Methods for tuning foundation models

Learn about the different tuning methods that are available in IBM watsonx.ai so that you can choose the method that's right for your generative AI solution.

You can tune foundation models in watsonx.ai by using one of the following methods:

- **Full fine tuning**: Using the base model's previous knowledge as a starting point, full fine tuning tailors the model by tuning it with a smaller, task-specific dataset. The full fine-tuning method changes the parameter weights for a model whose weights were set through prior training to customize the model for a task.

  The result of full fine tuning is an entirely new model. Because all of the model weights are tuned, full fine tuning is a more expensive technique than parameter-efficient tuning techniques. More compute and storage resources are required to tune and host the new tuned model that you create by fine tuning a foundation model. See Full fine tuning.

- **Low-rank adaptation (LoRA) fine tuning**: Adapts a foundation model for a task by changing the weights of a representative subset of the model parameters, called *low-rank adapters*, instead of the base model weights during tuning. At inference time, weights from the tuned adapters are added to the weights from the base foundation model to generate output that is tuned for a task. See Low-rank adaptation (LoRA) fine tuning.

- **Quantized low-rank adaptation (QLoRA) fine tuning**: QLoRA is a variant of LoRA that incorporates quantization to further reduce the memory footprint and computational resources that are required during tuning. See Quantized low-rank adaptation (QLoRA) fine tuning.

- **Prompt tuning**: Adjusts the content of the prompt that is passed to the model to guide the model to generate output that matches a pattern you specify. The underlying foundation model is not changed. Only the prompt input is altered.

  Although the result of prompt tuning is a new tuned model asset, the asset merely adds a layer of function that runs before the input is processed by the underlying foundation model. Because the foundation model itself is not changed, the same model can be used to address different business needs without being retrained each time. See Prompt tuning.

Tuning methods comparison

The following table compares the tuning methods that are available in watsonx.ai based on common criteria for choosing a tuning method.

Criteria	**Prompt tuning**	**Full fine tuning**	**LoRA fine tuning**	**QLoRA fine tuning**
Tuning technique	Prompt vectors are tuned; base model parameters remain fixed.	All base model parameters are fine-tuned on the target task.	Adapters that represent a subset of model parameters are tuned; base model parameters remain fixed during tuning.	Adapters that represent a subset of model parameters are tuned; base model parameters remain fixed during tuning.
Tuned model outcomes	Effective when the target task is similar to the pretrained knowledge of the model.	Effective at customizing a model for a new task or domain when given sufficient data and compute resources.	High performance with reduced risk of overfitting; might not reach level of full fine tuning performance.	High performance with reduced risk of overfitting; might not reach level of full fine tuning performance. Potential quality degradation introduced by quantization.
Required compute resources	Low. Minimum resources are required. Only the prompt vector is tuned; the underlying model is unaltered.	High. Large computational resources and memory are required to fully update the model parameters.	Moderate. Requires fewer resources than full fine tuning because only the adapters are tuned; the underlying model is unaltered during tuning.	Low. Requires fewer resources than LoRA fine tuning because the model weights are quantized to reduce computational and storage needs.

Tuning time duration	Short. Quickest process because only the prompts are changed; can range from 10 minutes to a few hours.	Long. Exact duration depends on the model and dataset sizes.	Moderate. Faster than full fine tuning, but takes time to modify the adapters; can range from one to many hours.	Moderate. Faster than full fine tuning, but takes time to modify the adapters; can range from one to many hours.

Full fine tuning

Full fine tuning uses the base model's previous knowledge as a starting point to tailor the model by tuning it with a smaller, task-specific dataset. The full fine-tuning method changes the parameter weights for a model whose weights were set through prior training to customize the model for a task.

How full fine tuning works

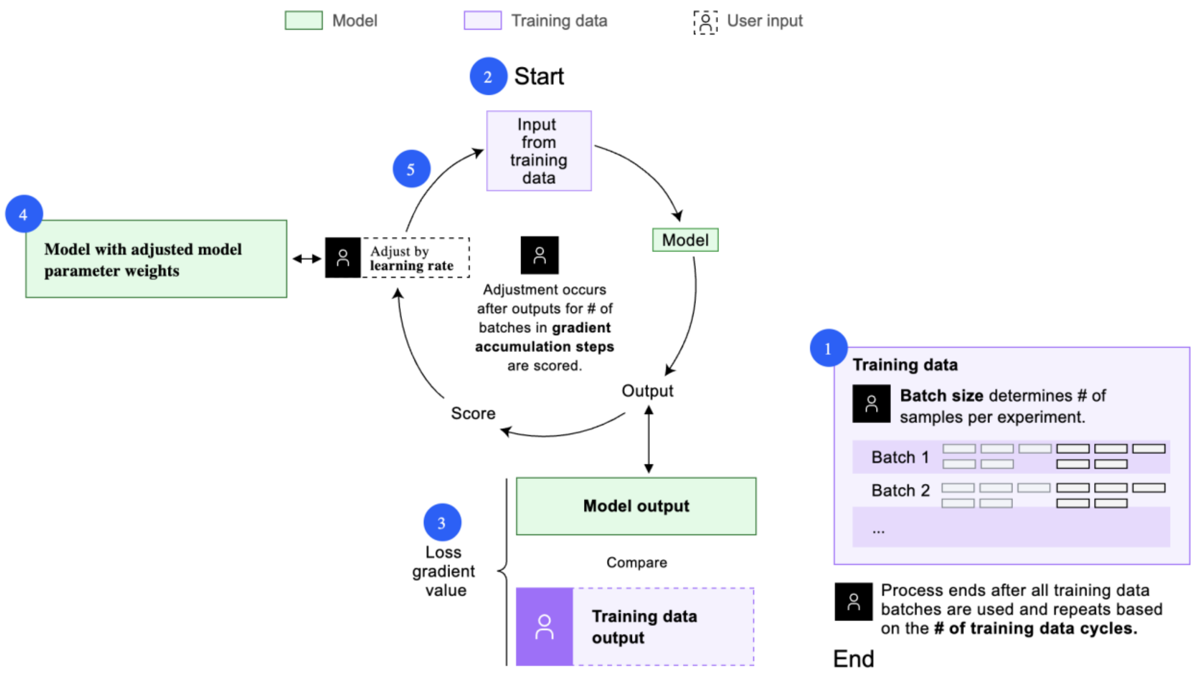
Use the Tuning Studio in watsonx.ai to run a fine-tuning experiment that uses supervised learning to train a foundation model on a specific task. You provide training data that consists of example pairs of user input and corresponding foundation model output that is appropriate for your task.

The Tuning Studio runs a fine-tuning experiment that manages a series of training runs in which the output that is generated by the foundation model is compared to example output from your training data. Based on the differences between the two responses, the experiment adjusts the underlying foundation model parameter weight values. After many runs through the training data, the model finds the parameter weights that generate output that more closely matches the output you want. The result of the fine-tuning experiment is a new foundation model that is tuned for your task.

Full fine-tuning workflow

During a full fine-tuning experiment, the parameter weights of the tuning model are repeatedly adjusted so that its predictions get better over time.

The following diagram illustrates the steps that occur during a fine-tuning experiment run. The parts of the experiment flow that you can configure are highlighted by a **User input** icon. These decision points correspond with experiment tuning parameters that you control.



Model
Training data
User input
2 Start
Input from training data
Model
Output
Score
5
4 Model with adjusted model parameter weights
Adjust by learning rate
Adjustment occurs after outputs for # of batches in gradient accumulation steps are scored.
3 Loss gradient value
Model output
Compare
Training data output
1 Training data
Batch size determines # of samples per experiment.
Batch 1
Batch 2
...
Process ends after all training data batches are used and repeats based on the # of training data cycles.
End

The diagram shows the following steps of the full fine-tuning experiment:

1. The experiment reads the training data, tokenizes it, and converts it into batches.

   The size of the batches is determined by the batch size parameter.

2. Sends input from the examples in the batch to the foundation model for the model to process and generate output.

3. Compares the model's output to the output from the training data that corresponds to the training data input that was submitted. Then, the experiment computes the loss gradient, which measures the difference between the predicted output and the actual output from the training data.

The experiment adjusts the foundation model parameter weights based on the computed loss of the model. When this adjustment occurs depends on how the accumulation steps parameter is configured.

4. Adjustments are applied to the parameter weights of the foundation model. The degree to which the weights are changes is controlled by the learning rate parameter.

5. Input from the next example in the training data is submitted to the foundation model as input.

6. The process repeats until all of the examples in all of the batches are processed.

7. The entire set of batches are processed again as many times as is specified in the number of epochs parameter.

Learn more about full fine tuning

- IBM.com: What is fine-tuning?

Low-rank adaptation fine tuning (LoRA)

Low-rank adaptation (LoRA) fine tuning adapts a foundation model for a task by changing the weights of a representative subset of the model parameters, called *low-rank adapters*, instead of the base model weights during tuning. At inference time, weights from the tuned adapters are added to the weights from the base foundation model to generate output that is tuned for a task.

How low-rank adaptation (LoRA) fine tuning works

Low-rank adaptation (LoRA) is a parameter-efficient fine-tuning (PEFT) technique that adds a subset of parameters to the frozen base foundation model and updates the subset during the tuning experiment, without modifying the parameters of the base model. When the tuned foundation model is inferenced, the new parameter weights from the subset are added to the parameter weights from the base model to generate output that is customized for a task.

How the subset of parameters is created involves some mathematics. Remember, the neural network of a foundation model is composed of layers, each with a complex matrix of parameters. These parameters have weight values that are set when the foundation model is initially trained. The subset of parameters that are used for LoRA tuning is derived by applying rank decomposition to the weights of the base foundation model.

The rank of a matrix indicates the number of vectors in the matrix that are linearly independent from one another. Rank decomposition, also known as *matrix decomposition*, is a mathematical method that uses this rank information to represent the original matrix in two smaller matrices that, when multiplied, form a matrix that is the same size as the original matrix. With this method, the two smaller matrices together capture key patterns and relationships from the larger matrix, but with fewer parameters. The smaller matrices produced are called *low-rank matrices* or *low-rank adapters*.

During a LoRA tuning experiment, the weight values of the parameters in the subset, in the low-rank adapters, are adjusted. Because the adapters have fewer parameters, the tuning experiment is faster and needs fewer resources to store and compute changes. Although the adapter matrices lack some of the information from the base model matrices, the LoRA tuning method is effective because LoRA exploits the fact that large foundation models typically use many more parameters than are necessary for a task.

The output of a LoRA fine-tuning experiment is a set of adapters that contain new weights. When these tuned adapters are multiplied, they form a matrix that is the same size as the matrix of the base model. At inference time, the new weights from the product of the adapters are added directly to the base model weights to generate the fine-tuned output.

You can configure parameters of the LoRA tuning experiment, such as the base foundation model layers to target and the rank to use when decomposing the base model matrices.

When you deploy the adapter asset, you must deploy the asset in a deployment space where the base model is also deployed. You can use the LoRA fine tuning method in watsonx.ai to fine tune only non-quantized foundation models.

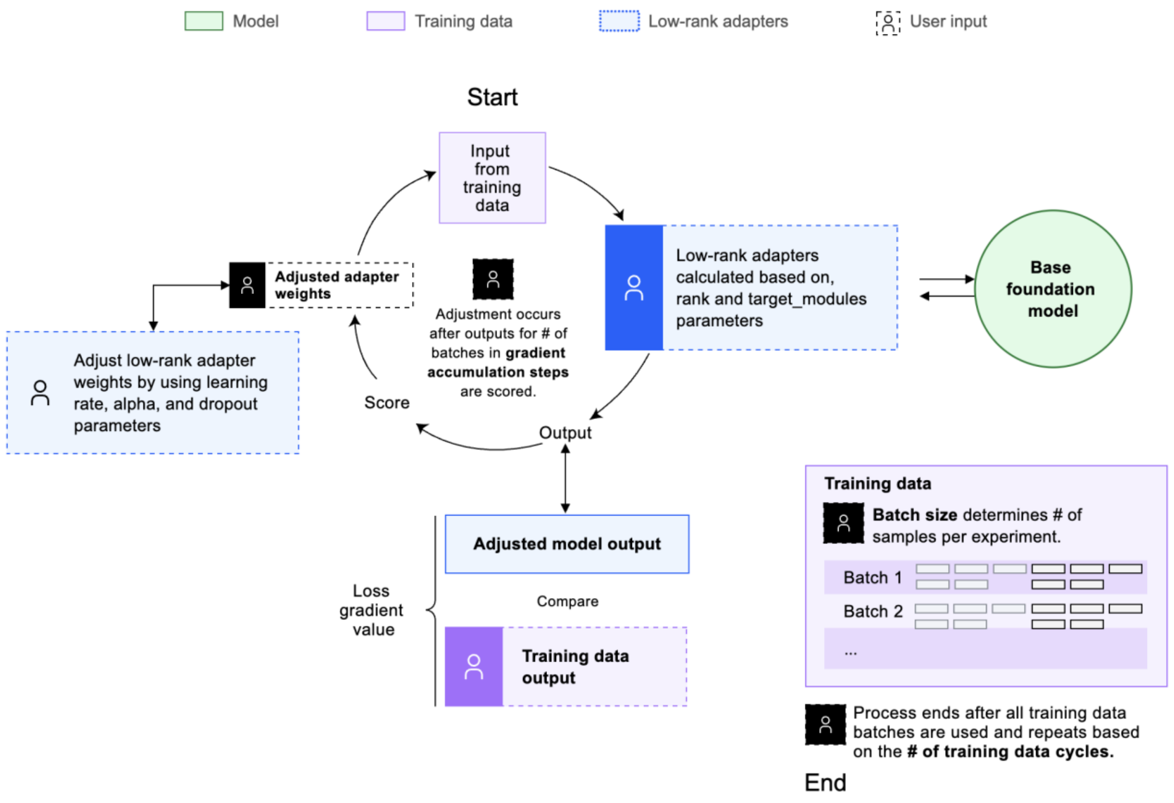
The benefits of using the LoRA fine-tuning technique include:

- The smaller, trainable adapters used by the LoRA technique require fewer storage and computational resources during tuning.
- Adjustments from the adapters are applied at inference time without impacting the context window length or the speed of model responses.
- You can deploy one base foundation model and use the model with different adapters to customize outputs for different tasks.

Low-rank adaptation fine-tuning workflow

During the LoRA fine-tuning experiment, the parameter weights of a representative subset of the model parameters, called low-rank adapters, are repeatedly adjusted so that the predictions of the tuned foundation model get better over time.

The following diagram illustrates the steps that occur during a LoRA fine-tuning experiment run. The parts of the experiment flow that you can configure are highlighted by a **User input** icon. These decision points correspond with experiment tuning parameters that you control.



Model
Training data
Low-rank adapters
User input
Start
Input from training data
Low-rank adapters calculated based on, rank and target_modules parameters
Base foundation model
Adjusted adapter weights
Adjustment occurs after outputs for # of batches in gradient accumulation steps are scored.
Adjust low-rank adapter weights by using learning rate, alpha, and dropout parameters
Score
Output
Adjusted model output
Compare
Loss gradient value
Training data output
Training data
Batch size determines # of samples per experiment.
Batch 1
Batch 2
...
Process ends after all training data batches are used and repeats based on the # of training data cycles.
End

The diagram shows the following steps of the experiment:

1. The experiment reads the training data, tokenizes it, and converts it into batches. The size of the batches is determined by the batch size parameter.

2. Low-rank adapters, which are a representative subset of the base model parameters, are devised. The initial weights of the low-rank adapters are applied to the model layers that you specify in the target_modules parameter and are calculated based on the value that you specify in the rank parameter.

3. Sends input from the examples in the batch to the LoRA adapters, and then to the foundation model to process and generate output.

4. Compares the model's output to the output from the training data that corresponds to the training data input that was submitted. Then, the experiment computes the loss gradient, which is the difference between the predicted output and the actual output from the training data.

   The experiment adjusts the LoRA adapter parameter weights based on the computed loss of the model. When this adjustment occurs depends on how the accumulation steps parameter is configured.

5. Adjustments are applied to the parameter weights of the LoRA adapters. The degree to which the weights are changed is controlled by a combination of the learning rate, alpha, and dropout parameter values.

6. Input from the next example in the training data is submitted to the LoRA adapter as input. The adapter applies the latest weight changes and adds them to the base foundation model weights to adjust them for the task.

7. The process repeats until all of the examples in all of the batches are processed.

8. The entire set of batches are processed again as many times as is specified in the number of epochs parameter.

Learn more about LoRA

- IBM.com: What is LoRA (low-rank adaption)?
- Research paper: Low-rank adaptation of large language models

Quantized low-rank adaptation fine tuning (QLoRA)

Quantized low-rank adaptation (QLoRA) fine tuning is a variant of LoRA that incorporates quantization to further reduce the memory footprint and computational resources that are required during tuning.

How quantized low-rank adaptation fine tuning works

The quantized low-rank adaptation (QLoRA) is a parameter-efficient fine-tuning (PEFT) method that enables you to apply the LoRA tuning method to quantized foundation models. If you don't know much about low-rank adaptation, first see Low-rank adaptation (LoRA) fine tuning.

Foundation model quantization is a method by which the resources required to store a foundation model are reduced by converting the data type of the model weights to a less precise data type. The weights are then returned to their original precision for inferencing.

Models that support quantization can also be quantized during the tuning process. For example, parameter weights of a foundation model that use a 16-bit float format are converted to a 4-bit integer format when the weights are stored and adjusted during a tuning experiment. The tuned model parameter weights are then returned to their original data type for inferencing.

You can use the QLoRA tuning method in watsonx.ai to fine tune only quantized foundation models.

Learn more about QLoRA

- Research paper: Low rank quantization adaptation for large language model

Prompt tuning

Prompt tuning adjusts the content of the prompt that is passed to the model to guide the model to generate output that matches a pattern you specify. The underlying foundation model and its parameter weights are not changed. Only the prompt input is altered.

How prompt tuning works

Foundation models are sensitive to the input that you give them. Your input, or how you prompt the model, can introduce context that the model will use to tailor its generated output. Prompt engineering to find the right prompt often works well. However, it can be time-consuming, error-prone, and its effectiveness can be restricted by the context window length that is allowed by the underlying model.

Prompt tuning a model in the Tuning Studio applies machine learning to the task of prompt engineering. Instead of adding words to the input itself, prompt tuning is a method for finding a sequence of values that, when added as a prefix to the input text, improve the model's ability to generate the output you want. This sequence of values is called a prompt vector.

Normally, words in the prompt are vectorized by the model. Vectorization is the process of converting text to tokens, and then to numbers defined by the model's tokenizer to identify the tokens. Lastly, the token IDs are encoded, meaning they are converted into a vector representation, which is the input format that is expected by the embedding layer of the model. Prompt tuning bypasses the model's text-vectorization process and instead crafts a prompt vector directly. This changeable prompt vector is concatenated to the vectorized input text and the two are passed as one input to the embedding layer of the model. Values from this crafted prompt vector affect the word embedding weights that are set by the model and influence the words that the model chooses to add to the output.

To find the best values for the prompt vector, you run a tuning experiment. You demonstrate the type of output that you want for a corresponding input by providing the model with input and output example pairs in training data. With each training run of the experiment, the generated output is compared to the training data output. Based on what it learns from differences between the two, the experiment adjusts the values in the prompt vector. After many runs through the training data, the model finds the prompt vector that works best.

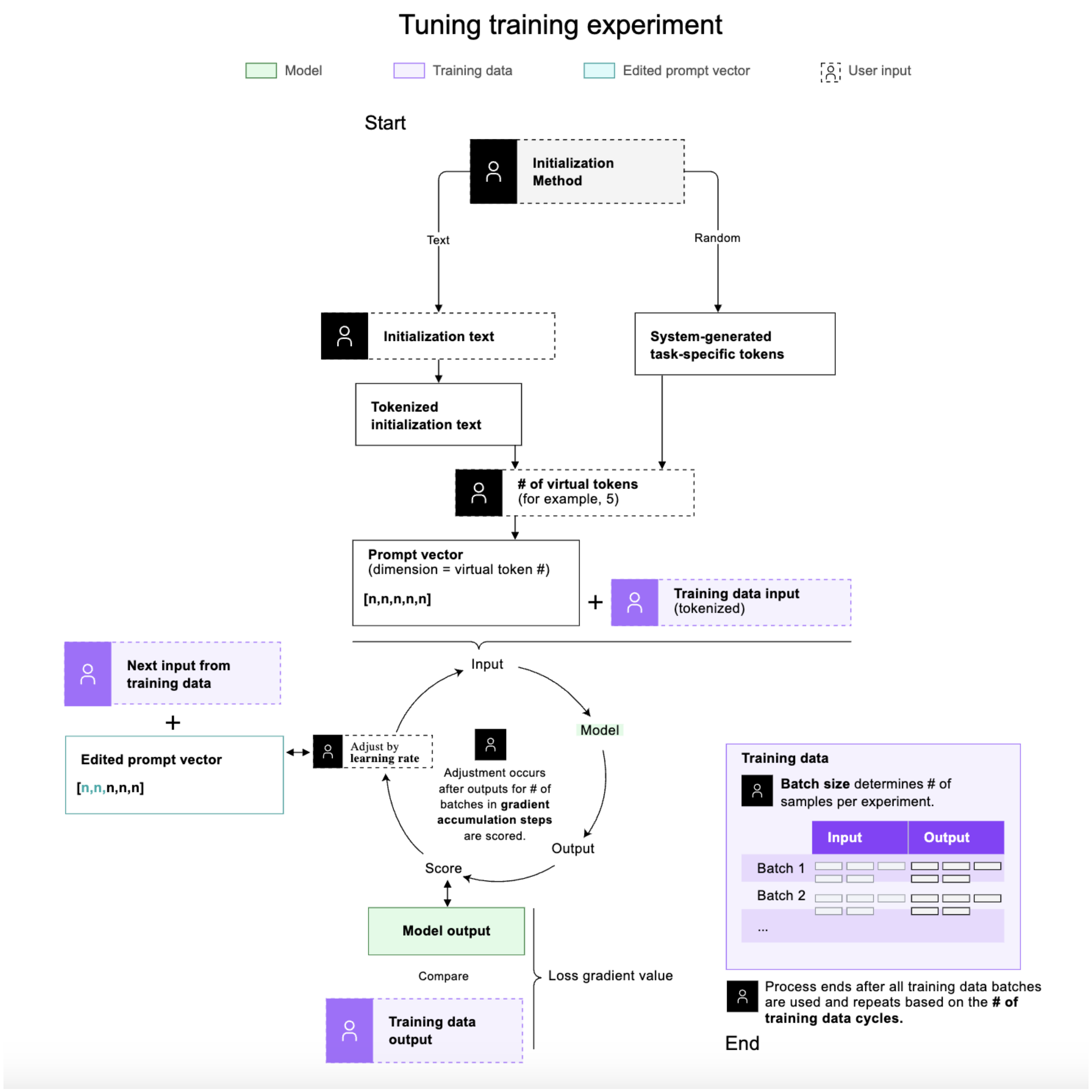
You can choose to start the training process by providing text that is vectorized by the experiment. Or you can let the experiment use random values in the prompt vector. Either way, unless the initial values are exactly right, they will be changed repeatedly as part of the training process. Providing your own initialization text can help the experiment reach a good result more quickly.

The result of the experiment is a tuned version of the underlying model. You submit input to the tuned model for inferencing and the model generates output that follows the tuned-for pattern.

Prompt-tuning workflow

During the experiment, the tuning model repeatedly adjusts the structure of the prompt so that its predictions can get better over time.

The following diagram illustrates the steps that occur during a prompt-tuning experiment run. The parts of the experiment flow that you can configure are highlighted by a **User input** icon. These decision points correspond with experiment tuning parameters that you control.


Tuning training experiment
Model
Training data
Edited prompt vector
User input
Start
Initialization Method
Text
Random
Initialization text
System-generated task-specific tokens
Tokenized initialization text
# of virtual tokens (for example, 5)
Prompt vector (dimension = virtual token #)
[n,n,n,n,n]
+
Training data input (tokenized)
Input
Model
Output
Score
Next input from training data
+
Edited prompt vector
[n,n,n,n,n]
Adjust by learning rate
Adjustment occurs after outputs for # of batches in gradient accumulation steps are scored.
Model output
Compare
Training data output
Loss gradient value
Training data
Batch size determines # of samples per experiment.
Input
Output
Batch 1
Batch 2
...
Process ends after all training data batches are used and repeats based on the # of training data cycles.
End

The diagram shows the following steps of the experiment:

1. Starts from the initialization method that you choose to use to initialize the prompt.

   If the initialization method parameter is set to text, then you must add the initialization text.

2. If specified, tokenizes the initialization text and converts it into a prompt vector.

3. Reads the training data, tokenizes it, and converts it into batches.

   The size of the batches is determined by the batch size parameter.

4. Sends input from the examples in the batch to the foundation model for the model to process and generate output.

5. Compares the model's output to the output from the training data that corresponds to the training data input that was submitted. Then, the experiment computes the loss gradient, which is the difference between the predicted output and the actual output from the training data.

   The experiment adjusts the prompt vector that is added to the input based on the computed loss of the model. When this adjustment occurs depends on how the accumulation steps parameter is configured.

6. Adjustments are applied to the prompt vector that was initialized in Step 2. The degree to which the vector is changed is controlled by the learning rate parameter. The edited prompt vector is added as a prefix to the input from the next example in the training data and is submitted to the model as input.

7. The process repeats until all of the examples in all of the batches are processed.

8. The entire set of batches are processed again as many times as is specified in the Number of epochs parameter.

Note: No layer of the base foundation model is changed during this process.

Learn more

- IBM.com: What is prompt-tuning?
- Research paper: The Power of Scale for Parameter-Efficient Prompt Tuning