13 Synergies with Generative AI

13.1 Background On Large Language Models

Large Language Models (LLMs), such as OpenAI's GPT series, represent a significant advancement in the field of artificial intelligence. These models are characterized by their vast scale and capacity to process and generate human-like text, making them powerful tools for a variety of language-based tasks. The backbone of these models is the transformer architecture, introduced by Vaswani et al. 2017, which employs a self-attention mechanism allowing the model to consider the importance of all other words in a sentence, regardless of their positional distance from the word being processed.

This self-attention mechanism simplifies the handling of long-range dependencies in text, which is critical for tasks such as summarizing a long document or understanding the nuances in a conversation. Unlike models that rely on recurrent layers, the transformer's architecture allows for parallel processing of data, increasing efficiency and scalability when managing the large datasets essential for training LLMs. Self-attention was described in Section 4.4.

LLMs undergo extensive pre-training on large text corpora, learning to predict the next token in a sequence. This foundational training, as seen in models like BERT (Devlin et al. 2018), is not just about massive data ingestion. Researchers also fine-tune various aspects such as the ratio of different data types in the training set, the learning rate, and other training parameters to optimize performance. This meticulous tuning process enhances the model's ability to understand context and generate coherent responses.

The performance of LLMs also adheres to scaling laws, as explored in Kaplan et al. 2020. These laws demonstrate that model performance improves logarithmically with increases in size, data volume, and computational power, emphasizing the crucial role of data scale. Large-scale data not only aids in training more accurate models but also ensures a broader linguistic coverage, allowing the models to generalize better across various tasks. This extensive data requirement underpins the significance of scaling laws in predicting the effectiveness of LLMs as they grow increasingly large.

Regarding applications, LLMs extend far beyond powering chatbots and enter the realm of broader uses, such as serving as autonomous agents that oversee software interactions or operate robotic interfaces. These ventures into agent-centric and tool-use applications underscore the versatility and transformative potential of LLMs across diverse technological and human interaction domains.

However, despite their extensive pre-training, LLMs in their raw form are not fully equipped to handle specialized tasks directly. The transition from a general linguistic understanding to specific real-world applications requires significant post-training optimization. This critical phase involves fine-tuning the model on task-specific datasets, which refines its responses according to particular needs. Additionally, the use of prompt-engineering enhances how models interpret and respond to queries, making them more effective and adaptable. These adaptations are essential for tailoring LLMs to specialized functions, ranging from conversational AI to more intricate and domain-specific applications.

While the current trend predominantly focuses on constructing larger models trained on increasingly vast datasets—a strategy consistently rewarded by the scaling law—there exists a parallel strand of research that employs evolutionary computing to enhance LLMs in innovative and less conventional manners (Wu et al. 2024; Chao et al. 2024), as we will explore in subsequent sections.

13.2 Evolutionary Computing Helps Improve LLMs

13.2.1 Evolutionary Prompt Engineering/Adaptation

To adapt LLMs for specific downstream tasks, adding an instruction to the input text, known as a discrete prompt, directs the LLMs to perform desired tasks with minimal computational cost. This method eliminates the need for direct manipulation of parameters and gradients, making it especially suitable for LLMs with black-box APIs like GPT-4 (OpenAI 2023) and Gemini (Anil, Borgeaud, et al. 2023). However, the efficacy of LLMs in executing specific tasks heavily relies on the design of these prompts, a challenge commonly addressed through prompt engineering.

Prompt engineering often requires extensive human effort and expertise, with approaches ranging from enumerating and selecting diverse prompts to modifying existing ones to enhance performance. These methods can lead to a cycle of exploration, which might consume resources without substantive gains, or exploitation, which may confine the search to local optima and stifle broader improvements. Evolutionary algorithms, which are particularly suited for discrete prompt optimization, offer a robust alternative. These algorithms view sequences of phrases in prompts as gene sequences, allowing for natural evolutionary processes to be mirrored in prompt adaptation.

Taking this concept further, the evolutionary process can be used to maintain a diversity of prompts, helping to avoid diminishing returns seen in conventional prompt engineering methods. This is achieved by using the LLM itself to modify prompts as well as the strategy for prompt modification, leading to self-referential self-improvement. This approach not only harnesses the LLM's linguistic capabilities but also its ability to iteratively refine the prompts based on performance feedback. By doing so, LLMs can adapt prompts more effectively to the domain at hand, continually improving both the prompts and the methodology by which they are evolved.

This integrated approach leverages evolutionary algorithms and the self-referential capabilities of LLMs to create a dynamic prompt engineering process that not only enhances the performance of LLMs on specific tasks but also propels the field towards more autonomous

and efficient use of foundational models for a wide range of applications. As representative works in this area, we introduce two approaches *EvoPrompt* (Guo et al. 2023) and *Promptbreeder* (Fernando et al. 2023) in this section.

EvoPrompt optimizes prompts for language models by employing evolutionary algorithms, specifically the Genetic Algorithm (GA) and the Differential Evolution (DE). These algorithms are especially suited for situations where direct access to the model's gradients and parameters is not possible, such as when interacting with LLMs through black-box APIs. Figure 13.1 give the GA process in EvoPrompt. The DE process is similar and is not elaborated here.

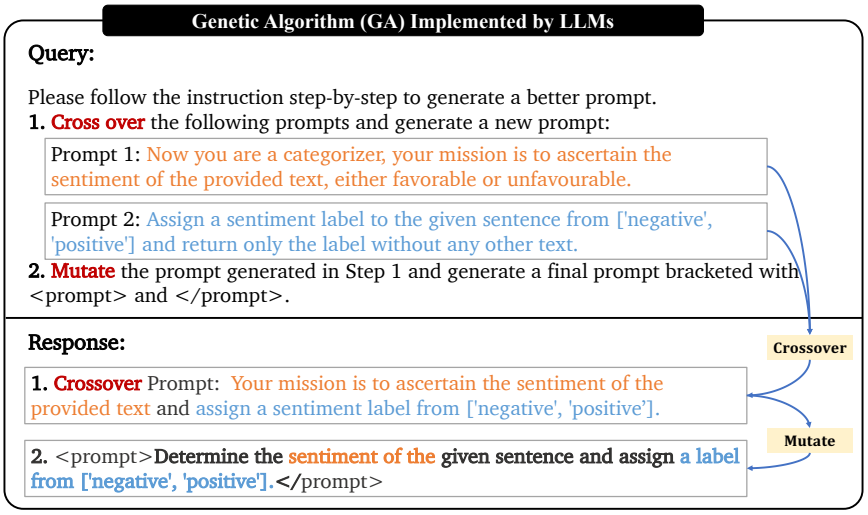


Figure 13.1: **GA process in EvoPrompt.** In Step 1, LLMs perform *crossover* on the given two prompts (words in orange and blue are inherited from Prompt 1 and Prompt 2, respectively). In Step 2, LLMs perform *mutation* on the prompt. (Figure from Guo et al. 2023)

The evolutionary process begins with a set of initial prompts that leverage the wisdom of humans and a development dataset, where each prompt is evaluated based on how effectively it elicits the desired responses from the language model. Throughout a series of iterations, prompts are selected based on their performance scores. New prompts are then generated through evolutionary operations that include combining elements from multiple selected prompts (crossover) and introducing random variations (mutation). The prompts to introduce these operations are illustrated in Figure 13.1. These newly created prompts are subsequently evaluated, and those with superior performance are retained for further refinement in subsequent iterations. This cycle of selection, generation, and evaluation repeats, progressively enhancing the quality of the prompts. A key innovation of this method is the use of the LLM itself to generate new candidate prompts based on evolutionary instructions. This melds the model's advanced natural language processing capabilities with the strategic optimization power of evolutionary algorithms.

In a comprehensive evaluation of the EvoPrompt method, various experiments were conducted across multiple tasks, including language understanding, language generation, and the particularly challenging Big Bench Hard (BBH) tasks. While the EvoPrompt method demonstrated impressive results across all tasks, the performance on BBH is notably representative of its capabilities and generalizes well as BBH is a widely accepted benchmark. For the BBH tasks, the EvoPrompt method was applied to optimize prompts specifically for the GPT-3.5 model. A subset of the test set was used as the development set to iteratively refine the prompts, with the final performance reported as normalized scores, see Figure 13.2. The results were striking: EvoPrompt achieved substantial improvements across all 22 evaluated tasks. Specifically, the DE variant of EvoPrompt led to as much as a 25% improvement in some tasks, with an average improvement of 3.5%. In comparison, the GA variant also performed well but slightly lower, reaching a peak improvement of 15% and an average of 2.5%.

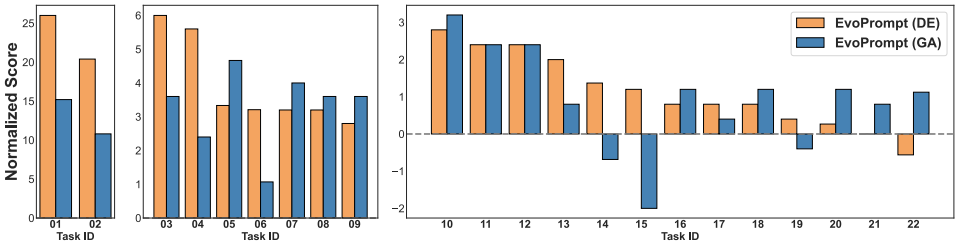


Figure 13.2: **Normalized scores on Big Bench Hard (BBH) tasks for EvoPrompt.** Since the tasks are challenging, GPT-3.5 was used as the LLM. Score normalization is calcuated in comparison to the prompt "Let's think step by step" with a 3-shot Chain-of-Thought demonstration. (Figure from Guo et al. 2023)

Promptbreeder introduces a sophisticaed system for evolving prompts that enhance the performance of LLMs in specific domains, see Figure 13.3 for an overview of the method. Like EvoPrompt, it automates the exploration of prompts by utilizing evolutionary algorithms to generate and refine task prompts that condition LLMs for better responses. In promptbreeder, each task prompt serves to condition the context of an LLM before additional input, aiming to elicit superior responses. The system evaluates the effectiveness of each prompt by testing it on a batch of domain specific Q&A pairs. This evalution informs the evolutionary process, where prompts are iteratively refined. A unique and novel feature of Promptbreeder is its self-referential mechanism, where it applies evolutionary algorithms not just to task prompts but also to the mutation prompts. These mutation prompts guide the generation of new task prompts and are themselves subject to evolution. This "meta-learning" style evolution process ensures continuous improvement in the quality and the relevance of the prompts.

Concretely speaking, Promptbreeder starts with an initial set of task prompts and mutation prompts, derived from combining domain specific problem descriptions with varied "thinking styles" and mutation strategies. This initial population is crucial as it sets the baseline for the evolutionary process, incorporating a rich diversity of approaches and

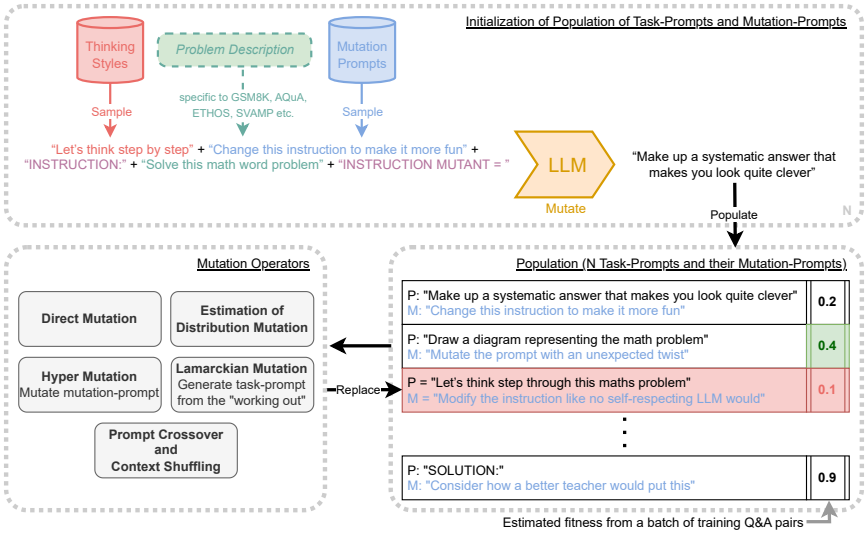


Figure 13.3: **Overview of Promptbreeder.** This process begins with a set of problem description and initial prompts, creating evolution units with task and mutation-prompts. Using a binary tournament genetic algorithm, it evaluates and iteratively refines these prompts across generations, enhancing their effectiveness and domain-specific adaptation. (Figure from Fernando et al. 2023)

perspectives right from the beginning. As depicted in Figure 13.4, Promptbreeder utilizes LLMs as a fundamental component of its mutation operators. These operators are tasked with generating new versions of task prompts by applying transformations dictated by mutation prompts. Each mutation prompt influences how a task prompt is altered, ensuring that the evolution is guided by strategies that are likely to improve performance based on previous iterations.

The mutation process in Promptbreeder includes direct mutations where new task prompts are generated from existing ones by applying simple changes, and more complex mutations where multiple prompts are combined or significantly altered to explore new prompt spaces. This process is depicted through various mutation mechanisms in the Figure 13.4. One of the standout features of Promptbreeder is its self-referential mechanism, where the system not only evolves task-prompts but also the mutation-prompts that guide their evolution. This recursive improvement process ensures that the system becomes increasingly effective over time. The mutation-prompts themselves are subject to evolution, optimized to produce more effective task-prompts as the system learns from its successes and failures.

Promptbreeder has been tested across a variety of domains to evaluate its effectiveness in optimizing prompts for LLMs. These domains include arithmetic reasoning, commonsense reasoning, instruction induction, and hate speech classification. The results indicate that Promptbreeder consistently outperforms the previously considered state-of-the-art Plan-and-Solve (PS+) technique. In tests using the underlying LLM PaLM 2-L, PB showed superior performance on almost all datasets. Notably, its zero-shot accuracy surpasses that

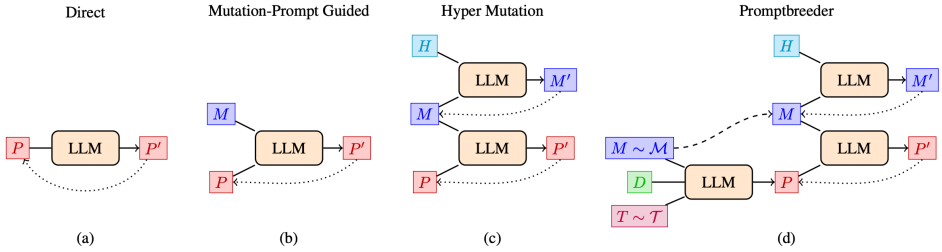


Figure 13.4: **Overview of multiple variants of self-referential prompt evolution.** In (**a**), the LLM is directly used to generate variations P' of a prompt strategy P. Using a mutation prompt M, we can explicitly prompt an LLM to produce variations (**b**). By using a hyper mutation prompt H, we can also evolve the mutation prompt itself, turning the system into a self-referential one (**c**). Promptbreeder (**d**) improves the diversity of evolved prompts and mutation prompts by generating an initial population of prompt strategies from a set of seed thinking-styles $\mathscr{T}$, mutation-prompts $\mathscr{M}$, as well as a high level description D of the problem domain. (Figure from Fernando et al. 2023)

of PS+ in all tests. When few-shot examples are incorporated with the prompts, Promptbreeder shows even more significant improvement, highlighting its robustness in both zero-shot and few-shot scenarios. A specific example of Promptbreeder's capability is demonstrated in its application to the ETHOS Hate Speech Classification problem. Promptbreeder evolved a strategy involving two sequentially applied, relatively long prompts that significantly outperformed the manually designed prompt, see the following box for reference. This adaptation resulted in an accuracy improvement from 80% to 89%, illustrating Promptbreeder's potential for intricate domain-specific task adaptation.

Prompt 1: "Text contains hate speech if it includes any of the following: * Words or phrases that are derogatory, disrespectful, or dehumanizing toward a particular group of people. * Statements that generalize about a particular group of people in a negative way. * Statements that incite hatred or violence against a particular group of people. * Statements that express a desire for the extermination or removal of a particular group of people. 7 : Text contains hate speech if it contains language that is hostile or discriminatory towards a particular group of people. This could include language that is racist, sexist, homophobic, or other forms of hate speech. Hate speech is harmful because it can create a hostile environment for marginalized groups and can lead to discrimination and violence."

Prompt 2: "You are given a piece of text from the internet. You are supposed to label the text as Hate Speech or Not based on the given criteria. Hate Speech is defined as speech that is intended to incite hatred or violence against a particular group of people based on their race, religion, sex, sexual orientation, or national origin. The given criteria are: 1. Identify the target group of the speech. This can be a specific group of people, such as a race, religion, or gender, or it can be a more general group, such as people with disabilities or sexual minorities. 2. Identify the harmful speech. This can be speech that is threatening, abusive, or derogatory. 3. Evaluate the context of the speech. This can include the speaker's intent, the audience, and the time and place of the speech. The advice was: Remember to always evaluate the context of the speech when making a determination as to whether it is hate speech or not. Speech that is intended to be humorous or satirical may not be considered hate speech, even if it contains harmful language."

While both Promptbreeder and EvoPrompt utilize evolutionary algorithms to optimize prompts, there are distinct differences in their methodologies and focus. EvoPrompt primarily concentrates on refining prompts through direct evolutionary operations, such as crossover and mutation, driven by performance evaluations. It uses a more traditional approach where the evolutionary process is straightforward and focused primarily on task prompts alone. In contrast, Promptbreeder introduces a more complex and layered approach by not only evolving the task prompts but also the mutation prompts that guide the task prompt evolution. This self-referential approach allows Promptbreeder to adapt more dynamically to the nuances of different domains by continually refining the mechanisms of prompt evolution itself. Despite these differences, both examples demonstrate the potential of evolutionary computing to significantly enhance the performance of LLMs in seemingly straightforward ways. In the following section, we will explore how evolutionary algorithms can be applied to merge multiple LLMs, resulting in a composite model that embodies a superset of the capabilities of its constituent models.

13.2.2 Evolutionary Model Merging

The intelligence of the human species is not based on a single intelligent being, but based on a collective intelligence. Individually, we are actually not that intelligent or capable. Our society and economic system is based on having a vast range of institutions made up of diverse individuals with different specializations and expertise. This vast collective intelligence shapes who we are as individuals, and each of us follows our own path in life to become the unique individual, and in turn, contribute back to being part of our ever-expanding collective intelligence as a species. Some researchers believe that the development of artificial intelligence will follow a similar, collective path. The future of AI will not consist of a single, gigantic, all-knowing AI system that requires enormous energy to train, run, and maintain, but rather a vast collection of small AI systems–each with their own niche and specialty, interacting with each other, with newer AI systems developed to fill a particular niche.

A noticing and promising trend in the open-source AI ecosystem is that, open-source foundation models are readily extended and fine-tuned in hundreds of different directions to produce new models that are excellent in their own niches. Unsurprisingly, most of the top performing models on Open LLM leaderboards are no longer the original open base models such as LLaMA or Mistral, but models that are fine-tunes or merges of existing models. Furthermore, open models of different modalities are being combined and tuned to be Vision-Language Models (VLMs) which rival end-to-end VLM models while requiring a fraction of the compute to train. Model merging shows great promise and democratizes up model-building to a large number of participants. However, it can be a "black art", relying heavily on intuition and domain knowledge. Human intuition, however, has its limits. With the growing diversity of open models and tasks, we need a more systematic approach. Evolutionary algorithms, inspired by natural selection, can unlock more effective merging solutions. These algorithms can explore a vast space of possibilities, discovering novel and unintuitive combinations that traditional methods and human intuition might miss. In their paper (Akiba et al. 2024), researchers introduced Evolutionary Model Merge, a general evolutionary method to discover the best ways to combine different models. Their method combines two different approaches: (1) Merging models in the Data Flow Space (Layers), and (2) Merging models in the Parameter Space (Weights). See Figure 13.5 for illustration.

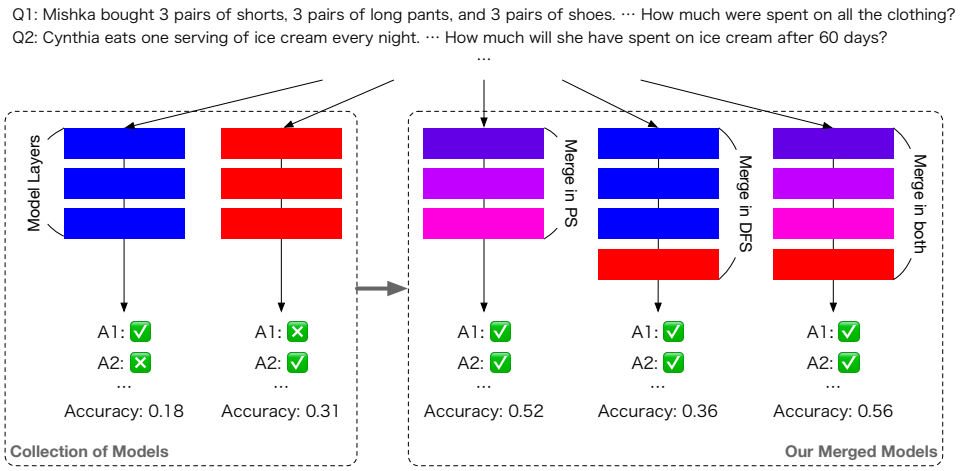


Figure 13.5: **Overview of evolutionary model merging.** (Figure from Akiba et al. 2024)

Info Box: The Intersection of EC and LLMs

At the beginning of generative AI innovation, I (Yujin Tang) began my journey at Google Brain, and later merged into Google DeepMind, primarily focusing on evolutionary algorithms and their applications. The release of GPT-3 inspired me to explore the symbiotic potential between evolutionary computing (EC) and LLMs. With access to a suite of Google internal LLMs and early tests of Gemini, a bunch of us recognized LLMs as exceptional pattern recognition machines. This led to our works (Lange, Tian, and Tang 2024a, 2024b) that explored the possibility of enhancing EC with pre-trained and fine-tuned LLMs.

At the same time, despite the prowess of LLMs in understanding of generating complex patterns, I noted the significant challenges associated with fine-tuning these models for specific tasks. This process demanded extensive engineering, predominantly leaning on gradient-based methods, also a path heavily tread by giants like Google, Meta and OpenAI.

Later when I joined Sakana AI, I attempted to apply the NEAT algorithm to LLMs, treating each layer as an independent node. This approach initially seemed promising but was quickly met with challenges due to the vast search space and the high sensitivity of LLM to local failures-even a small percentage of suboptimal nodes could dramatically affect overall model performance. To combat these issues, I had to implement some strategic constraints such as limiting connections to serial formations and applying scaling matrices, thereby refining the data flow space model merging method. These are all early works in marrying EC and LLMs, but are already demonstrating the transformative power of integrate the two for more adaptive and robust AI systems.

At a high-level, merging in the data flow space uses evolution to discover the best combinations of the layers of different models to form a new model. In the model merge

Table 13.1
Performance Comparison of the LLMs. Models 1–3 are source models, Models 4–6 are merged models, and Models 7–11 are provided for reference. PS stands for Parameter Space merging, and DFS is the abbreviation for Data Flow Spacing merging. (Table from Akiba et al. 2024)

Id.	Model	Type	Size	**MGSM-JA** (acc ↑)
1	Shisa Gamma 7B v1	JA general	7B	9.6
2	WizardMath 7B v1.1	EN math	7B	18.4
3	Abel 7B 002	EN math	7B	30.0
4	**Akiba et al. (2024) (PS)**	1 + 2 + 3	7B	**52.0**
5	**Akiba et al. (2024) (DFS)**	3 + 1	10B	**36.4**
6	**Akiba et al. (2024) (PS+DFS)**	4 + 1	10B	**55.2**
7	Llama 2 70B	EN general	70B	18.0
8	Japanese StableLM 70B	JA general	70B	17.2
9	Swallow 70B	JA general	70B	13.6
10	GPT-3.5	commercial	-	50.4
11	GPT-4	commercial	-	78.8

community, intuition and heuristics are used to determine how and which layers of one model are combined with layers of another model. But one can see how this problem has a combinatorially large search space which is best suited to be searched by an optimization algorithm such as evolution. On the other hand, merging in the parameter space evolves new ways of mixing the weights of multiple models. There are an infinite number of ways of mixing the weights from different models to form a new model, not to mention the fact that each layer of the mix can in principle use different mixing ratios. This is where an evolutionary approach can be applied to efficiently find novel mixing strategies to combine the weights of multiple models. Finally, both Data Flow Space and Parameter Space approaches can be combined to evolve new foundation models that might require particular architectural innovations to be discovered by evolution.

Researchers were eager to see how far this automated method can go by finding new ways to combine the vast ocean of open-source foundation models, especially in domains that are relatively far apart, such as Math and Non-English Language, or Vision and Non-English Language. Reported in their paper's experiments, the authors were able to create new open models with new emergent combined capabilities that had not previously existed: a Japanese Math LLM, and a Japanese-capable VLM, all evolved using this approach and achieve state-of-the-art performance on Japanese language and vision language model benchmarks.

Concretely, they first set out to evolve an LLM that can solve math problems in Japanese. Although language models specialized for Japanese and language models specialized for Math exist, there were no models that excelled at -solving mathematical problems in Japanese. To build such a model, they selected 3 source models: a Japanese LLM (Shisa-Gamma) and math-specific LLMs (WizardMath and Abel). In the merging process, the evolution process went on for a couple hundred generations, where only the fittest (the models who score highest in the population on the Japanese math training set) would survive, and repopulate the next generation. The final model is the best performing model (evaluated the training set) over 100-150 generations of evolution, and this model is then evaluated once on the test set.

Table 13.2
Performance Comparison of the VLMs. LLaVA 1.6 Mistral 7B is the source VLM and Japanese Stable VLM is an open-sourced Japanese VLM. While JA-VG-VQA-500 measures general VQA abilities in Japanese, JA-VLM-Bench-In-the-Wild evaluates the model's handling of complex VQA tasks within Japanese cultural contexts. (Table from Akiba et al. 2024)

Model	JA-VG-VQA-500 (ROUGE-L ↑)	JA-VLM-Bench-In-the-Wild (ROUGE-L ↑)
LLaVA 1.6 Mistral 7B	14.3	41.1
Japanese Stable VLM	-	40.5
Akiba et al. (2024)	**19.7**	**51.2**

Table 13.1 summarizes their results. Model 4 is optimized in parameter space and Model 6 is further optimized in data flow space using Model 4. The correct response rates for these models are significantly higher than the correct response rates for the three source models. The authors reported that it was incredibly difficult for an individual to manually combine a Japanese LLM with Math LLMs. But through many generations, evolution is able to effectively find a way to combine a Japanese LLM with Math LLMs to successfully construct a model with both Japanese and math abilities. Notably, the performance of the merged models are approaching those of GPTs and surpassing larger models that are only specialized in Japanese.

In constructing the Japanese VLM, the authors used a popular open-source VLM (LLaVa-1.6-Mistral-7B) and a capable Japanese LLM (Shisa Gamma 7B v1), to see if a capable Japanese VLM would emerge. This was the first effort to merge VLMs and LLMs, demonstrating that evolutionary algorithms can play an important role in the success of the merge.

Table 13.2 summarizes the performance of the merged VLM and the baselines. JA-VG-VQA-500 and JA-VLM-Bench-In-the-Wild are both benchmarks for question and answer about images. The higher the score, the more accurate the description is answered in Japanese. Interestingly, the merged model was able to achieve higher scores than not only LLaVa-1.6-Mistral-7B, the English VLM on which it is based, but also JSVLM, an existing Japanese VLM.

13.3 LLMs Enhances Evolutionary Computing

13.3.1 Evolution through Large Models

In the seminal paper "Evolution through Large Models" (Lehman et al. 2023), researchers delve into the integration of evolutionary algorithms and LLMs, with a focus on the innovative domain of code generation. This study is motivated by the capability of LLMs to significantly enhance genetic programming (GP) by facilitating advanced mutation operations. LLMs, trained on datasets featuring sequential code changes and modifications, are adept at simulating probable alterations that a human programmer might make. This enables these models to guide the evolution of code in sophisticated, contextually aware manners that surpass the capabilities of traditional mutation operators used in GP.

The research team implemented an integration of LLMs with the MAP-Elites algorithm (see more in Section 5.4 to learn about MAP-Elites) within the Sodarace simulator, a platform for evolving robotic models. This approach leverages the code generation capabilities of LLMs to direct the development and refinement of control software for simulated robotic entities. The experiments conducted yielded impressive outcomes, generating hundreds of thousands of functional Python programs that effectively controlled robotic models in various simulated environments. Notably, these tasks were new to the LLMs, highlighting their ability to generate relevant and functional outputs in untrained scenarios. The success of these applications showcases the potential of LLMs to revolutionize evolutionary algorithms, enabling not only optimization but also fostering innovation through the creation of diverse and high-quality solutions.

At the core of the methodological innovation in this paper is the rethinking of the mutation operator, a fundamental component in GP. Traditionally, GP mutations are stochastic, applying random or simple deterministic changes that may not always respect the underlying logic or syntax of the code. In contrast, the approach taken in this work leverages the sophisticated capabilities of LLMs to introduce a "diff" based mutation process which, unlike conventional methods, utilizes the deep learning insights of LLMs, trained on vast repositories of code changes (diffs) from real-world projects (e.g., projects on GitHub). By understanding both the context and the functionality of code segments, LLMs can generate diffs that are not only syntactically correct but also semantically meaningful. These diffs reflect plausible changes that a human developer might make, thus ensuring that each mutation step is both relevant and potentially beneficial, avoiding the pitfalls of random modifications. Figure 13.6 highlights a performance comparison between the diff mutation in this paper and the conventional GP mutation in fixing bugs. The success rate of generating new code that fixes bugs dropped dramatically for the GP mutation, while the diff mutation is able to retain the success rate until encountering the 5th bug in the code. This enhancement of the mutation process, enabled by the sophisticated capabilities of LLMs, highlights the significant benefits that such models can bring to invent more evolutionary algorithms.

The second key methodological aspect is the coupling of the diff-based mutation process with MAP-Elites, an algorithm known for maintaining a diverse set of solutions during the search process. This pairing is particularly potent because MAP-Elites operates on the principle of illuminating the "fitness landscape" by categorizing solutions into different "niches" based on their features and performance. Each niche represents a unique combination of traits, and the goal is to find the best possible solution within each niche. By integrating the diff model with MAP-Elites, the study harnesses the capability of LLMs to propose targeted, intelligent modifications to existing Python solutions. Each modified solution is then evaluated and, if it offers an improvement or explores a new niche, is added to the evolving map of solutions. Over time, this process not only enriches the diversity of the solution space but also enhances the overall quality of the solutions produced. This method enables the systematic exploration of the solution landscape, ensuring that the evolutionary process is both broad (covering many different types of solutions) and deep (refining solutions to achieve high performance).

Finally, the fine-tuning of the diff model represents a critical enhancement. Recognizing the pre-trained LLM diff model, while capable, is not familiar with the Sodaracer task and

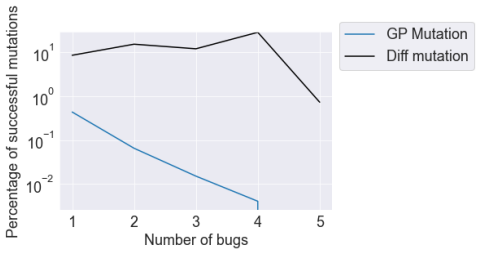


Figure 13.6: **Comparing diff mutation to GP mutation.** Success rate for GP mutation decreases exponentially in the number of mutations, and produces no solutions when there are five bugs. In contrast, diff mutation degrades only with the fifth bug. The conclusion is that LLM-based mutation can indeed make multiple sensible coupled changes to code. (Figure from Lehman et al. 2023)

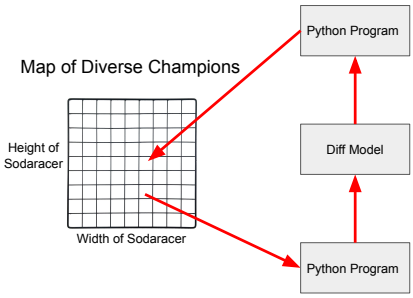


Figure 13.7: **MAP-Elites with diff mutation.** In each iteration, a Python solution is sampled from the archive for each replica of a diff model. Each replica generates a batch of diffs applied to the sampled solution to produce modified candidates. These candidates are evaluated and used to update the archive. Over time, a single seed program evolves into a variety of high-performing Python programs. (Figure from Lehman et al. 2023)

may not be aligned with the specific requirements of evolutionary code generation, the researchers undertook a fine-tuning phase. This process involved training the LLM further on a dataset generated during the evolutionary search process, which comprises targeted code diffs that were particularly relevant to the tasks at hand. By doing so, the fine-tuned diff model could more effectively contribute to the evolutionary search, because the fine-tuning process refined the model's ability to predict and generate code diffs that are not only plausible and syntactically correct but also highly functional within the specific context.

In their experimental setup, the authors initiated the MAP-Elites algorithm with four simple yet diverse seed solutions designed to span a range of foundational geometries. These seed solutions, specifically labeled as the Square seed, the Radial seed, and two seeds inspired by CPPNs, provided a varied starting point for evolutionary exploration. For more detailed information about CPPNs, refer to Section 4.3.1, and see Figure 13.8 for visual illustrations of these seed designs.

As the evolutionary search progressed, it led to the discovery of creatures with novel and complex body designs, synthesized through the advanced capabilities of the program. These innovative designs are showcased in Figure 13.9, highlighting the algorithm's ability to push beyond conventional design boundaries. Furthermore, a detailed behavior analysis of the evolutionary method is provided in Figure 13.10, which presents three critical metrics: the percentage of niches discovered, the QD score, and the percentage of runnable code generated by the diff model. This analysis includes a comparative study between the outcomes using the pre-trained diff model and the model that was fine-tuned during the QD process.

The results demonstrate that even with the pre-trained diff model, the method achieved respectable scores across the evaluated tasks. However, it was the fine-tuned LLM that significantly enhanced performance, underscoring the effectiveness of integrating LLMs with evolutionary computing techniques. This synergy not only boosted the algorithm's efficiency but also its ability to generate highly functional and innovative solutions, thereby showcasing the substantial potential of this integrative approach.

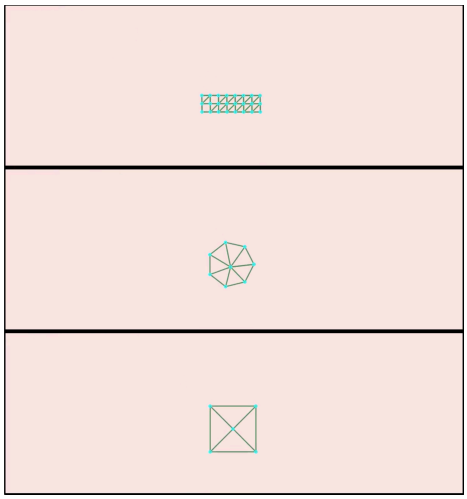


Figure 13.8: **Seed solutions.** From top to bottom: CPPN seed, radial seed, and square seed. (Figure from Lehman et al. 2023)

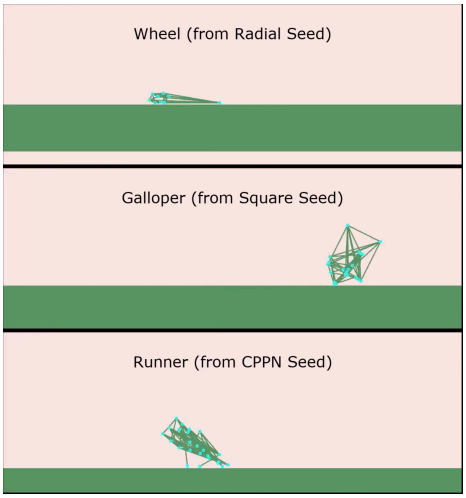
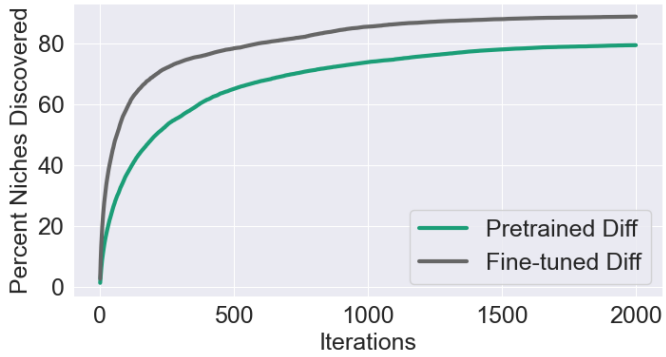


Figure 13.9: **Generalization tests.** From top to bottom: Wheel, from radical seed; Galloper, from square seed; Runner, from CPPN seed. (Figure from Lehman et al. 2023)

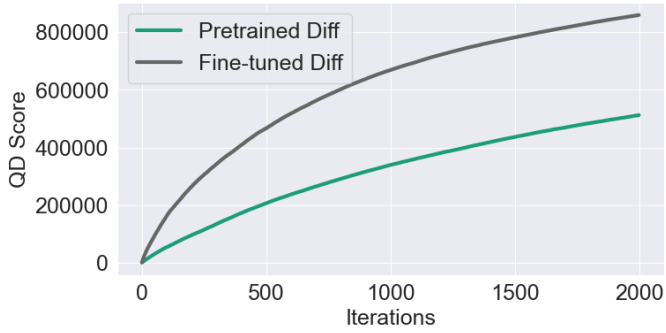
13.3.2 LLM As Evolution Strategies

LLMs trained on extensive text data have shown remarkable abilities in in-context learning. These models, without any additional training and merely through the information provided in context, can not only grasp but also enhance complex patterns. For instance, they can deduce rules from presented sequences and suggest improvements (Mirchandani et al. 2023). This ability to adapt and generate solutions without changing underlying model structures is particularly fascinating and appears to extend to various types of abstract sequences.

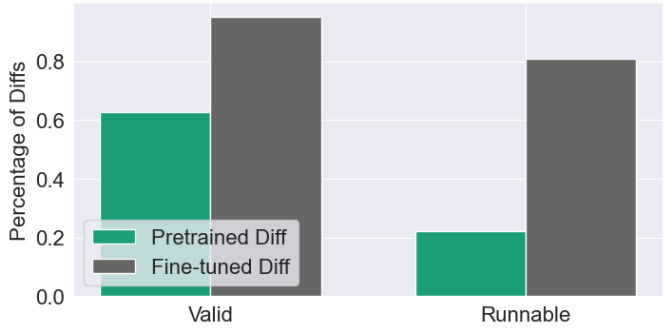
Given their ability to handle complex patterns, a natural question arises: can these text-trained language models function as effective optimization tools? More specifically, can they mimic the workings of evolutionary strategies (ES), where they help evolve the weights of neural networks? In Lange, Tian, and Tang 2024b, the authors' exploration into this started by using LLMs as "general pattern machines" to develop a novel strategy that transforms a standard language model into a recombination operator. This operator processes sequences of function evaluations and their corresponding solutions, and from there, it can



(a) Niches Reached



(b) QD Score



(c) Diff Quality

Figure 13.10: **The impact of fine-tuning the diff model on the performance of ELM.** For both the pretrained diff model and the fine-tuned one, shown are (a) the number of niches reached, (b) QD score of the produced map, and (c) percentage of valid/runnable diffs proposed. The experiments demonstrate that fine-tuning the diff model improves performance of the evolutionary process across all three metrics. (Figure from Lehman et al. 2023)

generate new solution proposals. The strategy they crafted, EvoLLM, involves reimagining the language model as a core component in evolutionary computing. This approach not only asks the LLM to identify potential solutions but actively involves it in the evolutionary cycle, allowing it to suggest optimal sampling points for further evaluation (see the

left part of Figure 13.11). The method also integrates techniques like integer-based search space discretization and the use of decision transformer-style queries to enhance fitness. This innovative prompting strategy has proven effective, as EvoLLM can successfully perform black-box optimization on a variety of BBOB functions (Hansen et al. 2010) and control tasks (see the right part of Figure 13.11), demonstrating that LLMs can indeed act as powerful tools in evolutionary computing.

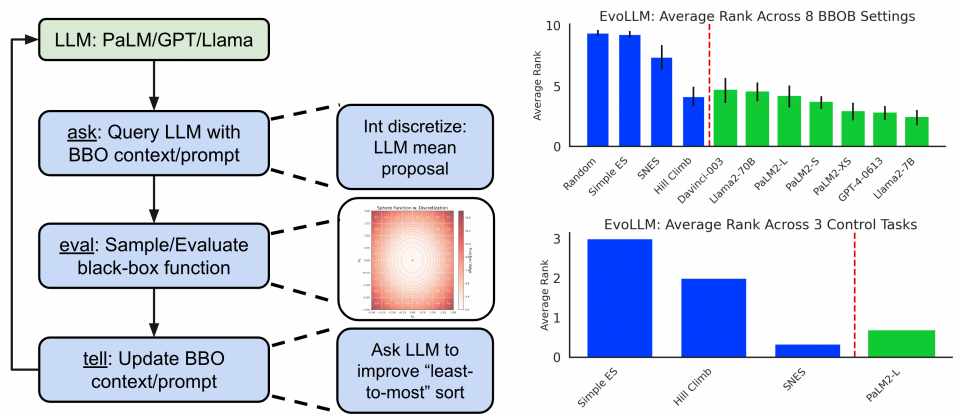


Figure 13.11: **Overview of EvoLLM procedure & Aggregated Results across 8 BBOB settings.** (Figure from Lange, Tian, and Tang 2024a)

Concretely, EvoLLM's design can be described from the combination of a high-level prompt design space (macro-view) and a detailed API space (micro-view), see Figure 13.12 for an illustration.

In the high-level prompt design space, EvoLLM first follows the paradigm established in Mirchandani et al. 2023 and construct an LLM prompt by representing the solution candidates as integers resulting from a discretized search space with a pre-specified resolution. They use integers instead of raw floating point numbers to avoid the difficulty LLM tokenizers face when dealing with non-text data. For example, they return different numbers of tokens per individual number, and this severely prevents EvoLLM from inferring improvement sequences. To construct a query that EvoLLM can better understand and generate improvement efficiently, the authors keep a record of all the population evaluations and sort the set of previous records $H = \{X_g, F_g\}_{g=1}^{G}$ by their fitness within and across generations, here X_g's are the solutions in generation g, and F_g's are their fitness scores. They then select the top-K performing generations and top-M solutions within each generation, and organize them in a formated manner in the LLM's input context. Finally, similar to the design Decision Transformer (Chen et al. 2021), EvoLLM appends a desired fitness level $f_{\text{LLM}}^{\text{query}}$ as the target for the proposal at the end of the input context, see the bottom left light purple box in Figure 13.12 (Prompt 1) for an illustration of the input prompt. Although there are violations, most LLMs robustly follow the pattern outlined in this prompt design and continue the string format by outputing a new mean x_{LLM} with the correct delimiter. The caller of EvoLLM in the user space can then use this as the proposed mean to sample

a new set of candidates and evaluate them in the task to update the records H, and this loop continues.

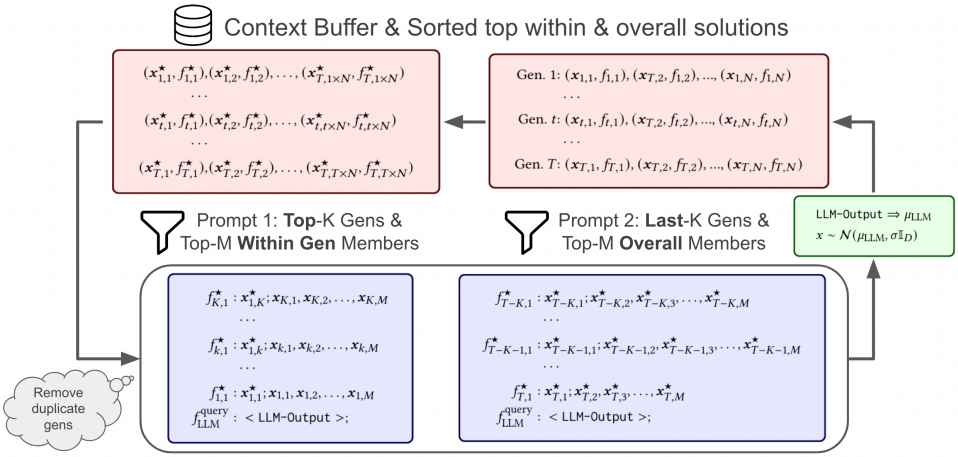


Figure 13.12: **EvoLLM Prompt Design Space & API.** (Figure from Lange, Tian, and Tang 2024a)

EvoLLM includes a set of detailed design choices in the API space, and the list below summarizes the most important ones:

1. **Context Buffer Initialization.** EvoLLM uses random search, a standard BBOB algorithm, to fill up the context buffer as initial solutions and evaluations.
2. **Context Buffer Discretization and Augmentation.** EvoLLM represents the solutions as integers (i.e., remap the inputs and the tokens) and keeps track of the candidates and their fitness scores.
3. **Select & Sort Context Generations.** In addition to the default way of picking the best-performing solutions seen so far, EvoLLM also considers selecting randomly from the buffer or selecting the most recent K generations evaluated on the problem (see Prompt 2 in Figure 13.12).
4. **Select & Sort Context Candidates.** Similarly, besides the default option of taking the "best-within-generation", EvoLLM supports random selection and picking the "best-up-to-generation" options.
5. **Query LLM for Search Improvement.** EvoLLM samples and constructs the prompt repeatedly at each generation. When the generated solution failed to improve the fitness, EvoLLM uses a backup strategy and samples around the previous best evaluted solution.
6. **Sample & Evaluate New Candidate.** EvoLLM samples around the proposed mean x_{LLM}, evaluates all the populations and adds them to the context buffer.
7. **Scale to Larger Search Spaces.** Once the context becomes too long, LLMs start to give non-informative outputs. To avoid this limitation when handling high dimensional data, EvoLLM groups a set of dimensions that fits into the context of an LLM and perform multiple queries per generation. In the extreme case, each LLM call processes a single

dimension d. This trade-off of increased inference time allows EvoLLM to scale to a larger number of search dimensions.

To evaluate the performance of EvoLLM, the authors measured its performance on 4 different tasks from BBOB and compared with standard ES algorithms. The LLM-based ES outperformed random search and Gaussian Hill Climbing with different search dimensions and population sizes. On many of the considered tasks, EvoLLM is even capable of outperforming diagonal covariance ES algorithms. Moreover, EvoLLM is more efficient in generating solutions, i.e. less than 10 generations. EvoLLM's design is generally applicable across different LLMs. In their paper, the authors conducted experiments with Google's PaLM2 (Anil, Dai, et al. 2023), OpenAI's GPT-4 (OpenAI 2023), and the open-source Llama2 (Touvron et al. 2023). An interesting observation is taht the LLM model size inversely affects the performance of EvoLLM - larger models tend to perform worse than smaller models (see the right part of Figure 13.11. Later on, the authors expanded their evaluation to include control tasks CartPole-v1 and Acrobot-v1 from OpenAI's Gym tasks (Brockman et al. 2016), where EvoLLM needs to evolve 16 to 40 parameters of a feedforward neural network that acts as the control policy. EvoLLM was able to evolve the control policy to solve both tasks, and again capable of even outperforming competitive baselines with smaller compute budgets.

The promising results from the evalution of EvoLLM suggests that language model based ES holds significant potential for enhancing optimization processes. As this field is still in its nascent stages, the full scope and impact of integrating LLMs with EC are yet to be fully realized.

13.4 World Models

Deep learning models, in particular, deep *generative models*, are effective tools of learning representations of vast amounts of training data. Furthermore, such models are able to *generate* data to resemble the actual data distribution it learned from real training data, and such models can be primed with relatively low dimensional *latent vectors* to produce rich and expressive outputs.

Given the expressiveness of deep generative models, one can attempt to use these models to learn all about the environment in which an artificial agent interacts with. We call a generative model of the agent's environment a "world model" because like our own internal "mental world model" of the world, an agent can incorporate such a model into its own decision making process.

Figure 13.13: In this section, we explore training generative models of popular visual environments such as CarRacing (Klimov 2016) and VizDoom (Kempka et al. 2016)

In this section, we describe methods and approaches which combine generative models with evolutionary computation. In particular, we explore the approach of using deep learning to train a world model on an agent's environment, and use evolutionary algorithms to train a controller that, as the name suggests, controls the actions of an agent. While much has been done in this area, this section is based on an early paper called World Models (Ha and Schmidhuber 2018), which laid the foundation for much work in this area.

13.4.1 A Simple World Model for Agents

Here, we will describe a simple model inspired by our own cognitive system, to demonstrate how a world model can be used by an agent acting in its environment. In this model, summarized in Figure 13.14, our agent has a visual sensory component that compresses what it sees into a small representative code. It also has a memory component that makes predictions about future codes based on historical information. Finally, our agent has a decision-making component that decides what actions to take based only on the representations created by its vision and memory components.

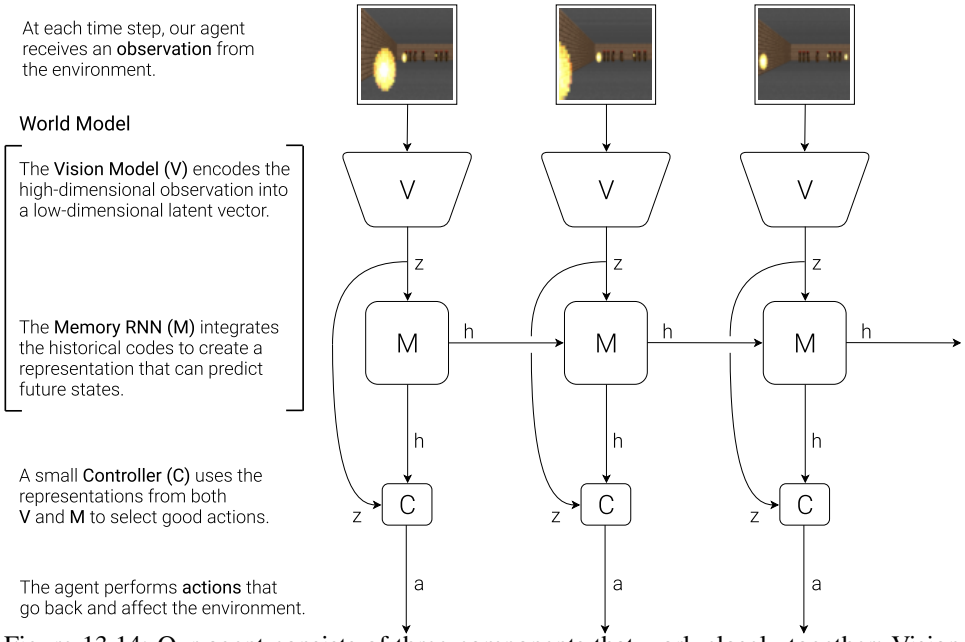


Figure 13.14: Our agent consists of three components that work closely together: Vision (V), Memory (M), and Controller (C).

Vision Model (V)

The environment provides our agent with a high dimensional input observation at each time step. This input is usually a 2D image frame that is part of a video sequence. The role of the V model is to learn an abstract, compressed representation of each observed input frame.

A Variational Autoencoder (VAE) (Kingma and Welling 2013) is used as the V model in our experiments. As shown in Figure 13.15, this VAE model can compress an image frame

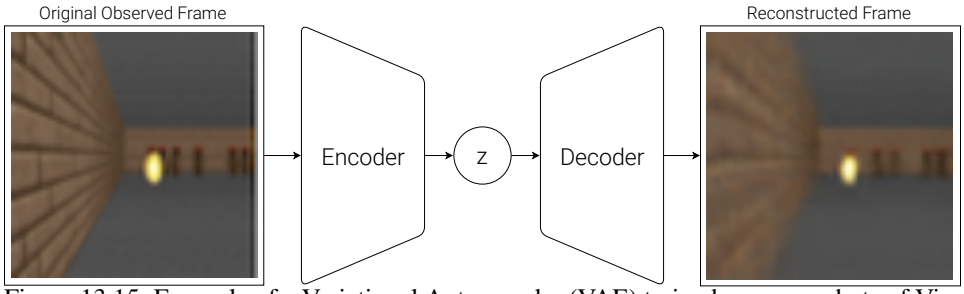


Figure 13.15: Example of a Variational Autoencoder (VAE) trained on screenshots of VizDoom.

into a low dimensional vector z. This compressed representation can be used to reconstruct the original image. In our experiments, the size of this latent vector is 16 dimensions, and is used to represent the spatial part of the agent's environment.

Memory Model (M)

While it is the role of the V model to compress what the agent sees at each time frame, we also want to compress what happens over time. For this purpose, the role of the M model is to predict the future. The M model serves as a predictive model of the future z vectors that V is expected to produce. While many models in the deep learning literature are able to train and predict sequential data, for the works described in this section, we will use simple recurrent neural networks (RNN) to train our RNN to predict the next latent vector z given the current and past information available to it. Given the predictive power of recurrent neural networks, our RNN's internal hidden state vector, h, can be used to represent the temporal part of the environment, and also be considered to be the internal state of our agent, encapsulating our agent's memory.

In our experiments, we gather data from the agent's environment using a random policy and collecting around 10,000 example rollouts, and use this data to train both V and M. Ha and Schmidhuber 2018 also further discuss promising approaches of iteratively gathering data to incrementally improve the agent's world models as the agent interacts with the world. We refer the reader to read the original paper for an accessible guide about the details of the particular VAE and RNNs architectures used in the experiments, and also the training procedures.

Controller Model (C)

The Controller (C) model is responsible for determining the course of actions to take in order to maximize the expected cumulative reward of the agent during a rollout of the environment. In our experiments, we can deliberately make C as simple and small as possible (a small linear layer), and trained separately from V and M, so that most of our agent's complexity resides in the world model (V and M).

The simplest C is a simple single layer linear model that maps t and I directly to action at each time step. We will also explore the effects of making C more complex later on, such as incorporating an additional hidden layer.

Putting Everything Together

Figure 13.16 is a flow diagram illustrates how V, M, and C interact with the environment.

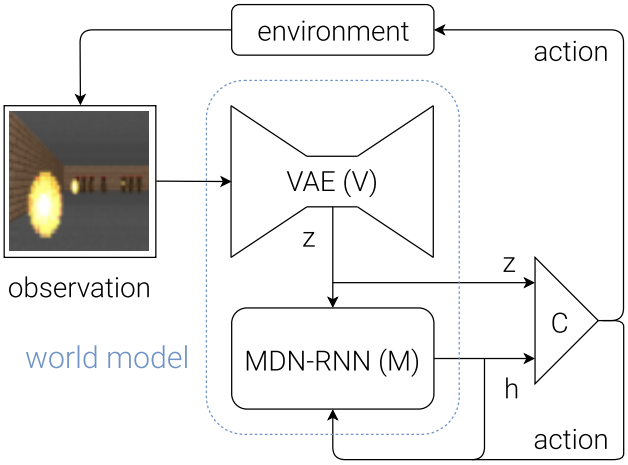


Figure 13.16: Flow diagram of our Agent model. The raw observation is first processed by V at each time step t to produce z_t. The input into C is this latent vector z_t concatenated with M's hidden state h_t at each time step. C will then output an action vector a_t for motor control. M will then take the current z_t and action a_t as an input to update its own hidden state to produce h_{t+1} to be used at time $t+1$.

This minimal design for C also offers important practical benefits. Advances in deep learning provided us with the tools to train large, sophisticated models efficiently, provided we can define a well-behaved, differentiable loss function. Our V and M models are designed to be trained efficiently with the backpropagation algorithm using modern GPU accelerators, so we would like most of the model's complexity, and model parameters to reside in V and M. The number of parameters of C, a linear model, is minimal in comparison. This choice allows us to use very flexible evolutionary algorithms to train C to tackle more challenging RL tasks where the credit assignment problem is difficult.

To optimize the parameters of C, we chose the Covariance-Matrix Adaptation Evolution Strategy (CMA-ES) as our optimization algorithm since it is known to work well for solution spaces of up to a few thousand parameters. We evolve parameters of C on a single machine with multiple CPU cores running multiple rollouts of the environment in parallel Ha and Schmidhuber 2018. We will also explore an additional interesting approach of also evolving the controller network's architecture using NEAT.

13.4.2 Using the World Model for Feature Extraction

A world model contains many useful internal latent information that the agent can leverage as useful features extracted from the environment into the model. These features can even be used entirely for the agent's decision making process, bypassing the direct use of the actual observations from the environment. We will demonstrate this concept using the CarRacing task.

CarRacing is a top-down car racing from a pixel-observation environment. The agent is given a high dimensional pixel frame at every timestep, and is tasked with navigating its car, controlled with three continuous commands (gas, steer, brake) is tasked with visiting as many tiles as possible of a randomly generated track within a time limit.

While it is possible, in principle, to feed the high dimensional input into a large policy network trained to output an action, such an approach requires training the entire neural network using the reward signal to guide changes in the weights, which will require the use of reinforcement learning or evolutionary algorithms applied to train the entire policy network.

By using a world model, we can considerably limit the size and complexity of the policy network. We find that even the vision model, a variational autoencoder, can be quickly trained to compress an entire input frame into a 16-dimensional latent vector z, which is expressive enough to reconstruct the image meaningfully enough for the driving task.

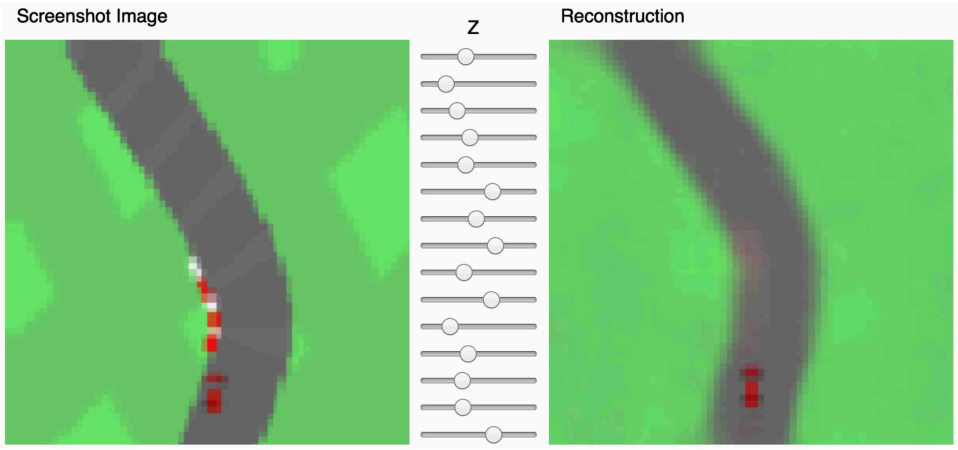


Figure 13.17: A variational encoder trained to compress a frame in CarRacing into a 16-dimensional latent vector z. The frame can be regenerated using the latent vector alone.

By using the vision model (V) alone, without even using the memory model (M), we were able to train a small linear network with 17 parameters (16 latent vectors and an additional bias) to compute the action vector (brake, gas, and steer), which required only evolving only 51 parameters for this simple linear model. The resulting model achieved an average score of 632 ± 251 over 100 trials. While the navigation policy makes the car go a bit wobbly, due to the simplicity of the linear model and the lack of predictive power from using the vision model alone, it does generally do the job of completing most tracks.

If we are confined to only using a vision model (V) but still want to increase the agent's performance, the next step is to beef up our controller, from a simple linear controller, to a controller with one hidden layer, to provide the agent with additional computational capabilities in its decision making. In our experiments, incorporating an extra hidden layer to the controller increases its performance to an average score of 788 ± 141 over 100 trials.

For those who want to stubbornly stick to using the vision model only for this task, and still not satisfied with this result, let's take this to the extreme! We can evolve the network with NEAT. To make things interesting, NEAT here is allowed to use all sorts of different activation functions such as sinusoids, step functions, and ReLUs. Figure 13.18 is the best NEAT network we evolved for our controller, which got us an incredible performance of an average score of 893 ± 74 over 100 trials!

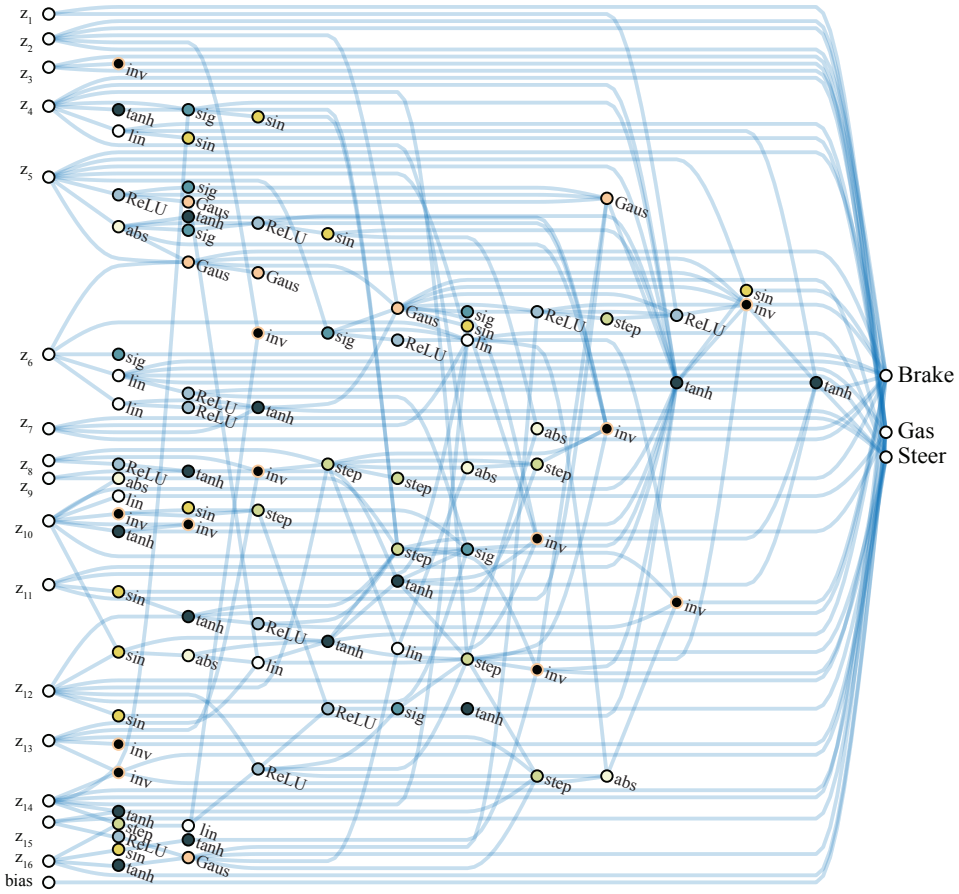


Figure 13.18: Pushing the limits of the Vision-only model by evolving a NEAT controller for CarRacing which only uses the latent vector z of the Vision model to output the action.

Okay, maybe we went too far with going all the way to evolve the architecture of the Controller network. Let's step back a bit, and question whether the performance of a simple linear-only controller can be drastically improved by giving the controller more features to work with. So far, we stubbornly only allowed ourselves to use the Vision model only, which contains only static features representing the spatial properties of the agent's environment, but has no predictive power.

Now let's incorporate the memory model (M) into the agent's world model, and allow our controller to use the full extent of the features from the vision model and the recurrent neural network. We showed that by concatenating the latent vector z from the vision model, and the hidden state h of the predictive recurrent neural network model, our controller achieved the very best performance resulting in an average score of 906 ± 21 over 100 trials, even though this controller is still a simple linear model. At the time of publication several years ago, this model was the first solution to have solved this car racing task, which required an average score above 900. The results for all of the approaches for different controllers (C) is described in the table here.

Controller and Input	Average Score
1-Layer, z	632 ± 251
2-Layer, z	788 ± 141
NEAT, z	893 ± 74
1-Layer, z and h	906 ± 21

Table 13.3
CarRacing scores with various controllers described.

13.4.3 Training an Agent using the World Model as a Neural Simulator of Reality

So far, we have demonstrated the usefulness of using a world model for the purpose of extracting important features that tell the agent useful things about its environment, particularly with spatiotemporal features through the Vision and Memory components of the world model.

But a world model is far more useful than being merely a feature extractor. If we are interested in feature extraction alone, there might be more direct ways of training neural networks for that purpose. The key capability of a generative world model is the ability to generate and simulate the actual environment, in latent space, kind of like running a quick simulation in our minds. For instance, the memory component of our world model, the recurrent neural network, is able to simulate approximate future trajectories of the environment from the data the agent has collected.

The agent, in principle, can even act inside this neural-network simulated environment of the world, and observe hypothetical responses, and learn from the consequences of its actions without actually performing such actions in reality. We believe this aspect of running mental simulations inside of the agent's world model is its most promising feature.

Figure 13.19: The VizDoom: Take Cover environment.

In the World Models work, we demonstrate this ability in an experiment that uses a VizDoom environment, which uses parts of the Doom video game. In our particular environment called VizDoom-TakeCover, which takes place in a closed virtual room, the agent's task is to learn to avoid fireballs shot by monsters from the other side of the room with the sole intent of killing the agent. There are no explicit rewards in this environment, so to

mimic natural selection, the cumulative reward can be defined to be the number of time steps the agent manages to stay alive during a rollout. Each rollout of the environment runs for a maximum of 2100 time steps (roughly a minute of actual gameplay), and the task is considered solved if the average survival time over 100 consecutive trials is greater than 750 time steps of gameplay.

To train our world model, like the CarRacing experiment, the agent explored the environment using a random policy, and recorded trajectories over thousands of random gameplays. Once the world models are trained, our agent is able to produce simulated gameplays in latent space, using the RNN module alone. Interestingly, since the vision model is already trained, one can even play inside of the world model as a human player with a keyboard, and we invite interested readers to try the demo referenced in Ha and Schmidhuber 2018.

The recurrent neural network is trained to produce not a deterministic prediction of the next latent states of the world, but a probabilistic distribution which can sample future latent states. As such, this distribution can be parametrized to artificially produce wider or narrower distributions using a temperature parameter τ. This allows us to bias the distribution to output the mode always, or produce outputs with more uncertainty, and this feature is quite important for training an agent entirely inside the world model. This table displays the results when we use an evolutionary algorithm (CMA-ES) to train a controller to perform well inside the world model, and how the policies learned transfer to the actual environment.

Temperature τ	Virtual Score	Actual Score
0.10	2086 ± 140	193 ± 58
0.50	2060 ± 277	196 ± 50
1.00	1145 ± 690	868 ± 511
1.15	918 ± 546	1092 ± 556
1.30	732 ± 269	753 ± 139
Random Policy	N/A	210 ± 108

Table 13.4
VizDoom: Take Cover scores at various temperature settings.

We note that in the deterministic model (low temperature), the agent can easily find faults in its model of the world, and exploit them so that the learned policy will only do well in its dream, but not in reality. In contrast, as we increase the uncertainty of the model, this makes the virtual environment generated by the agent's world model much more difficult to beat, leading to policies that are transferable to the actual environment. Varying the temperature in generation is just one of several possibilities of approaching the transfer problem between performing a task inside a learned world model and performing a task in the actual world.

The experiments outlined in this section describe only the simplest methods of combining generative world models with evolutionary algorithms. There are great opportunities to extend this concept, such as the aforementioned approach of iterative data collection for refining the world model, and incorporating other approaches in the evolution literature such as novelty search or multi-agent settings. We invite the reader to explore the limitless possibilities.

13.5 Chapter Review Questions

1. **Large Language Models (LLMs):** What role does the transformer architecture and self-attention mechanism play in the performance and scalability of Large Language Models (LLMs) like GPT?
2. **Evolutionary Prompt Engineering:** How do evolutionary algorithms, such as those used in EvoPrompt, enhance prompt engineering for LLMs? Why are these methods particularly suitable for black-box APIs?
3. **Promptbreeder:** What is the self-referential mechanism in Promptbreeder? How does it differ from EvoPrompt in optimizing task-specific prompts for LLMs?
4. **Performance of EvoPrompt:** How did EvoPrompt improve performance on challenging tasks like the Big Bench Hard (BBH) benchmark? What are the key contributions of the Genetic Algorithm (GA) and Differential Evolution (DE) approaches?
5. **Evolutionary Model Merging:** What are the key differences between merging models in data flow space and parameter space? How does evolutionary model merging generate new composite models with emergent capabilities?
6. **LLMs in Genetic Programming:** How are LLMs utilized in enhancing genetic programming through "diff-based mutation"? What advantages do these mutations offer over traditional random or deterministic approaches?
7. **EvoLLM as Evolutionary Strategies:** How does EvoLLM transform an LLM into an evolutionary strategy operator? What unique design choices make it effective for optimization tasks such as BBOB and control tasks?
8. **World Models:** What are the roles of the Vision (V), Memory (M), and Controller (C) components in world models? How do these components collectively allow agents to act effectively in simulated environments?
9. **Using World Models for Feature Extraction:** In the CarRacing task, how does the use of latent features from the vision model reduce the complexity of the controller? What are the advantages of evolving controllers with NEAT?
10. **Simulated Learning with World Models:** How do world models enable agents to train within a neural simulator of reality, as demonstrated in the "VizDoom: Take Cover" environment? How does adjusting the temperature parameter influence policy transfer to the actual environment?