Companion Chapters to AI for the Young and Curious, 2nd Edition

David Wu

Companion Chapters to AI for the Young and Curious 2nd Edition

For Mom and Dad

Contents

Preface

These three chapters are companion chapters to AI for the Young and Curious, 2nd Edition. They cover topics that were not included in the original book, but that I feel are important in modern AI. Hope you enjoy!

David Wu July 27th, 2025

Chapter 1

Reinforcement Learning

In the introduction chapter, we talked about the three main types of machine learning: supervised learning, unsupervised learning, and reinforcement learning. For most of this book, we discussed supervised learning, though we also briefly touched on clustering as a form of unsupervised learning in the Waymo chapter. In this chapter, we will focus on reinforcement learning.

In recent years, reinforcement learning has found more and more applications, from robotics to gaming and even in large language models as we will see later. Reinforcement learning has contributed to some of the most recognizable products, such as AlphaGo and ChatGPT. So what is it?

1.1 A Motivating Example

Let's start with a motivating example: suppose that you want to train an agent to play Pac-man. You've already set up the data processing so that the agent will take the raw Pac-man screen as input and output one of the joystick controls of up, down, left, or right. If we tried to train our agent using supervised learning, we would need a differentiable loss function that allows us to calculate how exactly changing one parameter of the model would impact the loss.

This is difficult because we need to turn the predicted probabilities of different actions into a single discrete action, which makes this step non-differentiable. Put another way, if we increased the probability of one action by 1%, we would not be able to definitively quantify

how this would impact the chosen action or the final reward given for taking the action. For example, we have no easy way of determining how exactly increasing the probability of each action by 1% would lead to greater or smaller reward. What if going up instead of down led to possibly better rewards 100 steps down the line? Furthermore, we don't know what the correct actions even are.

Instead, all we do have is a series of rollouts: where we can log actions taken by our agent while playing the game, as well as the reward given at various steps. Our goal is to maximize the total reward obtained by the agent. We thus need to develop a method to use this reward information to train our policy. That's where reinforcement learning comes in.

1.2 Environments

The formal definition of the RL problem centers on the *environment* and the *agent*. The agent is the decision-maker that we are training, and observes states from the environment. The agent then predicts an action based on the observed state, and takes the action in the environment. The environment returns a reward for the current step and also determines whether the episode ends. In our Pac-man example, the agent is the player, the state would be the image of the current state of the game, the action would be moving the joystick right, left, down or up, and the reward would be the score added for each action taken. The episode would end when Pac-man loses all of its lives.

Our goal is then to find a policy to control the agent that maximizes the total reward across episodes. More formally, if we denote the state as s, action as a, and reward as r for T steps in our episode, our goal is to find a function π where $a = \pi(\text{s})$ and maximize $\sum_{i=1}^{T} r_i$. Sometimes we value reward earlier in the episode more than later in the episode (e.g. if we want the agent to finish the game faster), so we add a discount factor that reduces the importance of reward from more steps into the episode and maximize $\sum_{i=1}^{T} \gamma^i r_i$.

1.3 Policy Gradients

Intuitively, a simple strategy would work as follows: if the game only ahd one step, we could simply randomly try all of the different possible

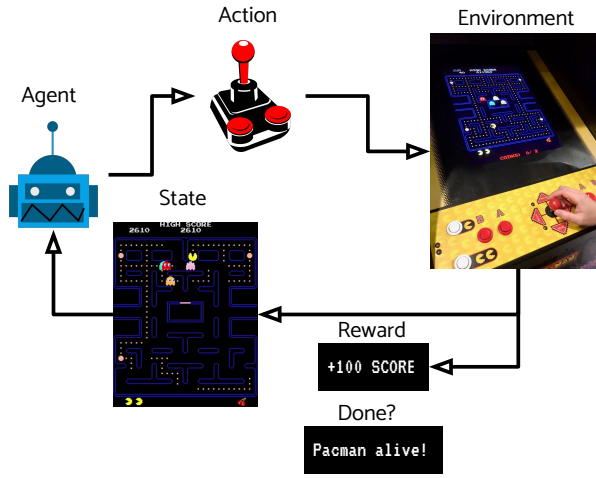

Figure 1.1: An RL Environment. "Pac-Man Gameplay" by Bandai Namco from is licensed under the Creative Commons Attribution 4.0 International license. Links: [1].

actions and look at the rewards from all of the actions. We could then upweight the actions that gave higher reward and downweight the actions with lower reward. This strategy allows us to learn a final policy that takes actions to maximize reward.

We can extend this strategy to multistep problems by upweighting or downweighting each action based on the total discounted reward obtained after taking this action (as this is all of the reward that would be affected by this action). We can then calculate this "value" of the action for every action in an episode and determine the rates at which different actions should be more or less likely. Note that this is imperfect because it's not correct to assume that every action before a good outcome is a good action or vice versa. But averaged over many trajectories, we may be able to learn through the noise and separate the good actions from the bad.

Finally, note that if we continue to use a random policy to rollout and obtain trajectories, it may be difficult to learn the best policy possible. This is because the random policy's actions may prevent learning the right actions in areas only visited by better policies (e.g.

if we keep on using a random policy, it's quite possible that the policy will never visit a hidden room). Therefore, we instead continually roll out using the current policy, which is constantly improving and exploring areas around its current rollout distribution.

Putting all of these pieces together, we come to our first major class of RL algorithms: policy gradient algorithms. The main idea behind policy gradient algorithms is that we can turn a non-differentiable objective into a differentiable objective by using rewards to upweight and downweight the probabilities of taking certain actions.

Specifically, we can mathematically lay out a simple policy gradient algorithm as follows. Given a trajectory of length T with states $s_{1...T}$, actions $a_{1...T}$, and rewards $r_{1...T}$, we can define the return of each step's action as $G_t = \sum_{i=t}^{T} \gamma^{i-t} r_t$. We can then update policy parameters θ_π for each step t with the formula $\theta_\pi + \alpha \gamma^t G_t \nabla_{\theta_\pi} \ln(\pi(a_t|s_t))$. Looking at this formula intuitively, we can see that we are trying to increase the log probability (denoted by $\pi(a_t|s_t)$) of actions based on their return G_t. Greater G_t means that an action will be upweighted more, so actions leading to higher future reward are incentivized. Furthermore, as the action probabilities of the policy are refined throughout learning, the policy will be able to reach higher reward areas and refine its exploration in those areas (e.g. it will find the hidden room and explore from there).

1.4 Q-values

While the previous class of policy gradient methods that we discussed focuses on learning a policy that maps states to actions, another class of methods focuses on learning the values of different state-action pairs in order to make a decision. Specifically, one class of methods learns a Q function to estimate the future value of being in a state and taking a certain action, denoted as $Q(s, a)$ assuming that the optimal actions are taken in the future. Once we are able to successfully learn this function, we can then choose the action that maximizes the Q value and take that action. An added benefit of this class of methods is that we can *bootstrap* Q-values, meaning that once we have learned the Q-value for one state, we can use that information to update the Q-values of other states.

Specifically, we can define Q-values as[1]:

$$Q(s_t, a_t) = r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \tag{1.1}$$

Looking at this equation intuitively, we can see that the Q value of the current state and action depend not only on the reward at the current step, but also on a discount of the value from future states. The Q-value from the next state is also defined in terms of the Q-value from the state after that, and goes until the end of the trajectory. This formulation allows us use information more efficiently as we are using information about different states and actions to inform previous decisions. For example, if you haven't taken a particular route to the supermarket before, but you see that an exit on the highway that leads to a road that you are familiar with, you can use that prior information from other routes to inform you that this exit is likely going to lead you to the right place.

Since our objective is to make the Q-values satisfy the equation above, we can define our final loss function as:

$$L = (Q(s_t, a_t) - (r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})))^2 \tag{1.2}$$

When we finish training our Q function, we can use the Q function to roll out by selecting an action that maximizes the Q value in a given state.

1.5 Imitation Learning

Our previous *exploration* of reinforcement learning primarily focused on problems where we only have access to an environment where the agent can roll out and explore. But in many problems in the real world, we may have some example trajectories from experts (such as humans) that we may want to imitate. For example, if we are trying to teach

[1] Note that this definition is primarily applicable for discrete tasks (i.e. tasks where there is a fixed number of possible actions). For continuous action spaces, finding the best possible action is more complicated and requires more complicated algorithms such as Soft Actor Critic.

a robot to make a dish, we may be able to show the robot a demo of a human chef making the dish first. This problem is called *imitation learning*.

One way to approach this problem is to formulate it as a supervised learning problem. In reinforcement learning, we don't have access to the "correct actions", so we can't train the model to memorize these correct actions. But in imitation learning, we do have these actions, so we can just imitate them, right? No.

The issue is in something called *covariate shift*. Essentially, if we directly imitate the expert's actions, we will have some error ϵ because we are not able to precisely replicate their actions or because of randomness in the environment. That leads us to be slightly off of the expert's trajectory. Being in this slightly off state then leads to a greater error as our model was not trained on this state and thus is more likely to make a mistake. These errors then compound until we are completely off of the original expert trajectory.

To fix this, we need to take advantage of the fact that we not only have access to the expert trajectories, but that we also have access to the environment. One method to address this issue is *inverse reinforcement learning*, where we learn two separate models: one is a reward model which is optimized to increase reward for the expert trajectory and decrease reward for the policy's trajectories, while the policy tries to optimize the reward function using the reinforcement learning algorithms we discussed earlier in this chapter. This method intuitively make sense as we are essentially disincentivizng any behavior that deviates from the expert, and crucially allows the policy to learn how to recover and get back to the expert's states and actions, rather than increasingly deviating from the provided expert trajectories.

1.6 Interesting Applications

Reinforcement learning has many interesting applications and has only become more important in recent years.

One such application is robotics. In robotics, there is no clear differentiable loss function that we can optimize. Instead, we can define reward functions based on goals that we would like the robot to accomplish, such as successfully picking up the object or reaching a certain goal destination. We can use reinforcement learning to learn

policies that maximize arbitrary reward functions and thereby learn desired behaviors.

Another application is large language models. Reinforcement learning has been used to align large language models to human feedback, by using human preferences between different responses in order to train reward models and then using reinforcement learning to optimize these learned reward functions.

Reinforcement learning is a powerful tool in the modern machine learning practitioner's toolbox. Have fun training policies!

Chapter 2

Diffusion Models

You've probably seen generated fake pictures online, often of celebrities, politicians or other public figures. It's difficult to tell if these images are real or not, and this is a testament to the power of diffusion models. In this chapter, we will explore how diffusion models work including both training and inference, their applications to not just image generation, but also other useful tasks.

2.1 The problem of image generation

While most of the problems that we have discussed in this book have been *discriminative problems*, where our goal is to predict the correct answer y given an input x, image generation is a *generative problem*, where our goal is to generate a new image x given some condition c. There is no single correct solution, but rather a distribution of possible correct solutions that would all satisfy the condition. For example, if we want to generate an image of a cat, there are many different possible images that could be considered correct, as long as they all look like cats. We could generate different breeds of cats, cats in different poses, or even cats with different colors. The goal of generative models is to learn this distribution of possible solutions and be able to sample new generated images from it.

In most cases, the conditions that we care about are that:

1. The generated image is realistic.

2. The generated image satisfies a text prompt that we provide, such as "a bear driving a car".

Now, how do we do this?

2.2 Adversarial Methods

One class of techniques developed to solve the problem of image generation is *adversarial methods*. The intuition here is that given that we want the generated images to be indistinguishable from real images, we can simply train a model that does this job for us, trying to discriminate between real and generated images. This model is called a *discriminator*, and we can train it using a binary cross entropy loss function, where the positive examples are real images and the negative examples are generated images. The discriminator will learn to output a high score for real images and a low score for generated images (note that the reverse is equally correct, we just want to separate the two).

We also simultaneously train a *generator* model that tries to generate images that fool the discriminator into thinking they are real. The generator is trained with a loss that back propagates through the discriminator to maximize the discriminator's score for the generated images and make it feel that the image is real. Through this back and forth process, the generator incorporates the discriminator's feedback and learns to generate images that are indistinguishable from the provided set of real images.

This all sounds great on paper, and generative adversarial networks have indeed been able to generate some realistic images. However, the adversarial nature of the training process leads to several issues: first, it's very difficult to keep the generator and discriminator balanced with each other. If one side is given too many update steps, it's quite possible that this side will "win" and the other side will not be able to learn anything useful that fools the other side. Furthermore, researchers have observed a phenomenon known as *mode collapse*, where the generator learns to generate a small set of high quality images that fool the discriminator, but does not learn to generate the full distribution of images. This leads to a lack of diversity in the generated images, which is not what we want. We wouldn't want to only be able to generate one cat or one house.

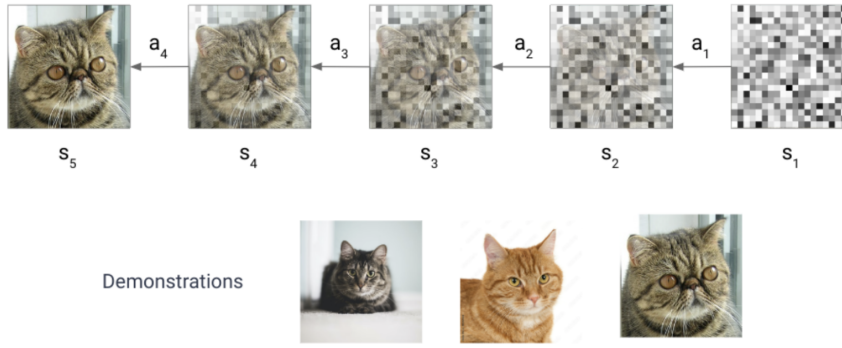

Figure 2.1: The diffusion process.

2.3 How do diffusion models work?

Diffusion models have emerged as a powerful alternative to generative adversarial networks. The intuition behind diffusion models is that in order to learn the distribution of images, we don't need to adversarially train a discriminator to give feedback. Instead, we can simply learn a process to gradually generate images from noise by teaching a model to map noisy images to slightly less noisy images that slowly move towards a final image.

2.3.1 Training

Specifically, in order to train a diffusion model, we need to first be given a set of real images composing the distribution that we want to learn. We then iterate through batches of our dataset and add noise to each image, teaching the model to predict the noise that was added and thereby allowing it to denoise images and obtain real images. More specifically, if x_0 is a real image, we can add a noise image z to it to obtain a noisy image x_t at time step t. The noise z is sampled from a normal distribution. We then train a model f_θ to predict the noise z given the noisy image x_t, such that $f_\theta(x_t) \approx z$. We can do this by minimizing the mean squared error (MSE) loss between the predicted noise and the actual noise added to the image.

It's important to note that when we add noise to the image, we

vary the degree of noise that is added at various levels compared to the original image. This ranges from adding a small amount of noise that is barely visible to adding so much noise that the original image is completely invisible. We train the model to reverse the process at *all of these different steps*, which sets it up to be able to reverse the process starting from the beginning during inference.

2.3.2 Inference

Once we have successfully trained our diffusion model, we can use it to generate new images. We do this by starting with a completely random image of noise and gradually asking the diffusion model to predict the noise that was added at each step. We then subtract a small amount of this predicted noise and move slightly closer to a real image. We repeat this process for a set number of steps according to the schedule and based on the magnitude of noise that we are subtracting. Finally, we end up with a generated image that should, theoretically, be drawn from the distribution of real images.

2.4 Other applications

Diffusion models have made their way into many modern image generation tools which are widely used, such as Stable Diffusion, Midjourney, and the DALL-E series from OpenAI. They have also been extended into other modalities, such as video generation. Images generated by diffusion models have become more and more realistic and have flooded social media as an easy, low-cost way to generate content.

But aside from the applications of generating media, diffusion models as a general method for generating samples from given distributions have also been applied in order fields and tasks.

For example, diffusion models have been used to generate new protein structures, which allows biological researchers to go beyond the set of proteins already present in human genomes in order to manufacture new proteins that serve various functions, such as fighting diseases or improving health. In fact, some of this work, such as the protein diffusion method RFDiffusion, has won the 2024 Nobel Prize in Chemistry, awarded to Professor David Baker at the University of Washington.

Diffusion models have even been extended to generate text, even though diffusion traditionally generates continuous data such as images rather than discrete data such as text tokens. Proponents of these methods have argued that diffusion models are a better alternative to traditional autoregressive models given that they can generate a lot of text at the same time (in parallel) rather than having to generate one token at a time. Diffusion text generation methods also allow for modifying earlier text given changes in later text, which is not possible with many existing decoder-only LLM architectures.

Though overshadowed by autoregressive transformers, diffusion models have distinguished themselves as swiss army knives with many different applications in image and text generation. It's clear that there will be many more exciting applications to come. Hopefully you can be a part of this!

Chapter 3

Large Language Models

Alas, large language models!

It's hard to find anyone these days who hasn't interacted with at least one large language model and even harder to find someone who hasn't at least heard of them. They are in TV commercials, in the news, on the internet, and even in schools. Large language models (LLMs) can be used in a wide variety of day-to-day tasks, such as writing emails, summarizing long documents that no one wants to read, helping to debug code, or providing advice. In this chapter, we will be going beyond how to *LLMs* and instead delve into how they work and how we can make them more effective.

3.1 What are large language models?

First off, what are large language models exactly? At the most basic level, large language models are deep learning models that are able to generate *language* and have a lot of parameters (typically in the billions or even trillions), hence the *large*. The ability to generate text can lead to many other abilities, such as generating code, solving math problems, or writing novels. There are many different ways of classifying large language models into different categories. We will go over a few below:

1. **Base vs. Fine-tuned**: While base language models are only trained directly on the internet and therefore simply complete your text prompt whatever they believe will most likely come

next (e.g. asking another question after you ask it a question), fine-tuned models are further supervised to perform specific tasks for users, which also often includes supervising them to respond to prompts given in certain formats. Almost all of the large language models you interact with on a daily basis are fine-tuned models, such as ChatGPT, Claude, and Gemini.

2. **Next token prediction vs Masked language modeling**: Some language models are trained to predict the next token given access only to all previous tokens, while other language models are trained to predict a masked token given access to all other tokens in the sequence (including some before and some after the masked tokens). The former is known as *next token prediction* and is used by autoregressive models such as GPT-3, while the latter is known as *masked language modeling* and is used by BERT from Google. Today, most LLMs you interact with on a daily basis are autoregressive models, such as ChatGPT, Claude, and Gemini, though masked models can also be used for other tasks, such as embedding text.

3. **Autoregressive vs. Diffusion**: While some models are autoregressive and generate text one token at a time, other models generate all response tokens at the same time. Both types of models have their advantages, as we discussed in the diffusion models chapter.

4. **Transformer vs State space models**: You may have heard of the "Attention is all you need" paper, which introduced the transformer architecture (then used for machine translation), which is now used for almost all large language models. However, the transformer architecture also has its disadvantages, one of which being that the computational cost of the attention mechanism scales quadratically with the number of tokens in the input. This means that it is more difficult for transformers to handle long texts as it requires a lot more computational resource. State space models, on the other hand, are a new class of models that have been shown to be able to handle longer texts more efficiently. For more information, check out the Mamba model.

In this chapter, we will focus on autoregressive transformer models,

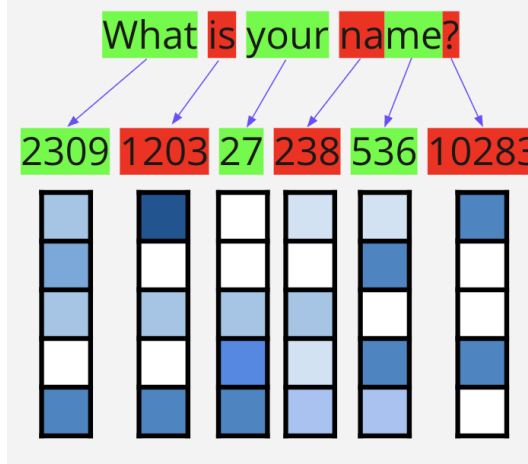

Figure 3.1: The process of tokenization and embedding text.

as they are the most widely used and most relevant to the current state of large language models.

3.2 How do they work?

The first step in understanding large language models is to understand tokenization. Tokenization is the process by which raw text is transformed into a sequence of discrete token numbers that our large language model can process and understand. However, this still isn't enough, as we cannot operate on these discrete token values (e.g. add two token numbers). Therefore, we transform the token values into embedding vectors representing each token, which we then pass into our main transformer model.

Next, the embeddings go through a series of attention layers, where each token interacts with all other tokens before it in the sequence. This allows the model to understand interactions between tokens in the text and gain meaning from the text. For example, it might understand that in the sentence "Sally went to the beach with her dog.", that the word "her" refers to "Sally".

Finally, we take the embeddings from the last layer of the model and apply a linear layer to predict the next token in the sequence.

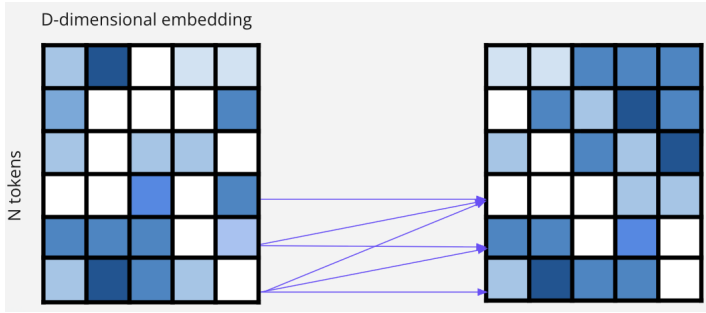

Figure 3.2: A high-level overview of the transformer architecture.

Since we have gone through multiple layers of processing, the hope is that the model has understood all of the previous text and can make an educated guess about what tokens comes next.

3.3 Pre-training

Now, how are large language models trained? Most autoregressive transformer models are trained using a next token prediction objective. The goal is for the model to be able to predict the next token in a sequence given only the previous tokens. We can formulate this as a classification task where the model needs to predict the correct token class out of a collection of thousands or tens of thousands of possible tokens. Through this process, the model needs to learn to understand what is going on the previous text to understand what comes next, thereby allowing the large language model to develop a general understanding of language.

Where does the data for this training come from? Typically, the data for large language model pre-training comes from the internet, books, media, video transcripts, and other sources of text. Modern large language models have billions of parameters and are trained on trillions of tokens taken from the internet, covering many different topics, cultures, and languages. This allows pre-trained language models to be exposed to a level of information that we as humans cannot possibly ingest.

3.4 Post-training

As we discussed earlier, pre-trained large language models aren't suitable for consumer use, as they simply seek to predict what is likely to come next if this sequence of text were on the internet or in a book. For example, if we ask a pre-trained model "What is the capital of France?", it may respond with "What is the capital of Germany?" or "What is the capital of Spain?" as it may have seen quizzes or tests with sequences of questions and does not understand that we want it to answer our question. Instead, we need to fine-tune the model in a process known as *post-training*. Post-training turns a pre-trained model into a model useful for consumers.

3.4.1 Supervised fine-tuning

One method for post-training is supervised fine-tuning (SFT). To perform SFT, we need to have a dataset of questions and the corresponding correct answers. We then train the model to predict the answer tokens when prompted with the question, as a typical classification problem. The difficulty with this method is that it requires human labelers to write the best responses to the prompts, which can be difficult, especially for complex questions such as rigorous proofs. Nevertheless, supervised fine-tuning is often used as the first step in post-training and gives the model a good starting point where it can respond to the user's requests.

3.4.2 Reinforcement Learning from Human Feedback

Next, another common post-training method is reinforcement learning from human feedback (RLHF). The goal of RLHF is to take an LLM possibly already fine-tuned through SFT and further adapt it to be aligned with human preferences. In RLHF, we don't necessarily need a set of human ground-truth answers in order to train. We can instead ask the LLM to generate multiple responses to a given prompt and ask human labelers to give their preferences about which responses are better or worse. This is often a lot easier than writing the best response (e.g. it's a lot easier to read a novel and determine whether it's good or bad than to write a novel). After obtaining these human preferences, we can then train a reward model that predicts higher

scores for responses that humans prefer and lower scores for those that humans do not prefer. We can then optimize the LLM to maximize the reward model's score using reinforcement learning algorithms, thereby ultimately upweighting the probabilities of good responses favored by humans and downweighting the probabilities of bad responses.

3.5 Prompting

Though fine-tuning is an effective method for ensuring that large language models are effective and useful for consumers, they can also be expensive. For many tasks after the initial instruction tuning phase, we don't need to completely fine-tune model. For example, if we want to adapt an instruction tuned model to perform a specific task of solving math problems or diagnosing a patient's disease according to our guidelines, we can use another technique: *prompting*. Unlike fine-tuning, which modifies the model's weights, prompting only adds specific text of your choice into the input to the model. This input can serve a similar purpose of guiding the model to generate responses that you prefer. There are a gazillion different prompting techniques, but we will go over a few of the most common ones below.

3.5.1 Few-shot prompting

Say that you have access to an instruction-tuned large language model, but you want it to respond to all questions with exactly 10 words. You tried asking it to do so, but it still doesn't. You could collect a dataset of questions and answers with exactly 10 words, or do reinforcement learning with a reward that gives a high score for responses with exactly 10 words. But that would take a lot more work. Instead, you might want to try few-shot prompting.

The basic idea of few-shot prompting is "show, not tell". If you want the LLM to respond in a certain way, you can show a few examples of how to do this in the prompt. For example, if you want it to respond with exactly 10 words, you could use a prompt as follows:

```
You are a helpful assistant. Please answer the following questions wi

Examples:
Question: What is the capital of France?
```

```
Answer: The capital of France is Paris, a beautiful city.

Question: Who was Robert Frost?
Answer: Robert Frost was an American poet known for nature poems

New question: What is a weak acid strong base titration?
```

The intuition is that once the model sees how to do the task that you are asking, it is able to more effectively extrapolate from the examples and generate a response that satisfies your conditions.

3.5.2 Chain-of-thought prompting

Next, another common task that we might want an LLM to do is solve a math problem. If we directly input the math problem into the LLM, it may sometimes say something like this:

```
Question: Mary has 23 apples and John as 7. Mary gives 5 apples
Answer: Mary has 29 apples.
```

This not desirable because a) it's quite possible that because the model did not think or show work, it got the answer wrong, as in this case, and b) because it doesn't show work, we cannot verify its answer or learn from its thought process, and have no trust in its work. Instead, we want the LLM to show its work and reasoning step by step.

To accomplish this, we can use a technique known as *chain-of-thought prompting*. The idea is to force the LLM to think in a step by step process by automatically adding a single sentence to its response and asking it to continue the response. For example, we can use the following prompt:

```
Question: Mary has 23 apples and John as 7. Mary gives 5 apples
Answer (we inserted): Let's think step by step.
Answer (LLM completed): 1. Mary starts with 23 apples.
2. She gives 5 apples to John, so she has 23 - 5 = 18 apples.
3. Then she buys 10 more apples, so she has 18 + 10 = 28 apples.
Therefore, Mary has 28 apples now.
```

Now, we can easily verify that the LLM's answer is correct. This class of prompting techniques that forces the LLM to think and use

reasoning processes, as well as output more "thinking" tokens before coming to a final answer has been applied to many difficult problems, even contributing the recent success of LLMs in math and coding competitions.

3.5.3 Retrieval Augmented Generation

Finally, one of the biggest bottlenecks on relying exclusively on the large language model's weights is that it can only use information that was in its pre-training data. If you want it to reference your company's internal documents or use the latest news, it will not be able to do so and may instead hallucinate information that is not true or tell you that it does not know. To solve this, we can give the LLM access to your information and documents and allow it to search through and find the information that is most relevant, using it to formulate a final response that is most useful.

This is known as *retrieval augmented generation* (RAG).

1. First, we take the set of documents and embed each document into a single vector that abstractly represents the information in the document.

2. Next, the LLM queries the set of documents by outputting a text query (e.g. "stock market news"), and we turn this into an embedding as well.

3. We search over the documents, calculating the distance between the query embedding and all of the document embeddings, finding the most relevant documents.

4. Finally, we take the text of the document and put them into the LLM's context window, allowing it to access the information in the documents and use it to generate a final response.

This process allows the LLM to access the latest information on the internet even when it has not been trained on the information, and also allows it to ground its responses in accurate external documents. Often, we can be unsure about whether an LLM's information is accurate, but using retrieval augmented generation, we can allow the LLM to use information from trusted sources and also examine the sources ourselves to verify the information.

3.6 Future Directions

LLMs are a *very* exciting field. There are tons of possible new applications in the future and it's not possible to go over every single one here. However, some interesting applications include:

1. **Multimodal LLMs**: While previous LLMs primarily focused on text, recent LLMs have also been able to operate on images and videos and even generate new images.

2. **LLMs for solving hard problems**: LLMs have already shown progress in solving math and coding problems, but they aren't perfect. There is still a lot of work that needs to be done to allow LLMs to solve difficult scientific problems and move society forward.

3. **LLMs for robotics**: LLMs have also been used to control robots and allow them to use world knowledge and common sense to perform tasks at a higher level and plan ahead as well.

Have fun exploring the wide world of LLMs!

Chapter 4

Additional Resources

4.1 Reinforcement Learning

For additional reading on reinforcement learning, I would recommend the following resources:

1. **Reinforcement Learning: An Introduction** by Richard S. Sutton and Andrew G. Barto. This is one of the most famous textbooks covering the mathematical foundations of reinforcement learning algorithms.

2. **Proximal Policy Optimization Algorithms**: This paper describes the PPO algorithm, which is still widely used today for training LLMs. https://arxiv.org/abs/1707.06347

3. **Soft Actor-Critic**: This paper describes the Soft Actor-Critic (SAC) algorithm, which is popular for continuous environments. https://arxiv.org/abs/1801.01290

4. **OpenAI Spinning Up**: This website provides a great overview of various RL algorithms along with simple, easy-to-understand implementations. https://spinningup.openai.com/en/latest/

5. **Gymnasium**: This package provides a series of standard RL environments that you can train agents for and test out RL algorithms on. https://gymnasium.farama.org/index.html

4.2 Diffusion Models

For additional reading on diffusion models, I would recommend the following resources:

1. **Denoising Diffusion Probabilistic Models**: This is one of the first papers to describe diffusion model algorithms. https://arxiv.org/abs/2006.11239

2. **Consistency Models**: This is a new variant of generative models that seeks to reduce the number of sampling steps in diffusion models, which is still a major cost of diffusion models. https://arxiv.org/abs/2303.01469

3. **RFDiffusion**: This method uses diffusion to design novel proteins, and was part of the work that was awarded the 2024 Nobel Prize in Chemistry. https://www.nature.com/articles/s41586-023-06415-8

4. **Understanding Diffusion Models: A Unified Perspective**: For the more mathy people, this paper offers a description of the math behind diffusion models and how they work. https://arxiv.org/abs/2208.11970

4.3 Large Language Models

For additional reading on large language models, I would recommend the following resources:

1. **Attention Is All You Need**: This is the original paper that introduced the transformer model architecture, used in most modern LLMs. https://arxiv.org/abs/1706.03762

2. **Language Models are Few-Shot Learners**: This is the paper from OpenAI that described GPT-3, one of the most well-known LLMs. https://arxiv.org/abs/2005.14165

3. **HuggingFace Transformers**: This is a popular framework that allows you to easily use and fine-tune open source LLMs on HuggingFace. https://github.com/huggingface/transformers

4. **vLLM**: This is a framework for fast inference of LLMs, and also allows you to inference LLMs on your own GPUs. `https://github.com/vllm-project/vllm`

Bibliography

[1] “Pac-Man Gameplay” (https://commons.wikimedia.org/wiki/File:Pac-Man_gameplay.png) by Bandai Namco (https://www.youtube.com/@BandaiNamcoAmerica) is licensed under the Creative Commons Attribution 4.0 International license (https://creativecommons.org/licenses/by/4.0/). 11