection

C.1 Data agent

To standardize the data collection and evaluation pipeline for both our expert agent and our LangProp agent, we implement a generic `DataAgent` that collects basic information of the CARLA environment which can be used. These are the 3D bounding box coordinates of the actors in the scene (pedestrians, vehicles, traffic lights, and stop signs), the velocity of the pedestrians and vehicles, distances to the next traffic light and stop sign in the current lane, and the next waypoint to navigate towards. In addition, it also collects the RGB, depth, lidar, segmentation, top-down view, and the expert's control actions which can be used to train image-based driving policies. We created this standardized data collection agent which is decoupled from our expert agent and the LangProp agent, and has the option of turning off sensors that are not used for data collection to save computation time and data storage.

The data collection agent itself does not have a driving policy. It expects a separate `AgentBrain` that takes a dictionary of scene information curated by the data agent as input and outputs a vehicle control action (throttle, brake, and steering). All driving agents inherit from the `DataAgent` class, each with an `AgentBrain` that implements its driving policy. It is also possible to chain multiple agent brains as an array, where the previous agent brain's control decision is provided as an additional input to the next agent brain. This is useful for our DAgger and online agents, which require expert supervision during online rollouts.

C.2 Expert agent

Our expert agent only uses the data collected by the data agent to ensure that the LangProp agent has access to the same privileged information as the expert agent. For every interval of $0.25\ s$ up to $2\ s$ into the future, we evaluate whether the ego vehicle polygon will intersect any of the actor polygons, assuming that the ego vehicle will maintain velocity, and the other actors will move in the current orientation with a speed less than or equal to the current speed. The ego vehicle polygon is padded forward by $2\ m$, and by $2\ m$ either left or right upon lane changes. Apart from lane changing, only actors that are ahead of the ego vehicle are considered, i.e. with a field of view of $180°$. The traffic light and stop sign that affect the vehicle are identified by querying the associated waypoints in the CARLA simulator. For pedestrians, vehicles, traffic light, and stop sign, the distances to the obstacles are calculated. The normal driving speed is 6 m/s ("MOVE"). If any of the distances are reachable within 2 s with a 2 m margin ("SLOW"), the target speed is set to the speed which allows a 2 s margin, and if the distance is below 2 m ("STOP"), the target speed is set to 0 m/s. Steering is evaluated by calculating the angle to the next waypoint, which is 4 m ahead of the current position of the ego vehicle. A PID controller is used for low-level control to convert the target speed and angle to throttle, brake, and steering.

D Training the LangProp agent

D.1 Training strategy

For all the LangProp agents, the training data is collected only on the training routes in CARLA leaderboard (CARLA, 2020), and data collected on the test routes by the expert agent with expert action labels is used as the validation dataset. See Appendix E.1 for more details on the routes. For the LangProp agent trained offline, we only use data collected by the expert agent as training data. For the online training, we only use data collected by the current LangProp model's inference policy, i.e. the policy code with the highest priority at the time of rollout. For DAgger training, we have a split of 1000 training samples collected offline and 1000 samples collected online in every replay batch to evaluate the objective score. Strictly speaking, DAgger (Ross et al., 2011) should incrementally add new online samples to a buffer initialized with offline samples. However, we found that this prevents the LangProp model from learning from infractions during the early stages of the training, since online samples with infractions are the minority of all the samples. For this reason, we maintained an even split between offline and online samples throughout the training, with a sampling weight of 100 for samples with infractions. Sampling is without replacements, so that a particular training sample is only sampled once per replay batch.

D.2 Training objective

The training objective for the LangProp driving agent is given as Equation (2),

$$
\begin{aligned}
S(a^{\pi}, a^{\pi_e}, a^{\pi_b}, I, E) = \mathbb{1}\big[(a^{\pi}_{\text{speed}} = a^{\pi_e}_{\text{speed}}) \wedge [\neg I \vee \{(a^{\pi}_{\text{speed}} \neq a^{\pi_b}_{\text{speed}}) \wedge (a^{\pi_e}_{\text{speed}} \neq a^{\pi_b}_{\text{speed}})\}]\big] \\
+ r_{\text{infrac}} \mathbb{1}(I \wedge (a^{\pi}_{\text{speed}} = a^{\pi_b}_{\text{speed}}) \wedge (a^{\pi_e}_{\text{speed}} \neq a^{\pi_b}_{\text{speed}})) \\
+ r_{\text{angle}} \mathbb{1}(|a^{\pi}_{\text{angle}} - a^{\pi_e}_{\text{angle}}| > \theta_{\text{max}}) + r_{\text{error}} \mathbb{1}(E)
\end{aligned} \tag{2}
$$

where a^{π}, a^{π_e} and a^{π_b} are actions taken by the current policy, expert policy, and behavior policy used to collect the training sample, respectively, I and E are boolean variables for infraction and exception occurrences, $r_{\text{infrac}} = r_{\text{error}} = r_{\text{angle}} = -10$ are penalties for infraction, exception, and exceeding angle error of $\theta_{\text{max}} = 10°$, and $\mathbb{1}$ equates to 1 if the boolean argument is true, and 0 otherwise. The expert is only imitated when there are no infractions, or if the expert was not the behavior policy that incurred the infraction, and an infraction cost is only given when the current policy takes the same action as the behavioral policy that caused the infraction when the expert chose a different action.

D.3 Hyperparameters

Notable training hyperparameters are the number of policies chosen for updates $N^U = 2$, the number of responses per query $N^R = 2$, the number of policies to keep $N^K = 20$, the frequency of batch updates (every 100 new samples in the replay buffer), batch sizes for online replay data (1000) and offline expert data (1000), the sampling weight for infractions (100), and the infraction, exception, and angle penalties ($r_{\text{infrac}} = r_{\text{error}} = r_{\text{angle}} = -10$). For better performance, it is possible to increase N^U, N^R, and N^K, but with a trade-off of computational time and the cost of using OpenAI API. With our experiment setting, around 700 training steps are taken, 1400 queries are made, and 2800 responses are received from GPT 3.5 per training job, which costs roughly \$150.

Table 2: A breakdown of the number of routes per town, the average length of the routes per town, and traffic density for the training routes, testing routes, and the Longest6 benchmark.

Routes	Training routes			Testing routes			Longest6		
	count	avg. dist.	density	count	avg. dist.	density	count	avg. dist.	density
Town 1	10	776.3	120	-	-	120	6	898.8	500
Town 2	-	-	100	6	911.7	100	6	911.7	500
Town 3	20	1392.5	120	-	-	120	6	1797.5	500
Town 4	10	2262.6	200	10	2177.8	200	6	2102.4	500
Town 5	-	-	120	10	1230.1	120	6	1444.7	500
Town 6	10	1915.4	150	-	-	150	6	2116.7	500

E EVALUATION

E.1 CARLA BENCHMARK, ROUTES AND TOWNS

The driving scores are computed by the CARLA leaderboard evaluator (CARLA, 2020), using the official training and test routes, and the Longest6 benchmark provided by Chitta et al. (2022). There are towns $1-6$ across the benchmarks. Towns $7-10$ are also used in the official online leaderboard. A breakdown of routes for each benchmark is shown in Table 2. Towns 2 and 5 are withheld in the training routes and only appear in the testing routes and the Longest6 benchmark. The Longest6 benchmark has longer routes with denser traffic.

The main metric of the leaderboard is the driving score, which is evaluated as $\frac{1}{N}\sum_i^N(R_i I_i)$, where i denotes the index of the N routes used for evaluation, R_i is the percentage of route completion of the i-th route, and I_i is the infraction factor of the i-th route. The infraction factor is a product of infraction coefficients for pedestrian collision (0.5), vehicle collision (0.60), collision with static objects (0.65), running a red light (0.70), and running a stop sign (0.80). The driving score per route is equal to the route completion R_i when there are no infractions, and is discounted for every infraction by a corresponding infraction factor. Note that in the Longest6 benchmark, the authors decided to remove the stop sign penalty by setting its infraction coefficient to 1.0, which we adhere to in our experiments.

E.2 SOFTWARE DETAILS

We use CARLA version 0.9.10 for the experiments to maintain consistency with other baseline experts that assume this version. Our expert has been tested both on CARLA version 0.9.10 and version 0.9.11. For LangProp training, we used GPT 3.5 Turbo 16k chat completion API by OpenAI. We used the 16k-token model since the update prompt often exceeds the 8k-token context size of a smaller model.

F CODE GENERATION EXAMPLES

F.1 SOLUTIONS FOR SUDOKU

F.1.1 INCORRECT SOLUTION GENERATED ZERO-SHOT

```
import numpy as np

def find_empty(sudoku):
    for i in range(len(sudoku)):
        for j in range(len(sudoku[0])):
            if sudoku[i][j] == 0:
                return (i, j)  # row, col
    return None

def is_valid(sudoku, num, pos):
    # Check row
    for i in range(len(sudoku[0])):
        if sudoku[pos[0]][i] == num and pos[1] != i:
            return False

    # Check column
    for i in range(len(sudoku)):
        if sudoku[i][pos[1]] == num and pos[0] != i:
            return False

    # Check 3x3 box
    box_x = pos[1] // 3
    box_y = pos[0] // 3

    for i in range(box_y*3, box_y*3 + 3):
        for j in range(box_x * 3, box_x*3 + 3):
            if sudoku[i][j] == num and (i,j) != pos:
                return False

    return True


def solve_sudoku(sudoku: np.ndarray, width: int, height: int) -> np.ndarray:
    find = find_empty(sudoku)
    if not find:
        return sudoku
    else:
        row, col = find

    for i in range(1, width*height + 1):
        if is_valid(sudoku, i, (row, col)):
            sudoku[row][col] = i

            if solve_sudoku(sudoku, width, height) is not None:
                return sudoku

            sudoku[row][col] = 0

    return None
```

Listing A.9: Example code to solve Sudoku generated zero-shot before LangProp optimization. The code is instructed to solve a general Sudoku with subgrids of size $H \times W$, but confuses it with the standard 3×3 Sudoku.

F.1.2 Correct solution after applying LangProp

```
import numpy as np

def find_empty(sudoku):
    for i in range(len(sudoku)):
        for j in range(len(sudoku[0])):
            if sudoku[i][j] == 0:
                return (i, j)  # row, col
    return None

def is_valid(sudoku, num, pos):
    # Check row
    for i in range(len(sudoku[0])):
        if sudoku[pos[0]][i] == num and pos[1] != i:
            return False

    # Check column
    for i in range(len(sudoku)):
        if sudoku[i][pos[1]] == num and pos[0] != i:
            return False

    # Check 3x3 box
    box_x = pos[1] // 3
    box_y = pos[0] // 3

    for i in range(box_y*3, box_y*3 + 3):
        for j in range(box_x * 3, box_x*3 + 3):
            if sudoku[i][j] == num and (i,j) != pos:
                return False

    return True


def solve_sudoku(sudoku: np.ndarray, width: int, height: int) -> np.ndarray:
    find = find_empty(sudoku)
    if not find:
        return sudoku
    else:
```

```python
        row, col = find

    for i in range(1, width*height + 1):
        if is_valid(sudoku, i, (row, col)):
            sudoku[row][col] = i

            if solve_sudoku(sudoku, width, height) is not None:
                return sudoku

            sudoku[row][col] = 0

    return None
```

Listing A.10: Example code to solve Sudoku after LangProp optimization. The code outputs a correct solution.

F.2 Solutions for CartPole

F.2.1 Incorrect solution generated zero-shot

```python
import numpy as np

def solve_cartpole(cart_position, cart_velocity, pole_angle, pole_angular_velocity) -> int:

    # Set the threshold for the pole angle
    threshold_angle = 0.1

    # Check the angle of the pole
    if np.abs(pole_angle) < threshold_angle:
        # If the angle is small enough, take no action
        action = 0
    else:
        # If the angle is too large, push the cart in the direction of the pole
        if pole_angle > 0:
            # Push the cart to the right
            action = 1
        else:
            # Push the cart to the left
            action = 0

    return action
```

Listing A.11: Example policy to solve CartPole generated zero-shot before LangProp optimization. The overly simplistic policy achieves a mean score of 9.9 out of 500.

F.2.2 Correct solution after applying LangProp

```python
import numpy as np

def solve_cartpole(cart_position: float, cart_velocity: float, pole_angle: float,
↪   pole_angular_velocity: float) -> int:

    # Define the constants for the PD controller
    kp_pole = 0.5
    kd_pole = 0.1

    kp_cart = 0.2
    kd_cart = 0.05

    # Compute error signals and their derivatives
    pole_error = pole_angle
    pole_error_derivative = pole_angular_velocity

    cart_error = cart_position
    cart_error_derivative = cart_velocity

    # Compute the control signals for the pole and cart using PD controllers
    pole_control_signal = kp_pole * pole_error + kd_pole * pole_error_derivative
    cart_control_signal = kp_cart * cart_error + kd_cart * cart_error_derivative

    # Combine the control signals to get the final action
    if pole_control_signal < -cart_control_signal:
        action = 0
    elif pole_control_signal > cart_control_signal:
        action = 1
    else:
        action = 1

    return action
```

Listing A.12: Example policy to solve CartPole after LangProp optimization. The policy learns a PID controller and achieves a mean score of 500 out of 500.

F.3 Driving code generated by LangProp

We show an example driving policy generated using LangProp, trained with both imitation learning and reinforcement learning, as described in Section 4.3.3. Please refer to our open-sourced code repository for the full prompts and code used to train the policy, and pre-trained checkpoints for each training setting used in our evaluation.

```
import numpy as np
from typing import List, Tuple, Union, Dict

def predict_speed_and_steering(scene_info: Dict, stopping_time_margin=2.0,
                               reaction_time=1.0, stopping_speed=0.1, slow_speed=2.5,
                               ↪  safety_margin=2.0) -> Tuple[str, float]:
    """
    Given the state of the scene, find the required speed level and turn angle of the ego
    ↪  vehicle.

    Args:
        scene_info (dict): Contains information about the scene.
        stopping_time_margin (float): Time margin to determine when to stop in seconds
        ↪  [default: 2.0]
        reaction_time (float): Time taken to react to external events in seconds [default:
        ↪  1.0]
        stopping_speed (float): Minimum speed below which the vehicle should stop in m/s
        ↪  [default: 0.1]
        slow_speed (float): Speed at which the vehicle should slow down in m/s [default:
        ↪  2.5]
        safety_margin (float): Safety margin in meters between the vehicle and other
        ↪  objects [default: 2.0]

    Returns:
        (str, float): A tuple containing the required speed level and turn angle of the ego
        ↪  vehicle.
                      The speed level can be "MOVE", "SLOW", or "STOP". The turn angle
                      ↪  should be a float between -180 to 180 degrees.
    """
    # Step 1: Given the input dictionary, extract all the relevant parameters.
    ego_location = np.array(scene_info['ego_location_world_coord'])
    ego_target_location = np.array(scene_info['ego_target_location_world_coord'])
    ego_orientation = np.array(scene_info['ego_orientation_unit_vector'])
    ego_forward_speed = scene_info['ego_forward_speed']
    ego_length = scene_info['ego_length'] + safety_margin
    ego_width = scene_info['ego_width'] + safety_margin
    vehicles = scene_info['vehicles']
    pedestrians = scene_info['pedestrians']
    distance_to_stop_sign = scene_info['distance_to_stop_sign']
    distance_to_red_light = scene_info['distance_to_red_light']

    # Step 2: Convert all coordinates for vehicles and pedestrians to relative coordinates
    ↪  in terms of the ego vehicle orientation.
    vehicles_rel = {}
    for k, v in vehicles.items():
        location = np.array(v['location_world_coord']) - ego_location
        orientation = np.array(v['orientation_unit_vector'])
        speed = v['forward_speed']
        length = v['forward_length'] + v['sideways_width'] + safety_margin
        width = v['sideways_width'] + safety_margin
        location_rel = np.dot(location, ego_orientation),
        ↪  np.abs(np.dot([-ego_orientation[1], ego_orientation[0]], location))
        if location_rel[0] > 0:
            vehicles_rel[k] = {'location_rel': location_rel, 'speed': speed, 'length':
            ↪  length, 'width': width}

    pedestrians_rel = {}
    for k, v in pedestrians.items():
        location = np.array(v['location_world_coord']) - ego_location
        orientation = np.array(v['orientation_unit_vector'])
        speed = v['forward_speed']
        length = v['forward_length'] + safety_margin
        width = v['sideways_width'] + safety_margin
        location_rel = np.dot(location, ego_orientation),
        ↪  np.abs(np.dot([-ego_orientation[1], ego_orientation[0]], location))
        if location_rel[0] > 0:
            pedestrians_rel[k] = {'location_rel': location_rel, 'speed': speed, 'length':
            ↪  length, 'width': width}

    # Step 3: Compute the Euclidean distance from the ego location to the target location.
    distance_to_target = np.linalg.norm(ego_target_location - ego_location)

    # Step 4: Calculate the threshold stopping distance and threshold slow distance based
    ↪  on the current speed.
    if ego_forward_speed < stopping_speed:
        stopping_distance = safety_margin
        slow_distance = ego_length / 2
    else:
        stopping_distance = ((ego_forward_speed - stopping_speed) ** 2) / (2 * 0.7)  +
        ↪  safety_margin + ego_length / 2
        slow_distance = ((ego_forward_speed - slow_speed) ** 2) / (2 * 0.7) + safety_margin
        ↪  + ego_length / 2

    # Step 5: Check if there is a stop sign and the distance is smaller than the stopping
    ↪  distance. If yes, initiate a stop action if the speed is greater than the stopping
    ↪  speed.
    if distance_to_stop_sign is not None and distance_to_stop_sign < stopping_distance:
        if ego_forward_speed <= stopping_speed:
            speed_level = "MOVE"
        else:
            stopping_speed_current = max(distance_to_stop_sign / 2, stopping_speed)
            if ego_forward_speed > stopping_speed_current:
                ego_forward_speed = stopping_speed_current
                speed_level = "STOP"
            else:
                speed_level = "MOVE"

    # Step 6: Check if there is a red light and the distance is smaller than the stopping
    ↪  distance. If yes, initiate a stop action.
    elif distance_to_red_light is not None and distance_to_red_light < stopping_distance:
        speed_level = "STOP"

    # Step 7: Check for vehicles and pedestrians that may cause collision course, and
    ↪  decide whether to STOP, SLOW or MOVE the ego vehicle.
    else:
        collision_vehicle = False
        collision_pedestrian = False
        min_longitudinal_distance = stopping_distance
        min_lateral_distance = float('inf')
        for k, v in vehicles_rel.items():
            location_rel = v['location_rel']
            speed = v['speed']
            length = v['length']
            width = v['width']
            longitudinal_distance = location_rel[0] - v['length'] / 2 - ego_length / 2

            # check if there is a collision course with the ego vehicle
            if np.abs(location_rel[1]) <= width / 2 + ego_width / 2 and
            ↪  longitudinal_distance <= stopping_distance:
                collision_vehicle = True
                if longitudinal_distance <= 0:
                    speed_level = "STOP"
                    break
            # check if the vehicle is within safety margin
            if longitudinal_distance < stopping_distance and np.abs(location_rel[1]) <=
            ↪  width / 2 + ego_width / 2:
                if np.abs(speed - ego_forward_speed) < 0.5 and speed <= ego_forward_speed:
                    continue
                min_longitudinal_distance = min(longitudinal_distance - v['length'] / 2 -
                ↪  ego_length / 2, min_longitudinal_distance)
                min_lateral_distance = np.minimum(width / 2 + ego_width / 2 -
                ↪  np.abs(location_rel[1]), min_lateral_distance)

        for k, v in pedestrians_rel.items():
            location_rel = v['location_rel']
            speed = v['speed']
            length = v['length']
            width = v['width']
            longitudinal_distance = location_rel[0] - length / 2 - ego_length / 2

            # check if there is a collision course with the ego vehicle
            if np.abs(location_rel[1]) <= width / 2 + ego_width / 2 and
            ↪  longitudinal_distance <= stopping_distance:
                collision_pedestrian = True
                if longitudinal_distance <= 0:
                    speed_level = "STOP"
                    break
            # check if the pedestrian is within safety margin
            if longitudinal_distance < stopping_distance and np.abs(location_rel[1]) <=
            ↪  width / 2 + ego_width / 2:
                if np.abs(speed - ego_forward_speed) < 0.5 and speed <= ego_forward_speed:
                    continue
                min_longitudinal_distance = min(longitudinal_distance - length / 2 -
                ↪  ego_length / 2, min_longitudinal_distance)
                min_lateral_distance = np.minimum(width / 2 + ego_width / 2 -
                ↪  np.abs(location_rel[1]), min_lateral_distance)

        # Step 8: Initiate a stop action if the ego vehicle is about to collide with a
        ↪  nearby vehicle or pedestrian.
        if collision_vehicle or collision_pedestrian or min_longitudinal_distance <=
        ↪  safety_margin/2 or min_lateral_distance <= safety_margin/2:
            speed_level = "STOP"
            ego_forward_speed = 0
        # Step 9: Initiate a slow action if the vehicles or pedestrian within the safe
        ↪  stopping distance margin.
        elif min_longitudinal_distance <= slow_distance and min_longitudinal_distance >=
        ↪  stopping_distance and min_lateral_distance <= ego_width:
            speed_level = "SLOW"
            if np.abs(min_lateral_distance) > 0 and np.abs(min_lateral_distance -
            ↪  ego_width) > 0:
                speed_factor = (min_longitudinal_distance - stopping_distance) /
                ↪  (slow_distance - stopping_distance/2)
                speed_factor = min(max(0.0, speed_factor), 1.0)
                ego_forward_speed = slow_speed * speed_factor + ego_forward_speed * (1 -
                ↪  speed_factor)
        # Step 10: Initiate a move action if no obstacles are present
        else:
            speed_level = "MOVE"
            ego_forward_speed = min(ego_forward_speed + 0.2, 6.0)

    # Step 11: Compute the angle between the ego vehicle orientation and the vector
    ↪  pointing to the target in world coordinates.
    target_direction = ego_target_location - ego_location
    target_direction_ego = np.dot(target_direction, ego_orientation),
    ↪  np.dot([-ego_orientation[1], ego_orientation[0]], target_direction)

    # Step 12: Rotate the vector to the coordinate system of the ego vehicle and return the
    ↪  angle.
    target_angle = np.arctan2(target_direction_ego[1], target_direction_ego[0]) * 180.0 /
    ↪  np.pi if np.linalg.norm(target_direction_ego) > 0 else 0.0
    target_angle = ((target_angle + 180) % 360) - 180

    return speed_level, target_angle
```

Listing A.13: Example driving policy generated by LangProp, trained with both imitation learning and reinforcement learning.

G Future work

LangProp is a framework that harnesses the capability of LLMs to apply data-driven optimization techniques to code optimization. We do not claim that a solution using LangProp is appropriate for all problems - in fact, neural networks excel in working with continuous state-action spaces and low-level control, whereas LLMs have advantages in handling high-level planning and reasoning tasks, rather than low-level control tasks. Our intention is to propose an alternative learning paradigm that allows LLMs to be used to learn high-level planning which has hitherto been a difficult problem for other machine learning approaches (e.g. neural networks).

There are numerous future research directions that could improve the capability of LangProp as a training framework, as well as give a better theoretical foundation, such as (a) chaining of modules with a full back-propagation algorithm, (b) improvements to the evolutionary algorithm (e.g. priority mechanism), (c) a robust sampling mechanism for failed examples upon updates, (d) incorporating human feedback in natural language during policy updates, and (e) using LangProp with LLMs fine-tuned for code correction and optimization tasks. In particular, scaling our approach to larger repositories and complex systems would require a multi-modular approach that can propagate useful learning signals to subcomponents if there are multiple failure points in the system.

Applying LangProp to reinforcement learning tasks has open questions in credit assignment and value estimation. We have demonstrated that reinforcement learning policies written as code can be improved using LangProp if either (a) the policy can be optimized on episodic returns with a Monte-Carlo method (e.g. CartPole), or (b) there is immediate feedback from the environment (e.g. infractions in CARLA). However, for complex tasks that have delayed rewards, it is necessary to have an accurate value/advantage estimator for credit assignment. Since replacing a neural value estimator with a code-based function is not feasible, it is most likely that a hybrid method (having an interpretable code-based actor policy trained with LangProp that uses a value function estimated by a neural network as a critic) would be a way to apply LangProp to complex reinforcement learning scenarios. However, this is also an open-ended question, which calls for further exploration.

Having an LLM in the RL optimization means that we could potentially harvest more useful signals from the environment, rather than relying just on sparse scalar rewards for updates. For instance, having descriptive feedback from the Gymnasium environment on the failure modes of the agent, given either as a warning or natural language feedback, can significantly accelerate the learning of the RL agent. This also allows a more seamless integration of human-in-the-loop feedback.

Finally, more investigation is required in terms of the robustness and safety of LLM-written applications. This is applicable to all systems that involve code generation. While our framework iteratively improves the quality of the code and filters out potential errors that make the final code policy less likely to contain errors, additional safety mechanisms and firewalls are necessary during the training process, since the code is evaluated based on execution, which could potentially be a source of attacks or risk. We stress the importance of additional safety precautions before deployment.

We believe that LangProp opens up new possibilities for data-driven code development. While zero-shot applications of LLMs have enabled tools such as GitHub Copilot, some suggestions are inaccurate or misaligned with the user's intentions, whereas if we have data or unit tests that the code needs to satisfy, the code suggestions can be made much more accurate by first running evaluations on these test suites and choosing the best possible suggestion that satisfies the requirements. Planning is one aspect of autonomous driving that has not yet successfully adopted a data-driven approach, for good reasons, since neural networks often struggle to produce generalizable high-level planning rules and are less interpretable. Therefore, most methods currently in deployment have human-engineered planning algorithms. Our LangProp framework is insufficient to replace such systems since it lacks the robustness that human-designed systems have to offer, and more research needs to be done in this direction. We hope that our work will provide inspiration for future research to make the framework more robust and safely deployable in the real world.