**5. Finetuning: The Theory**

Aarushi Kansal[**1**](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_5_Chapter.xhtml#Aff2)

(1)

Melbourne, Australia

In Chapter [4](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_4_Chapter.xhtml), you learned about making your LLM-powered application safer and more controllable. In particular, you focused on using NeMo to build guardrails around ensuring your LLM stays on topic, executes the right flow, and is able to block users. You looked into NeMo and understood how it combines LLMs, Colang, and embedding models to create a generalized set of rules, based on natural language rules you give it.

The last few chapters all involve using a foundational model as the “brain” of your application, in a plug-and-play kind of approach. You used RAG to augment your LLMs’ knowledge, avoid hallucination, and provide potentially private information to it.

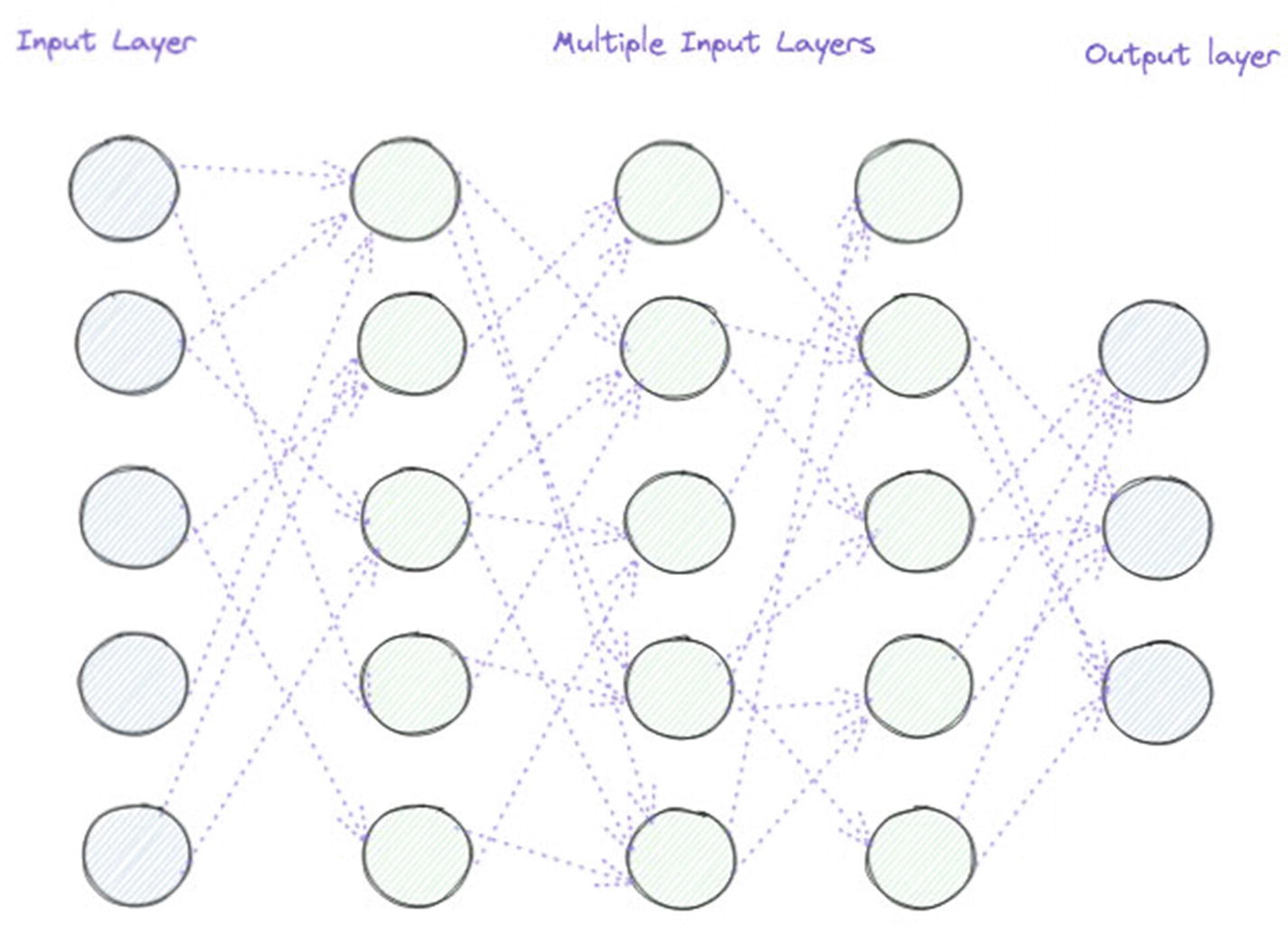
This chapter takes you through *fine-tuning*, which means taking a foundational model and updating one or more (generally not all) of its parameters, to make it suitable for a new task, to what it was originally trained for.

**Let’s Talk Foundational Models**

By now, you know about some of the different architectures (from Chapter [1](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_1_Chapter.xhtml)) that these foundational models are built on. Whether it’s an open source model (e.g., Llama 2) or a proprietary model (e.g., GPT-4), these models are trained on a *huge* amount of data. Some of this data is open source, some scrapped from the Internet, and some proprietary.

Regardless of the dataset, the point is that it’s a lot of data and models trained from scratch take up a lot of time, effort, and most importantly computing resources. On top of that, the way that the generative AI space is progressing, with the advent of foundational models, it’s going to become less and less likely that you’re going to have to train a model from scratch. More likely you’ll need to take an existing model and customize it to your own needs.

First, let’s look at a generalization of a model in Figure [5-1](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_5_Chapter.xhtml#Fig1).



***Figure 5-1***

General neural network showing layers and nodes

In Figure [5-1](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_5_Chapter.xhtml#Fig1), you can see an overall network is made up of multiple layers and nodes, starting with an input layer, feeding into the next layer, and so on, until you get a final output. The input and output depend on what the model is trained to do such as text generation, image generation, text summarization, and so on.

To build something like this from scratch, you would do a few things:

* **Data Collection**: The basis and often break or make of any model is the data it learns from. For language models, this could include a wide, diverse, range of text – from books to social media posts to online articles. For visual models, the dataset might consist of images or videos. The key is to collect a large and varied dataset that’s reflective of the tasks the model is expected to perform.
* **Data Cleaning/Labeling**: Quality data is the lifeblood of an effective model. This stage involves removing irrelevant, redundant, or erroneous information. For supervised learning tasks, it also includes labeling the data accurately, which can be a labor-intensive process needing a discerning human eye or sophisticated automation tools.
* **Designing the Model Architecture**: The architecture dictates how the data flows through the network. This involves selecting the type of neural network (e.g., convolutional, recurrent, transformer) and configuring the number of layers and nodes. The design is influenced by the nature of the task and the complexity of the data.
* **Training the Model**: Training involves using the prepared dataset to incrementally adjust the weights of the connections between nodes across layers. This process minimizes the difference between the model’s predictions and the actual data. It typically requires substantial computational resources and time, especially for large models.
* **Evaluation**: Post-training, the model is evaluated using a separate dataset not seen during training to assess its performance. Metrics such as accuracy, precision, recall, and F1 score for classification tasks, or BLEU score for translation tasks, help determine the model’s effectiveness.
* **Hyperparameter Tuning**: Hyperparameters are the settings that govern the training process. They can include learning rate, batch size, number of epochs, and layer configurations. Adjusting these parameters is crucial for optimizing model performance. Techniques like grid search, random search, or Bayesian optimization are employed to find the best combination.

So as you can see, there are quite a few steps, and building a model from scratch can be both resource and time consuming.

Given this, fine-tuning can be an attractive alternative; let’s talk a bit more about whys fine-tuning in the next section.

**The Whys of Fine-Tuning?**

Fine-tuning an existing large language model instead of building one from scratch can often be the more practical and efficient approach for several reasons:

1. 1.

**Resource Efficiency**: Training large models requires significant computational power and time. Fine-tuning leverages pre-trained models that have already undergone this intensive process, meaning you can achieve high performance without the same level of resource investment.

1. 2.

**Data Efficiency**: Large language models are typically pre-trained on vast, diverse datasets that individual organizations may not have access to. Fine-tuning allows you to benefit from this extensive pre-training, needing only a smaller, task-specific dataset to adapt the model to your particular application.

1. 3.

**Transfer Learning**: Pre-trained models have developed a general understanding of language, context, and even some domain knowledge. Fine-tuning transfers this learning to a specific task, which is much quicker than teaching a model from scratch.

1. 4.

**High Performance**: Pre-trained models have often been optimized and tested extensively by experts in the industry and open source community. Fine-tuning these models allows you to stand on the shoulders of giants, benefiting from state-of-the-art architectures that you might not have the resources to develop independently.

1. 5.

**Lower Barrier to Entry**: For organizations and individuals without access to enough of the necessary infrastructure, fine-tuning is a more accessible entry point into using advanced AI technologies.

1. 6.

**Continual Learning**: Pre-trained models can be updated continuously with new data or fine-tuned repeatedly for different tasks, making them highly versatile and adaptable to evolving needs and data.

1. 7.

**Broad Applicability**: A single pre-trained model can be fine-tuned for multiple domains and tasks, from translation and summarization to question-answering and sentiment analysis, making it a multipurpose tool that’s adaptable to various applications.

Essentially, fine-tuning can be a great way to take a model that already does one thing really well (e.g., generating language) and adapting it to another similar task (e.g., generating language specifically related to your product, or domain) – with less GPU, less time, and, more often than not, a lot less data.

Now that you know the *why*s of fine-tuning, let’s discuss what fine-tuning actually is.

**The Whats of Fine-Tuning**

Fine-tuning a pre-trained model involves several technical steps that tweak the model’s internal parameters to adapt it to a specific task. Here’s a closer technical look at what’s happening during the fine-tuning process:

**Starting Point: The Pre-trained Model**

* **Loaded Parameters**: The pre-trained model comes with a set of learned parameters (weights and biases) that encode knowledge from the pre-training dataset, typically a large corpus covering a wide range of topics.

**Preparation for Fine-Tuning**

* **Task-Specific Dataset**: You start with a dataset that is closely related to the task you want the model to perform. This dataset usually needs to be labeled, unless you’re performing unsupervised fine-tuning.
* **Feature Extraction**: The model processes the task-specific data, using its pre-trained layers to extract features. These features are complex patterns that the model has learned to recognize.

**Fine-Tuning Process**

* **Parameter Adjustment**: Fine-tuning involves backpropagation and gradient descent, just like initial training. But the updates to the parameters are smaller and more refined. This is because you’re not learning from scratch; you’re tweaking existing knowledge.
* **Learning Rate**: A critical aspect is using a smaller learning rate. This prevents the pre-trained parameters from changing too rapidly, which could cause the model to “forget” what it has learned (commonly referred to as catastrophic forgetting).
* **Epochs**: The number of epochs (complete passes through the training dataset) during fine-tuning is typically much less than during pre-training since you’re building on top of the pre-trained knowledge.

**During Training**

* **Loss Function**: The loss function measures how well the model is performing on the new task. During fine-tuning, you continue to minimize this loss. The gradients calculated from this loss are used to update the model’s weights.
* **Gradient Updates**: In fine-tuning, gradients are often smaller, and updates are more nuanced. Depending on the fine-tuning strategy, some layers of the model may have their weights frozen, and only the final layers are updated, or all layers may be fine-tuned together.
* **Regularization**: Techniques such as weight decay or dropout may be used during fine-tuning to prevent overfitting, especially since fine-tuning datasets can be smaller.

**Fine-Tuning Strategies**

* **Full Model Fine-Tuning**: All the weights in the model are updated during fine-tuning. This is often used when the fine-tuning dataset is large and diverse enough to warrant comprehensive retraining.
* **Partial Fine-Tuning**: Only the weights of the last few layers are updated. In neural networks, this often means adjusting the weights of the layers closer to the output (the “head” of the model) while keeping the earlier layers (the “body” or “base” of the model) frozen. This approach is common when the new task is quite similar to the pre-training task, or when the dataset is smaller.

**After Fine-Tuning**

* **Evaluation**: The fine-tuned model is tested against a validation dataset to measure its performance. Depending on the outcome, more rounds of fine-tuning might be necessary.
* **Hyperparameter Optimization**: Based on performance, you may need to adjust hyperparameters. This can involve methods like grid search, random search, or Bayesian optimization to find the best settings.

In the technical sense, fine-tuning is a delicate optimization process. You’re nudging the pre-trained model – shaped by vast amounts of data and training – toward a specific task or domain with the least amount of force needed to make it perform well on that new task.

**Network Level Changes**

When a neural network is fine-tuned, there are several changes that occur at the level of the network’s architecture and the individual neurons:

1. 1.

**Weight Adjustments**

* + The fundamental change during fine-tuning is the adjustment of the weights within the neural network. Weights are the parameters that determine the importance of input features and how they contribute to the output.
  + Each neuron in the network has an associated weight for its inputs, and these weights are incrementally adjusted during the training process.
  + In fine-tuning, these adjustments are based on the errors the model makes on the new task-specific data.

1. 2.

**Backpropagation and Gradient Descent**

* + Fine-tuning uses backpropagation to calculate gradients or changes needed to reduce error. These gradients indicate how the weights should be altered to minimize the loss function.
  + Gradient descent is then applied to iteratively adjust the weights in the direction that decreases the loss.

1. 3.

**Learning Rate**

* + A crucial aspect of fine-tuning is the use of a lower learning rate than in pre-training. This ensures that the model does not undergo drastic changes that could undo the general knowledge it has already acquired.

1. 4.

**Activation Function Outputs**

* + The outputs of the neurons’ activation functions are also modified as the weights change. Since each neuron’s output is a function of its weighted inputs, adjusting the weights alters the signal that each neuron outputs.
  + This is significant because it essentially means the representation of the data within the model changes, ideally becoming more aligned with features relevant to the new task.

1. 5.

**Layer-Specific Changes**

* + Depending on the approach, fine-tuning may involve changing only the upper layers (closer to the output) or all layers of the model.
  + The layers closer to the input (lower layers) tend to capture more general features, while the layers closer to the output (upper layers) capture more abstract, task-specific features. Therefore, fine-tuning often focuses on these upper layers.

1. 6.

**Freezing Layers**

* + In some fine-tuning practices (such as partial fine-tuning, mentioned previously), earlier layers are “frozen,” meaning their weights are kept constant, and only the weights of the higher layers are allowed to change.
  + This is done under the assumption that the lower layers capture universal features that are useful across different tasks, whereas the higher layers need to be more specialized.

1. 7.

**Regularization**

* + Techniques such as dropout may be implemented or adjusted during fine-tuning. For example, dropout randomly ignores a subset of neurons during each training pass, which helps to prevent overfitting by forcing the network to spread out learning over more neurons.

1. 8.

**Feature Space Adjustment**

* + As weights are updated, the way the network represents information (the feature space) changes. Fine-tuning aims to shift this feature space toward one that is more useful for the new task without losing the beneficial properties learned during pre-training.

1. 9.

**Final Layer Adaptation**

* + Often, the final layer of the network, which makes the final predictions or classifications, is completely replaced to fit the new task. For instance, if the pre-trained model was designed for 1,000 classes and the new task only has 10, the final layer would be adjusted accordingly.

1. 10.

**Batch Normalization Parameters**

* + If the network uses batch normalization, the parameters for this – such as mean and variance used to normalize each batch of data – can be updated during fine-tuning to better suit the new data distribution.

These changes happen iteratively over each pass of the dataset (epoch), and after sufficient epochs, the model’s performance on the new task should ideally improve. Fine-tuning allows the network to maintain its pre-trained “intuition” while reshaping its inner workings to address the specifics of the new task more effectively.

At this point, you’ve seen a brief look into the general world and concept of fine-tuning. One of the most interesting developments in this new world of generative AI is also the various ways of fine-tuning. All of these being aimed at finding the most optimal way to take a foundational, pre-trained model like Llama 2 and adapting it to specific domains like coding, law, psychology, and so on. In the next section, you’re going to learn about a few of these ways of fine-tuning. This chapter is primarily theory – so in the next chapter, you have a solid foundation for implementation.

**The Hows of Fine-Tuning**

Okay, so now that you know the whys and whats of fine-tuning, I want to take you through a few fine-tuning techniques:

* Reinforcement Learning with Human Feedback (RLHF)
* Parameter-Efficient Fine-Tuning (PEFT)
* Low-Rank Adaptation (LoRA)

Each of these is fairly recent and popular techniques. Let’s start with RLHF.

**Reinforcement Learning with Human Feedback (RLHF)**

Before we dive into RLHF, if you aren’t already familiar with reinforcement learning, I recommend you have a quick look and read up on it from a theoretical level.

Okay, so RLHF isn’t a single concept; it’s actually made up of three components:

1. 1)

Fine-tuning a pre-trained LLM with supervised learning

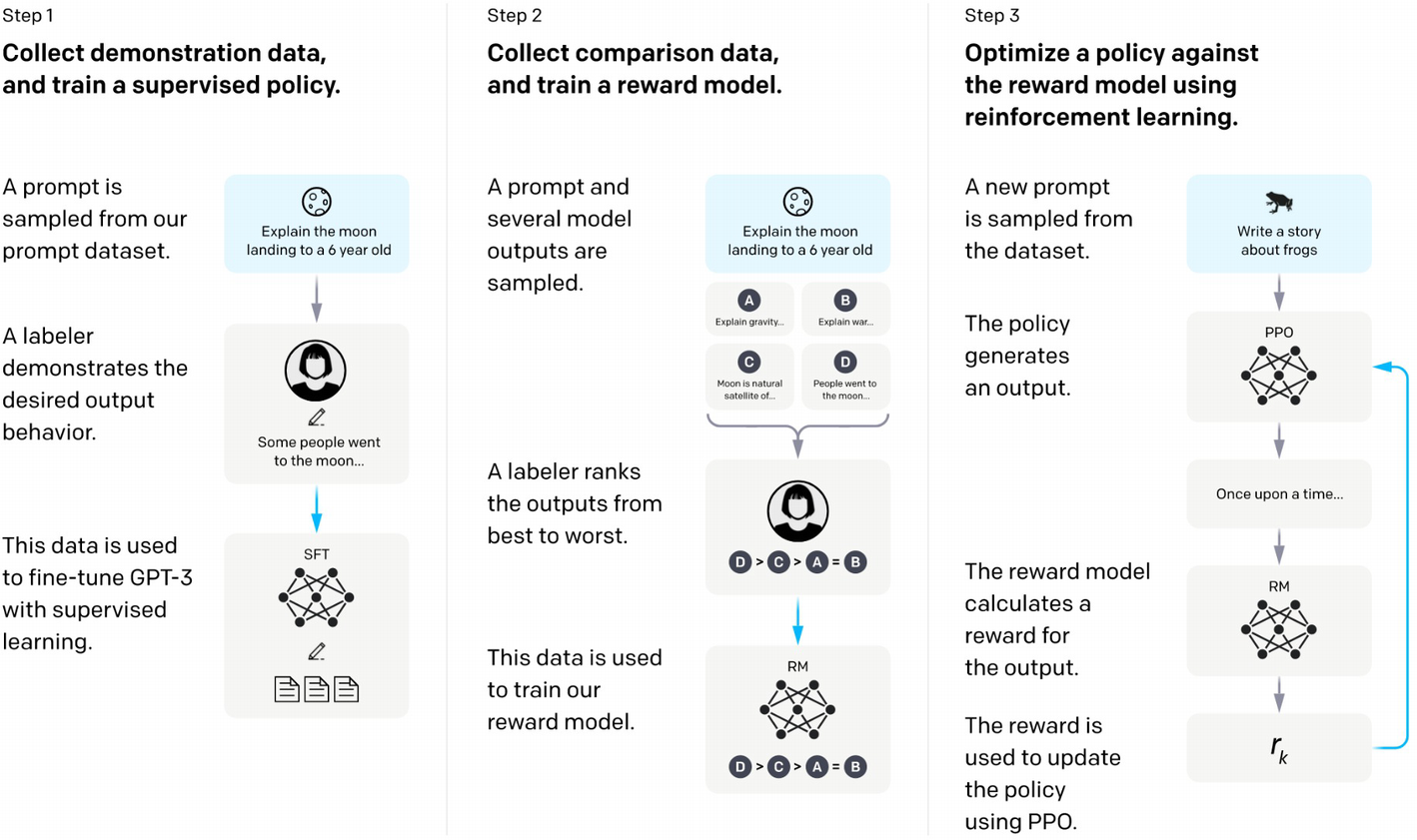
1. 2)

Data collection to train a new reward model

1. 3)

Fine-tuning the LLM with reinforcement learning

These components are shown in Figure [5-2](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_5_Chapter.xhtml#Fig2).



***Figure 5-2***

The three-step process in RLHF (source: [https://​openai.​com/​research/​instruction-following](https://openai.com/research/instruction-following))

Let’s dive into RLHF deeper now, starting with step 1, supervised fine-tuning.

**Supervised Fine-Tuning (SFT)**

Before reinforcement learning even begins, a model like GPT-3 is fine-tuned on a curated dataset of human interactions. This dataset consists of pairs of prompts and human-generated responses. The model learns from this dataset to predict the responses that a human would give. This initial step aims to steer the model toward generating outputs that are already in line with what humans consider appropriate or useful.

**Reward Modeling (RM)**

The reward model is the cornerstone of RLHF. Constructed using a dataset where human raters have provided feedback on the quality of model outputs, the reward model is trained to predict the “reward” or value of an output. In other words, it estimates how well an output aligns with human preferences. The feedback can come in different forms:

* **Direct Rating**: Raters assign scores to outputs based on criteria such as coherence, truthfulness, and helpfulness.
* **Relative Preference**: Raters compare pairs of outputs and judge which one is better, without assigning explicit scores.

The reward model essentially internalizes these human judgments and becomes a proxy for human evaluation, allowing the reinforcement learning process to occur at a larger scale without constant human intervention.

**Reinforcement Learning Algorithms**

Once the reward model is in place, the actual reinforcement learning takes place. A typical choice of algorithm is Proximal Policy Optimization (PPO), an on-policy algorithm known for its stability and reliability. The large language model is treated as the agent in reinforcement learning terminology, and it seeks to maximize the cumulative reward it receives over sequences of interactions.

During training:

* **Exploration**: The model tries out different ways of responding to inputs to discover strategies that lead to higher rewards.
* **Exploitation**: The model uses what it has learned to produce the outputs that it predicts will yield the highest reward.

This process is inherently more complex than standard supervised learning because the model is not just learning to replicate a fixed set of responses. It is actively trying to improve the quality of its outputs based on the moving target of the reward model’s predictions.

**Human Preference Comparison**

To refine the reward model and ensure it aligns with human preferences, an additional step often used is preference modeling. Here, raters are presented with pairs of model-generated outputs and asked to choose which one is preferable. These pairwise comparisons can sometimes be more intuitive and reliable than numerical scoring systems.

**Iterative Training**

The RLHF process is usually iterative:

1. 1.

The reward model is initially trained on a dataset of human judgments.

1. 2.

The policy model (the language model) is trained to maximize the reward using the current reward model.

1. 3.

The policy model’s outputs are then rated by humans to create a new dataset.

1. 4.

This new dataset is used to update the reward model, making it more accurate.

1. 5.

The policy model is fine-tuned again using the updated reward model.

Each iteration aims to refine the model’s understanding of human preferences, leading to better alignment with human values.

**AI Alignment and Safety**

RLHF is not just a training method – it’s an approach to AI safety and alignment. The goal is to develop systems that don’t just perform well on narrow tasks but also act in ways that are ethically and socially acceptable. For instance, if a model is generating content for children, RLHF could be used to align the model’s outputs with educational and ethical standards suitable for young audiences.

**Challenges and Considerations**

* **Scalability**: Even though the reward model makes the process more scalable, it still relies on a substantial amount of high-quality human feedback.
* **Bias and Fairness**: The feedback data can embed human biases, and the reward model might perpetuate or amplify these biases.
* **Complexity and Safety**: Crafting a reward function that captures all aspects of human values is incredibly complex. Moreover, reinforcement learning can lead to unexpected policy improvements that exploit loopholes in the reward function.

Overall, RLHF is about teaching AI systems to understand and replicate complex human judgments and preferences. It’s a dynamic and iterative process that combines the power of large-scale machine learning with the nuance of human evaluation. As models grow in capability, methods like RLHF are crucial for ensuring they act in ways that are beneficial – and acceptable – to humans.

While RLHF is powerful and has its benefits, one of the main challenges remains: scalability. Luckily there are other ways to overcome this challenge and still fine-tune and adapt your models in a high-quality way.

Parameter-Efficient Fine-Tuning (PEFT) and Low-Rank Adaptation (LoRA) are two methods used to fine-tune large language models while addressing the challenges of scalability and resource constraints.

**PEFT**

PEFT techniques aim to overcome several challenges:

1. 1.

**Avoid Catastrophic Forgetting**: When fine-tuning a model on a new task, there’s a risk of overwriting previously learned information. PEFT methods like Adapter layers ensure that the original pre-trained weights remain unchanged, thus preserving the model’s general knowledge while still learning task-specific nuances.

1. 2.

**Reduce Compute and Storage Costs**: Fine-tuning all the parameters of large models is compute-intensive and requires substantial storage for each version of the model. PEFT approaches require updating fewer parameters, thus reducing these costs significantly.

1. 3.

**Enable Task-Specific** **Adaptations**: For applications requiring models to perform well on a wide array of specialized tasks, PEFT methods allow for each task to have its own set of fine-tuned parameters without the need to re-train the entire model.

**Example**: Suppose we are adapting a language model for both medical diagnosis and financial forecasting. Using Adapter layers, we could insert small modules specifically tuned for each domain, while the core model remains unchanged. This allows the model to provide accurate medical diagnoses or financial insights without the risk of the medical information interfering with financial predictions, or vice versa.

**How Does PEFT Work**

PEFT approaches are designed to fine-tune pre-trained models by updating only a small subset of parameters. This allows the model to maintain most of its pre-trained knowledge while adapting to new tasks or domains efficiently. Let’s break down some of the common techniques:

* **Adapter Layers**: These are small trainable modules inserted between the layers of a pre-trained model. Each adapter consists of a down-projection that reduces dimensionality, a nonlinearity (like ReLU), and an up-projection that restores the original dimension. During fine-tuning, the main model weights remain frozen, and only the adapter parameters are updated. This technique allows for task-specific learning without large-scale weight modifications.
* **Prompt Tuning**: Instead of adding new parameters, prompt tuning introduces a set of learnable embeddings called “prompts” that are prepended to the input sequence. These prompts are designed to guide the model to activate relevant pathways within its existing weights for the target task. During fine-tuning, only these prompt embeddings are updated, acting as a form of “soft prompts” that modify the input space to elicit the desired output.
* **BitFit**: An even more parameter-efficient approach where only the bias terms in the model’s layers are fine-tuned. The idea is that bias terms have a significant impact on the decision boundaries of models and can be tweaked to adjust for new tasks while keeping all other weights fixed.

**Low-Rank Adaptation (LoRA)**

LoRA specifically addresses the balance between maintaining a model’s pre-trained performance and allowing significant flexibility for new tasks:

1. 1.

**Fine-Grained Control over Changes**: LoRA’s low-rank updates allow fine-grained control over the changes to the model’s behavior. The rank r acts as a knob, balancing between adaptability and parameter efficiency.

1. 2.

**Maintaining** **Computational Efficiency**: Despite updating the model, LoRA’s additive updates are efficient to compute, as they do not require a complete re-parameterization of the model.

1. 3.

**Widespread Impact** **with Minimal Changes**: Because the low-rank updates affect the model’s weight matrices, which are central to its predictions, even small changes can have a widespread impact on the model’s outputs, enabling significant task-specific adaptations.

**Example**: Imagine a language model trained on general web text being adapted to write poetry. Using LoRA, we can introduce low-rank updates to the self-attention mechanism, which would help the model understand the structure and style of poetry. The low-rank matrices AA and BB could be trained on a small dataset of poems, fine-tuning the model’s ability to generate poetic language and structure without needing to re-train the whole model on poetic text.

**Challenges Addressed**

* **Model Generality vs. Specificity**: PEFT and LoRA enable a balance between retaining the model’s broad capabilities and adapting to niche requirements.
* **Overfitting**: By updating fewer parameters, there’s a reduced risk of overfitting to the fine-tuning dataset, which can be a significant problem when completely re-training large models.
* **Resource Constraints**: These methods are especially relevant in scenarios with limited resources, where training or fine-tuning entire models isn’t feasible.
* **Model Personalization**: For applications that require personalized models (e.g., personalized AI assistants), PEFT allows creating numerous specialized models without duplicating the entire set of parameters for each user.

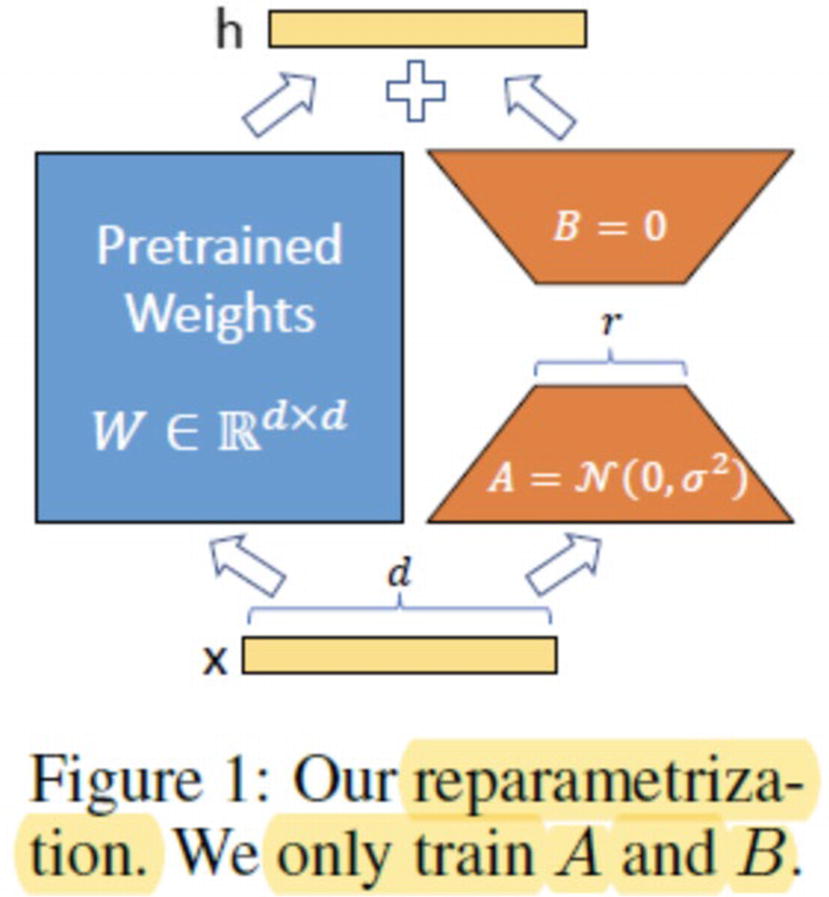
At its core, LoRA targets the weight matrices within the Transformer layers, which are key components in the model’s architecture. Transformers consist of multi-head self-attention mechanisms and feed-forward neural networks. LoRA specifically targets the self-attention mechanism’s query (Q), key (K), and value (V) matrices, as well as the feed-forward network’s weight matrices.

In a standard Transformer, the output of the self-attention for each head is computed as

* Attention(Q,K,V)=softmax(QK^T/sqrt(dk))V

Here, dk​ is the dimensionality of the keys.

In LoRA, instead of directly learning and updating the large weight matrices (WQ,WK,WV) of the self-attention or the feed-forward networks, the approach introduces low-rank matrices A and B for each original weight matrix that we wish to adapt – it’s these two matrices that are fine-tuned (as shown in Figure [5-3](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_5_Chapter.xhtml#Fig3)). The original weight matrix W is not changed; instead, LoRA adds a low-rank matrix product AB^T to W:



***Figure 5-3***

Diagram from a LoRA paper, only A and B are fine-tuned (source: [https://​arxiv.​org/​abs/​2106.​09685](https://arxiv.org/abs/2106.09685))

W′=W+AB^T

**Decomposing LoRA’s Mechanism**

1. 1.

**Low-Rank Matrix Factorization**

* + A and B are much smaller matrices compared to W, with dimensions d×r and r×m, where d is the original input dimension, m is the output dimension, and r is the rank.
  + The rank r is chosen based on the desired balance between adaptability and efficiency. A lower rank means fewer parameters to train but potentially less capacity for adaptation.

1. 2.

**Adaptation Without Complete Re-training**

* + During fine-tuning, only the A and B matrices are learned, while W remains frozen.
  + This is particularly advantageous for large models where updating all parameters is computationally prohibitive.

1. 3.

**Efficient Forward and Backward Pass**

* + During the forward pass, LoRA computes AB^T on the fly and adds it to W to form the adapted matrix W′.
  + In the backward pass, gradients are computed only with respect to A and B, leaving the pre-trained weights WW unchanged.

LoRA can be particularly effective in transformer models because it allows the modification of self-attention and feed-forward networks with a limited number of additional parameters. The low-rank structure leverages the redundancy present in the parameterization of these models, offering a balance between adaptability and parameter efficiency.

In essence, both PEFT and LoRA methods provide mechanisms to retain the extensive knowledge captured during pre-training while enabling the model to specialize and perform well on specific tasks, even with limited amounts of task-specific data and computational resources.

**Summary**

In this chapter, you focused on learning about fine-tuning on a theoretical level, starting with gaining an understanding of how foundational models are built from scratch and the potential challenges. From there, you learned about general fine-tuning and how it may be less resource and time consuming than building and training a new model. Next you learned about two main techniques: RLHF and LoRA. This chapter was a theoretical introduction to fine-tuning, to help build the foundations for the next chapter, where you will fine-tune a model yourself.