DeepLearning.AI

Core Techniques in Fine-tuning and RL

Data: What you need and how to prepare it

Post-training lives or die on data

The data you need:

Fine-tuning data

- Pairs: {input, target output}
  - For reasoning: target output = “think” + answer

Sample fine-tuning data

Input		Alice has 3 apples and buys 2 more. How many now?
Target Output		5

Pair: {input, target output}

Sample fine-tuning data

Input

Alice has 3 apples and buys 2 more.
How many now?

Target Output

<think>
Start with 3.
Buys 2 ⇒ 3+2=5.
</think>

<answer>5</answer>

Pair: {input, “think” + answer}

Post-training lives or die on data

The data you need:

Fine-tuning data

- Pairs: {input, target output}
  - Reasoning: {input, “think” + answer}

RL data

- List: {inputs} → Tuples: {input, output, reward} from input + graders + environment

Sample RL data

Input

Alice has 3 apples and buys 2 more.
How many now?

Model Output

<think>
Start with 3.
Buys 2 ⇒ 3+2=5.
</think>

<answer>5</answer>

Tuple: {input, output}

← “rollout”

Sample RL data

Input

Alice has 3 apples and buys 2 more.
How many now?

Model Output

<think>
Start with 3.
Buys 2 ⇒ 3+2=5.
</think>

<answer>5</answer>

Reward **+2**

Tuple: {input, output, reward}

Sample RL data

Input

Alice has 3 apples and buys 2 more.
How many now?

Model Output

<think>
Start with 3.
Buys 2 ⇒ 3+2=5.
</think>

<answer>5</answer>

Reward **+2** ← informed by reward model

Tuple: {input, output, reward} ← "trajectory"

Post-training lives or die on data

The data you need:

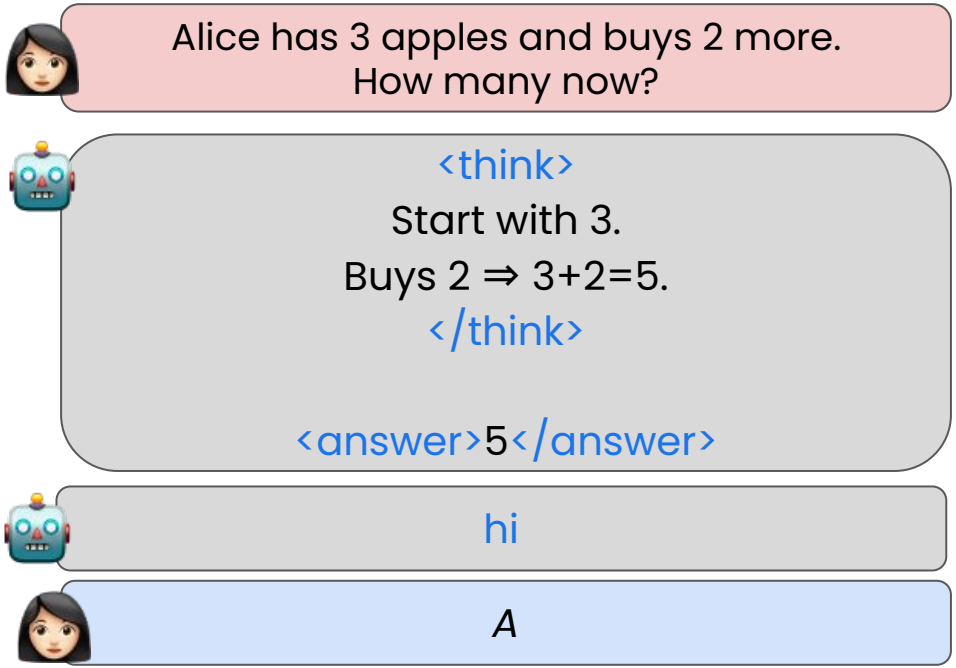
Fine-tuning data

- Pairs: {input, target output}
  - Reasoning: {input, “think” + answer}

RL data

- List: {inputs} → Tuples: {input, output, reward} from input + graders + environment
- [Optional] Tuples: {input, output A, output B, preference=which is better} for reward model

Sample reward model data for RL

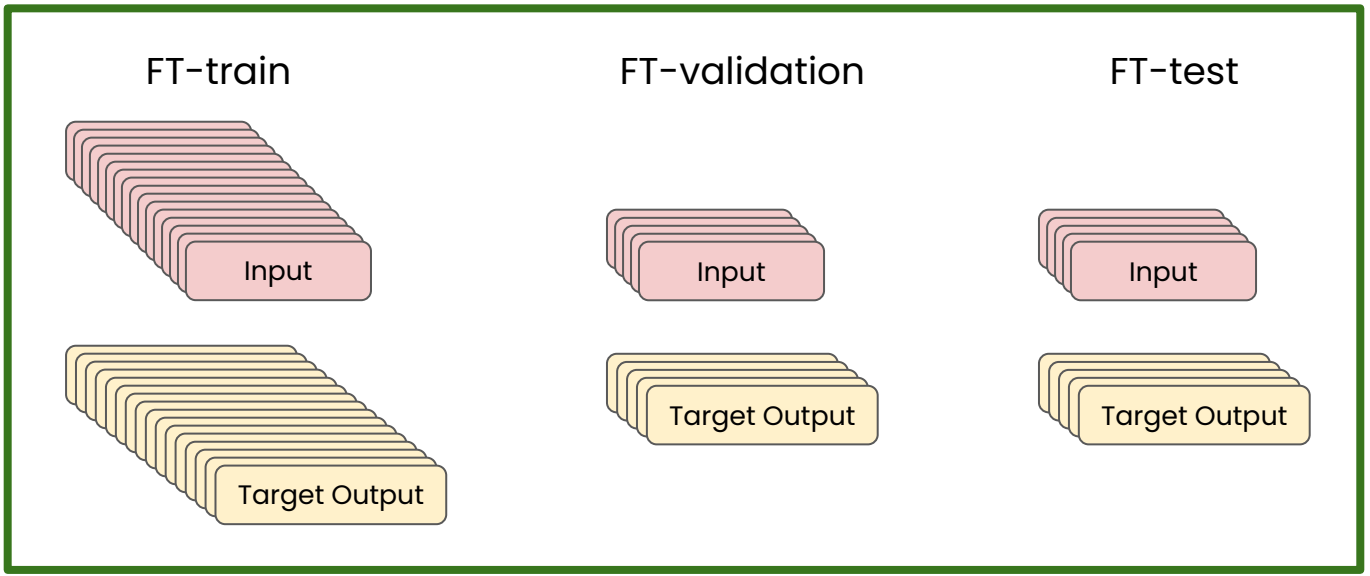


Tuple: {input, output A, output B, preference}

Split your data to trust your LLM

Fine-tuning data

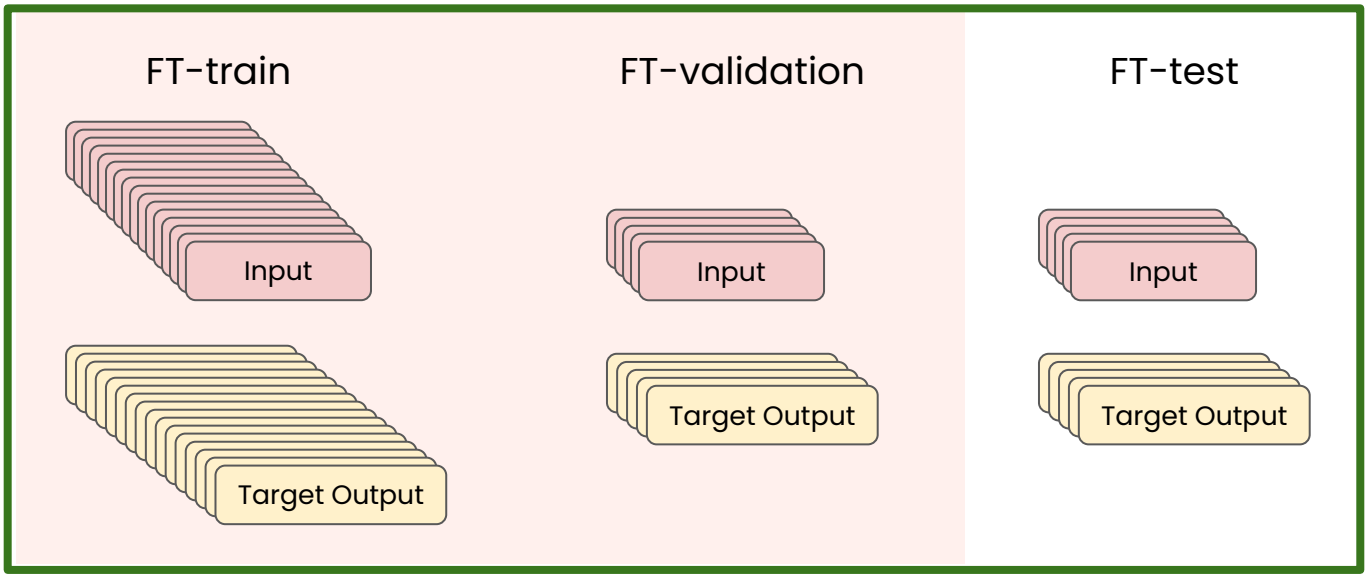
- Train: FT-train {input, target output}
- Hyperparameter tune: FT-validation {input, target output}
- Eval: FT-test {input, target output}



In your code

```
dataset = load_dataset('openai/gsm8k', 'main')
train = dataset['train']
test = dataset['test']
```

HuggingFace will take care of train/val split in the `train` dataset

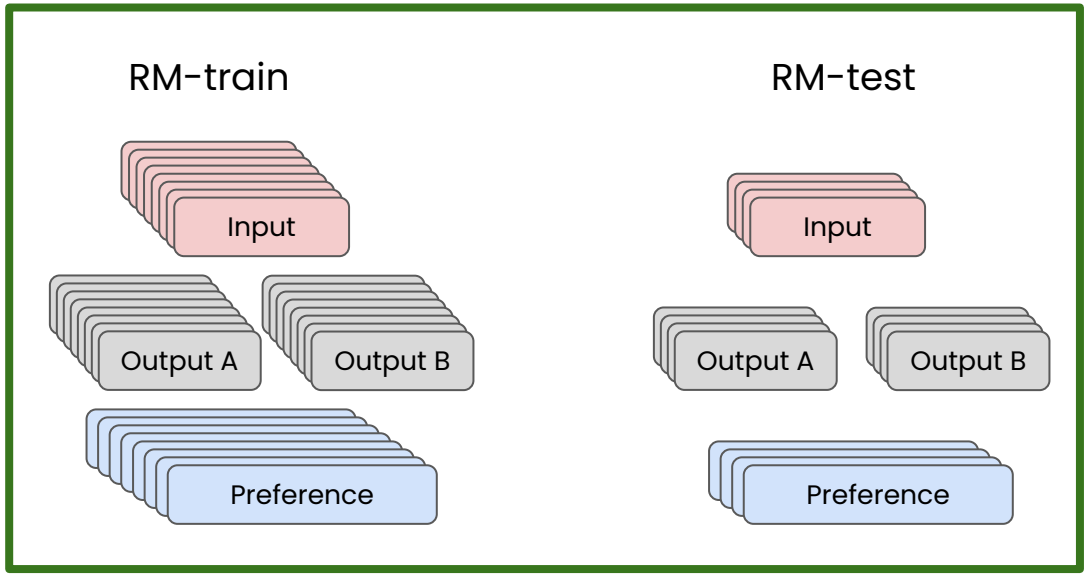


Full dataset

Split your data to trust your LLM

Reward model data

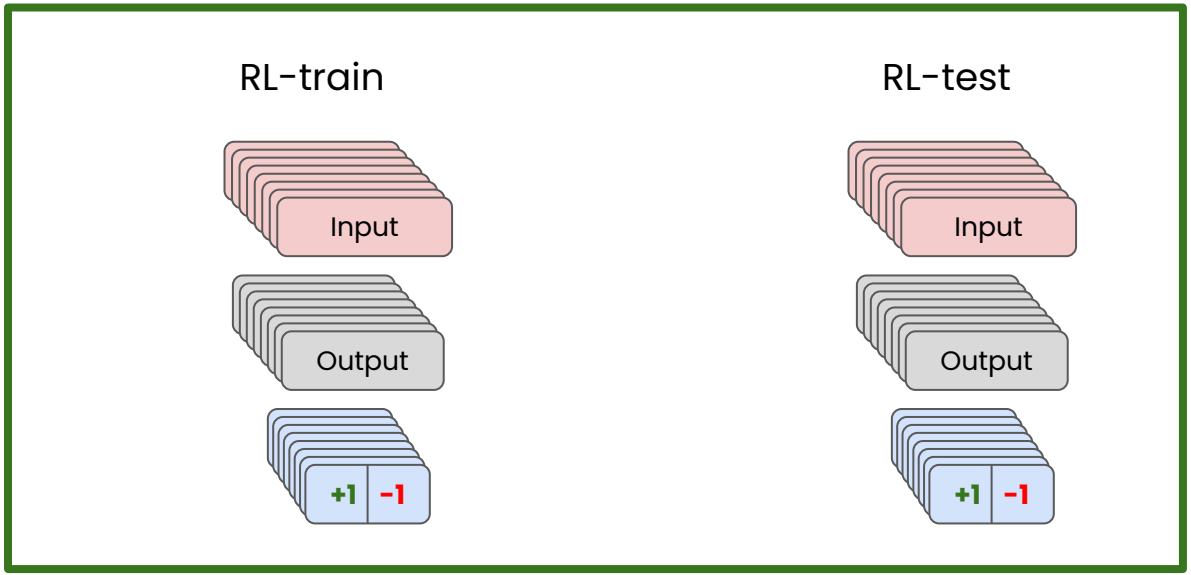
- Train: RM-train {input, output A, output B, preference}
- Eval: RM-test {input, output A, output B, preference}



Split your data to trust your LLM

RL data

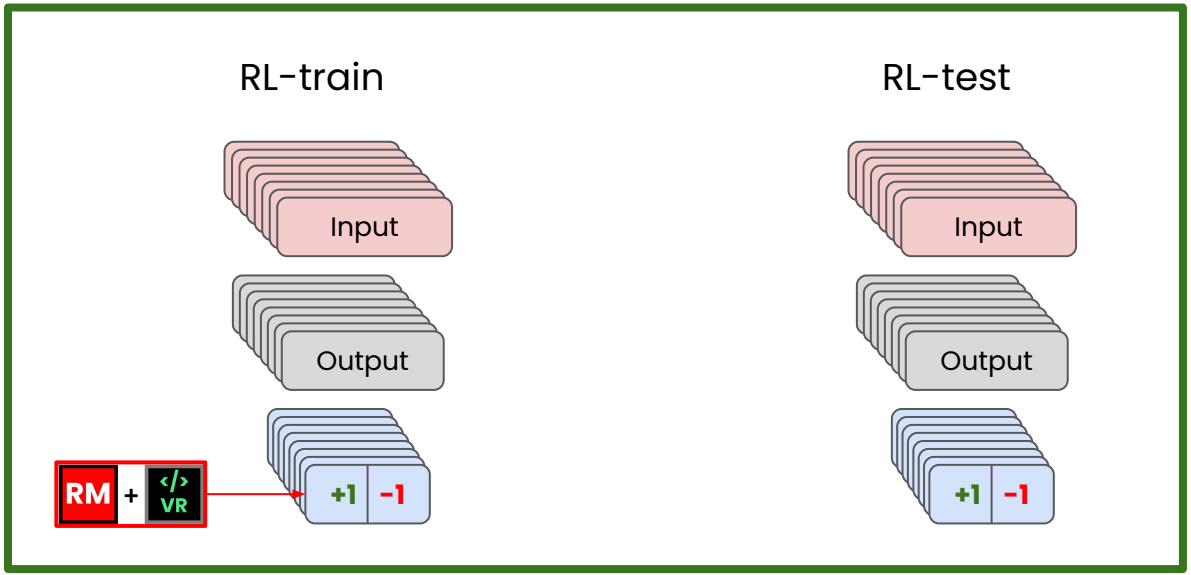
- Train: RL-Train {inputs} → {input, output, reward from RM + verifiers}
- Eval: RL-Test {inputs} → {input, output, reward from ***new*** RM + verifiers}



Split your data to trust your LLM

RL data

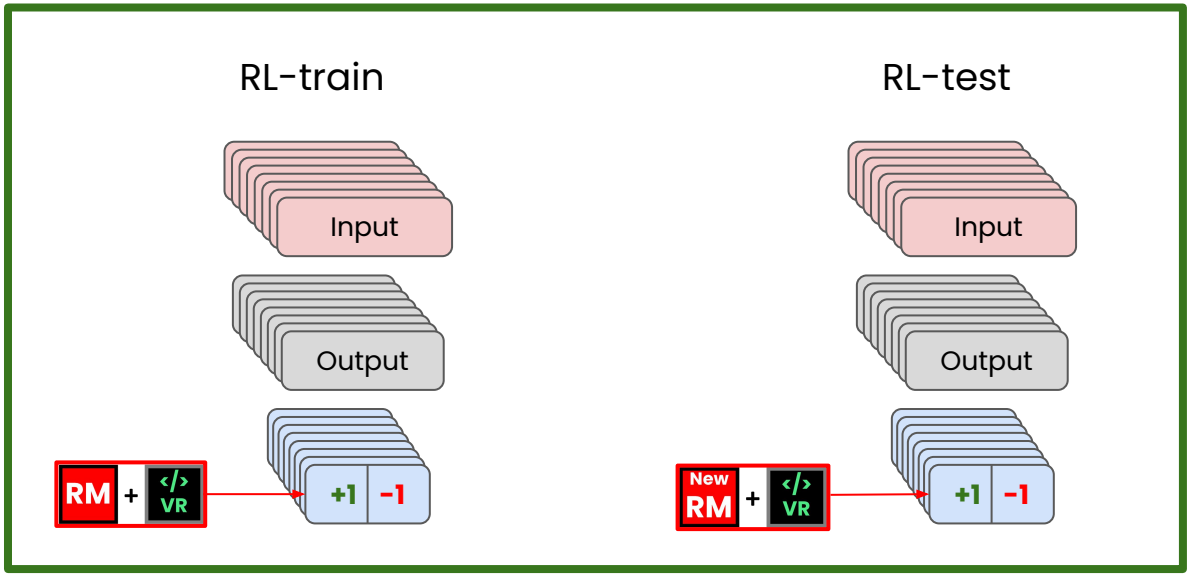
- Train: RL-Train {inputs} → {input, output, reward from RM + verifiers}
- Eval: RL-Test {inputs} → {input, output, reward from ***new*** RM + verifiers}



Split your data to trust your LLM

RL data

- Train: RL-Train {inputs} → {input, output, reward from RM + verifiers}
- Eval: RL-Test {inputs} → {input, output, reward from ***new*** RM + verifiers}

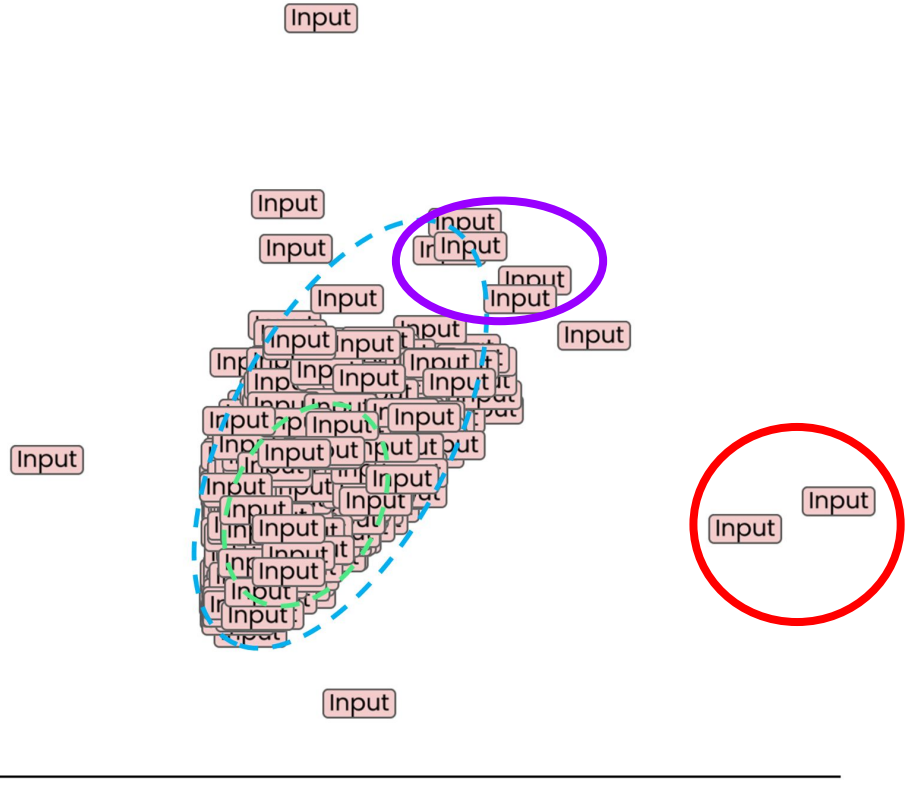


Full dataset

Split your data to trust your LLM

Final evaluation data

- Strictly unseen - mix in long-tail and out-of-distribution {inputs}



Best to keep splits apart from each other

Fine-tuning data
- Train: FT-train {input, target output}
- Hyperparameter tune: FT-validation {input, target output}
- Final: FT-test {input, target output}

Reward model data
- Train: RM-train {input, output A, output B, preference}
- Eval: RM-test {input, output A, output B, preference}

RL data
- Train: RL-Train {inputs} → {input, output, reward from RM + verifiers}
- Eval: RL-Test {inputs} → {input, output, reward from new RM + verifiers}

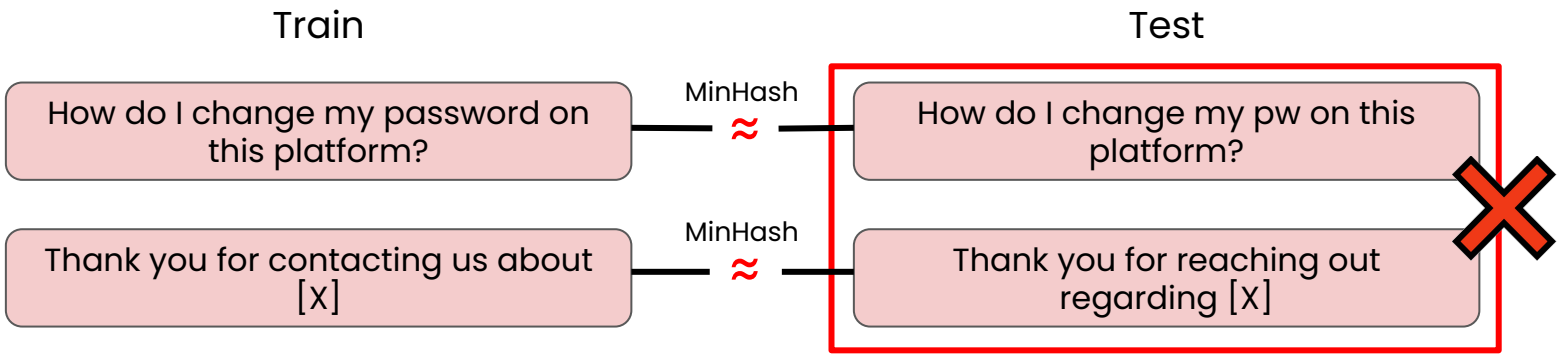
Evaluation data (evals)
- Final eval: Strictly unseen - mix in long-tail and out-of-distribution {inputs}

Leaky leaky

Leaking even similarly distributed data across splits will ruin your splits!

Control (near) duplicates: de-dupe within and across splits

- e.g., MinHash/LSH; template- and paraphrase-aware

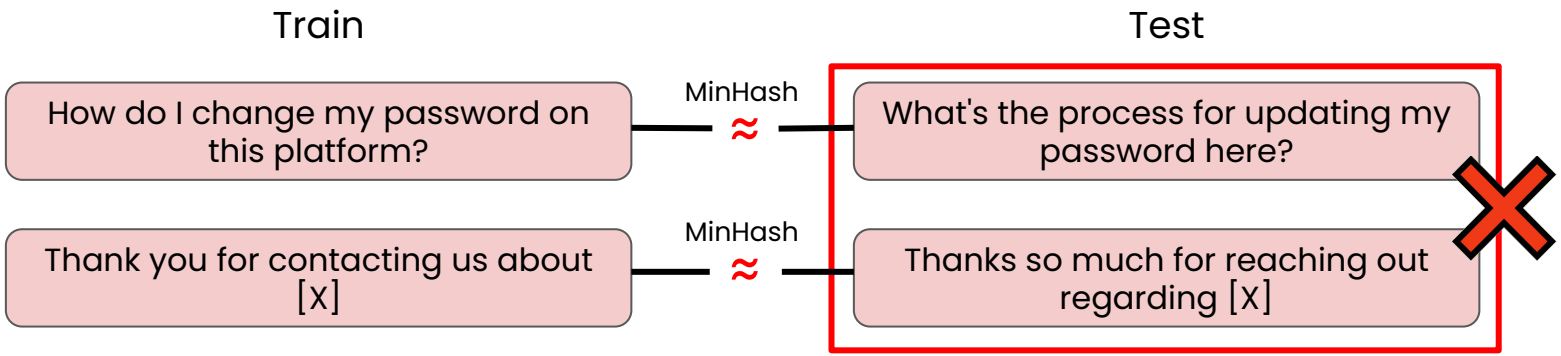


Leaky leaky

Leaking even similarly distributed data across splits will ruin your splits!

Control (near) duplicates: de-dupe within and across splits

- e.g., MinHash/LSH; template- and paraphrase-aware

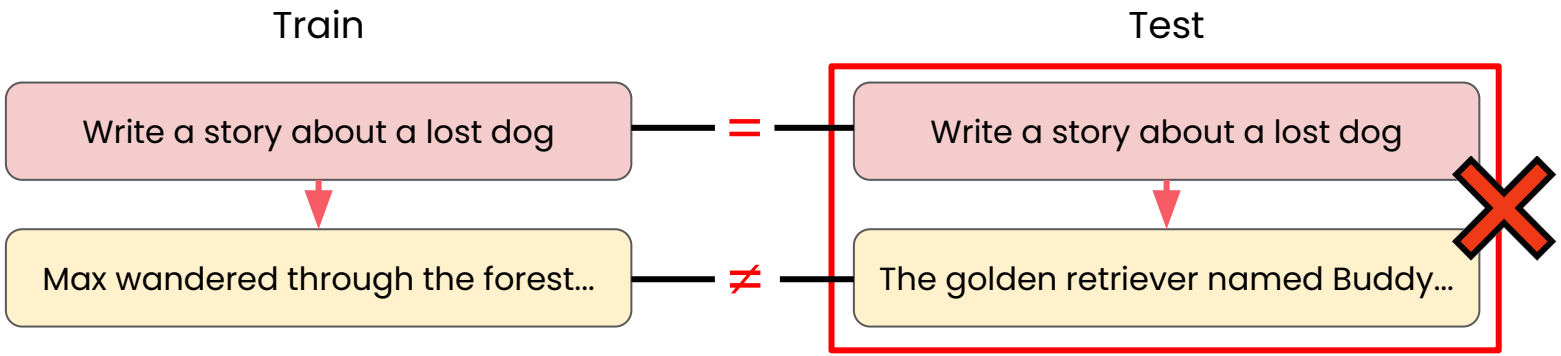


Leaky leaky

Leaking even similarly distributed data across splits will ruin your splits!

Don’t randomly split: it will likely result in leakage

- Avoid “same prompt, different target” across sets.



Leaky leaky

Leaking even similarly distributed data across splits will ruin your splits!

Time splitting: splitting your data based on time, ie. evals created after training data

- Helps with “generalizing into the future”

Medical dataset

Train (2019)

Fever, dry cough, fatigue, body aches

Seasonal flu. Rest, fluids, Tamiflu consideration

Test (2020)

Dry cough, body aches, fever, fatigue.

COVID suspected. Isolate 14 days immediately

Get paranoid about data splitting & contamination

Leaking even similarly distributed data across splits will ruin your splits!

Control (near) duplicates: de-dupe within and across splits
- e.g., MinHash/LSH; template- and paraphrase-aware

Don't randomly split: it will likely result in leakage
- Avoid "same prompt, different target" across sets.

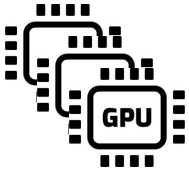
Time splitting: splitting your data based on time, ie. evals created *after* training data
- Helps with "generalizing into the future"

Why data preparation is a massive effort!

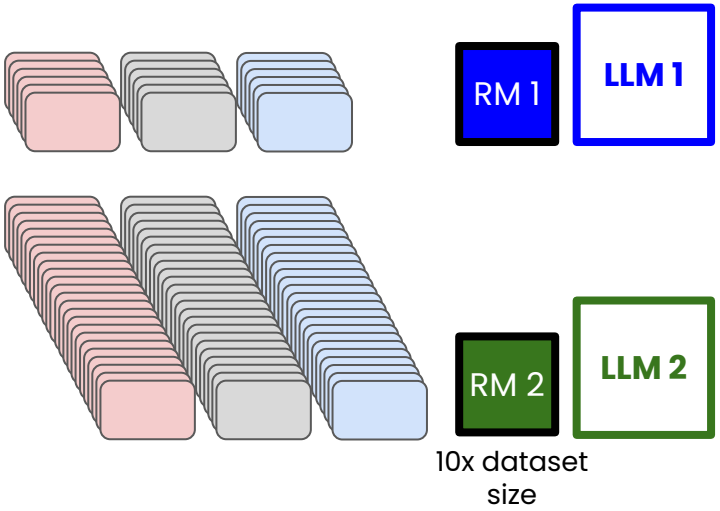
In next modules...

Avoid 10x more experiments, by designing your RL test envs and test data (evals) right.

Experiment #9540



Learn how much data you need for your LLM to successfully learn.



DeepLearning.AI

Core Techniques in Fine-tuning and RL

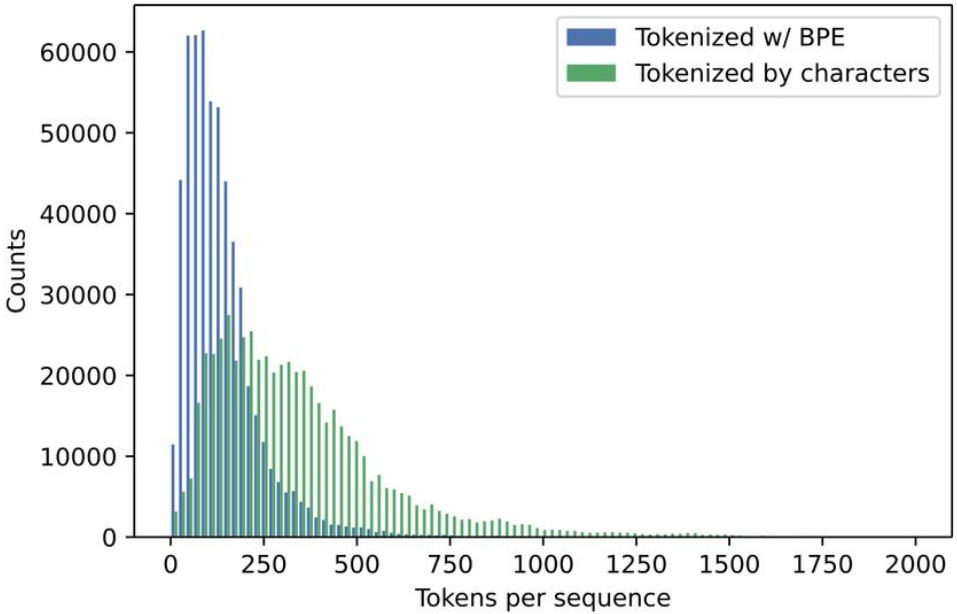
Data: Tokens for models to read/write data

Encode text into #s efficiently

Encode like people do:
- Words (dictionary)
- Characters (alphabet)

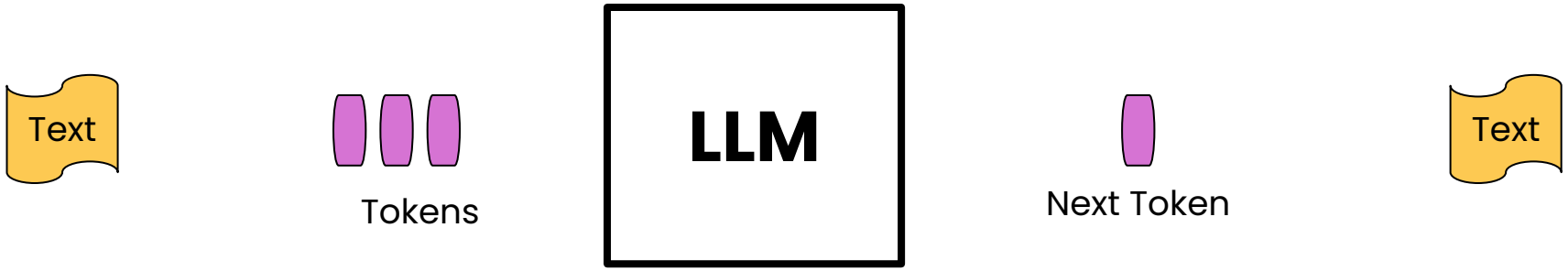
But: encode more efficiently? Tokens!
- Algorithms like BPE compress training text into most efficient strings: “ing”
- All tokens = model vocabulary
- GPT-3 has 50k BPE tokens

Fun problems with tokens: Count # r’s in strawberry

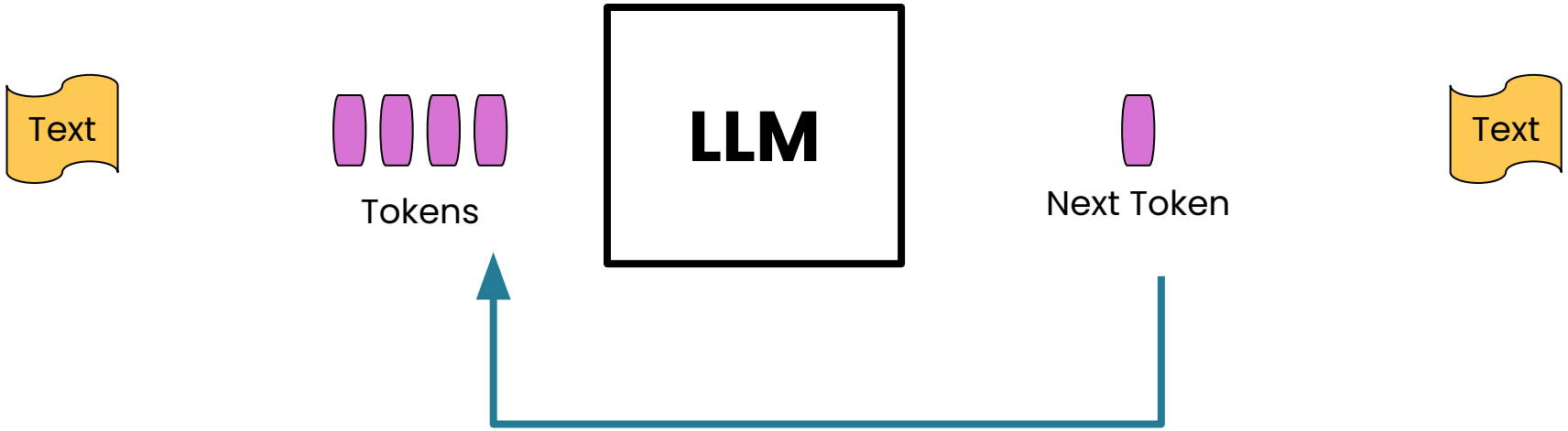


https://www.researchgate.net/figure/BPE-vocabulary-with-10K-tokens-captures-984-of-SwissProt-sequences-within-the-512-token_fig1_346701215

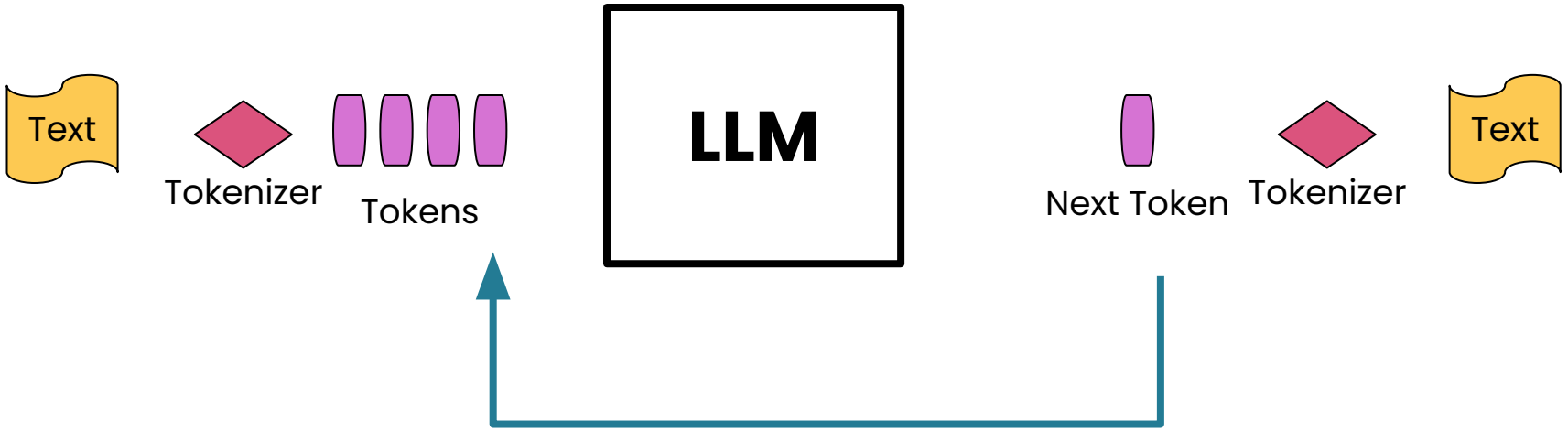
How tokens fit in



How tokens fit in



How tokens fit in



Tokenizer

Encode

Input: Text

"What words are indivisible?"

Tokens

What words are indivisible?

Output: Token IDs

2640 3073 418 3221 21142 30

Simple lookup table across *vocabulary*

```
input_ids = tokenizer(text)
```

Tokenizer

Decode

Output: Text

"What words are indivisible?"

Tokens

What words are indivisible?

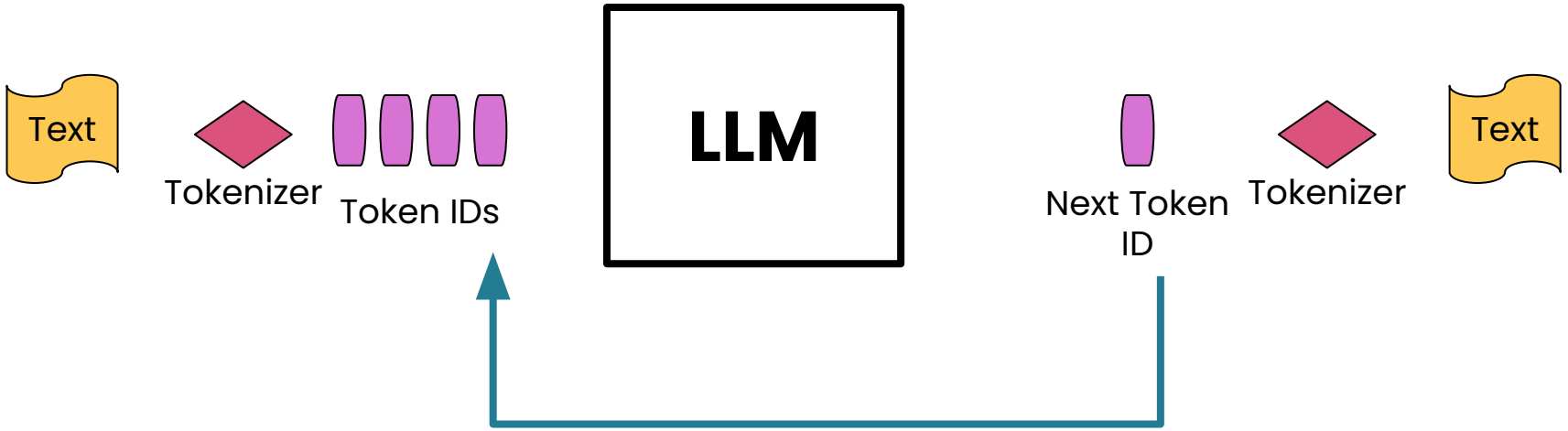
Input: Token IDs

2640 3073 418 3221 21142 30

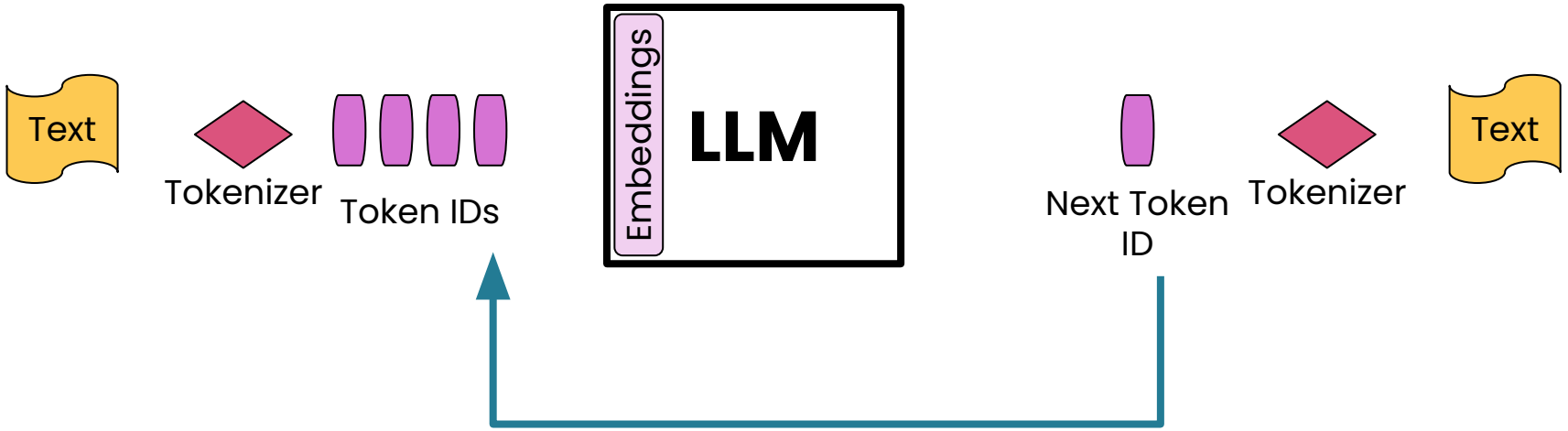
Simple lookup table across *vocabulary*

```
output_text = tokenizer.decode(output_ids)
```

How tokens fit in



How tokens fit in



Embedding lookup

Lookup

Token IDs

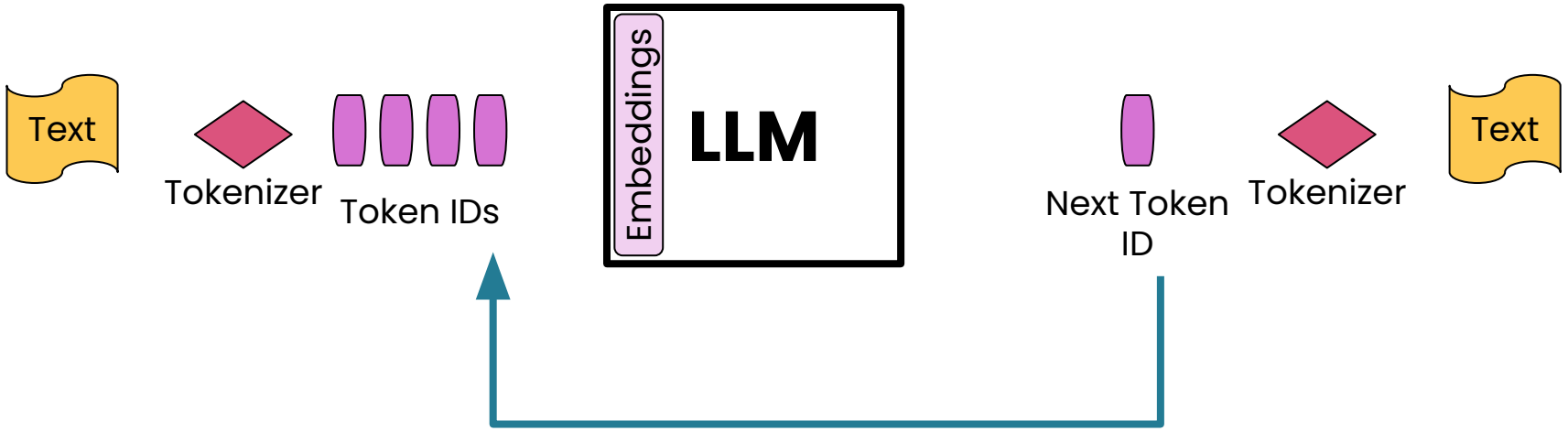
2640	3073	418	3221	21142	30

Embeddings

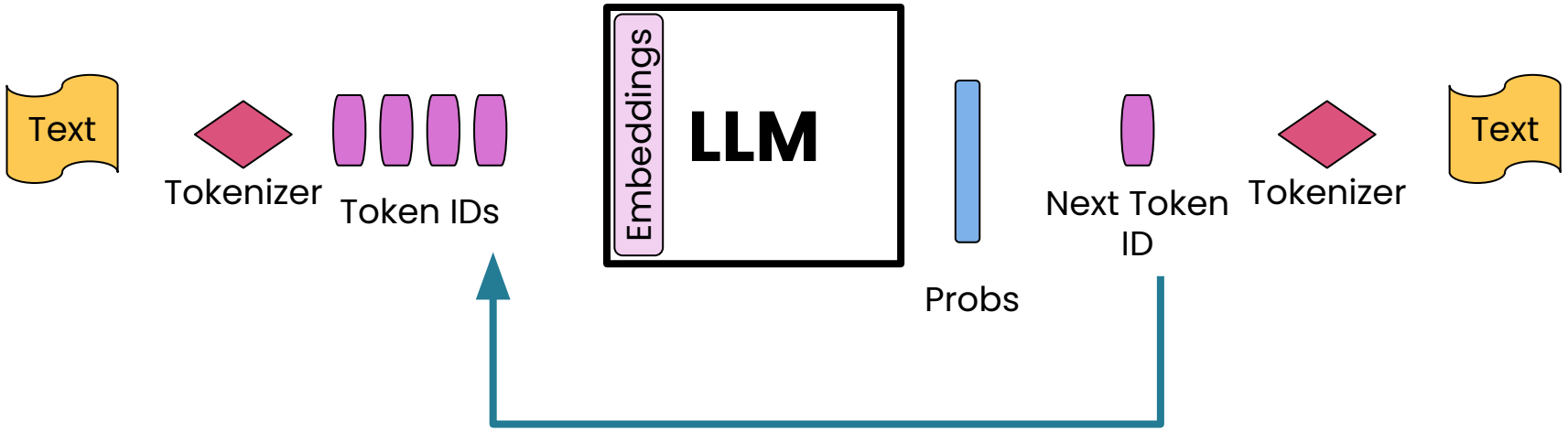
0.038	0.002	0.141	-0.085	-0.115	-0.042
-0.046	0.365	-0.027	-0.064	-0.157	-0.018
0.440	0.443	1.219	0.242	0.120	0.715
...	...	...	...	...	...

- 1st layer of model
- Size of the vocabulary, token IDs index into embedding matrix
- Trained with model to represent each token semantically

How tokens fit in



How tokens fit in



From probabilities to next token id

Sample

Probabilities over vocabulary

0.5 0.2 0.3 … 0.01

Token ID

0.5 → ID 0

- Greedy: highest probability

```
model.generate(**ids)
```

From probabilities to next token id

Probabilities over vocabulary

0.5 0.2 0.3 … 0.01

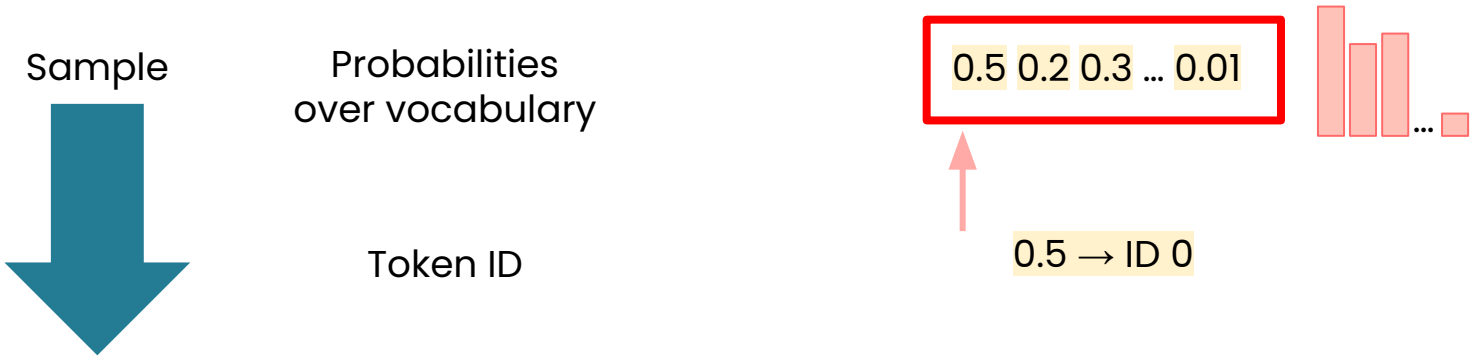
Token ID

0.5 → ID 0

- Greedy: highest probability

```
model.generate(**ids)
```

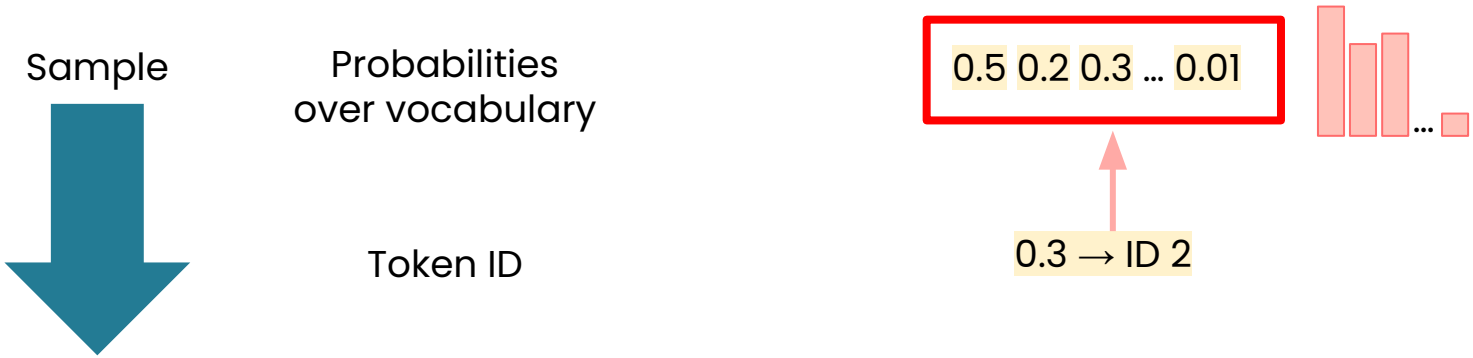
From probabilities to next token id



- Greedy: highest probability
- Sample: sample from distribution

```
model.generate(**ids, do_sample=True)
```

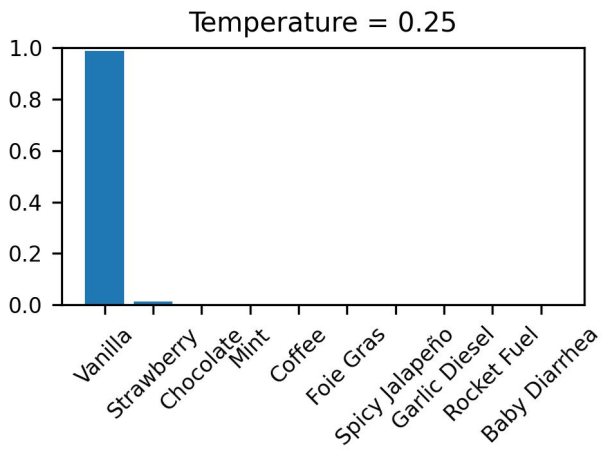
From probabilities to next token id

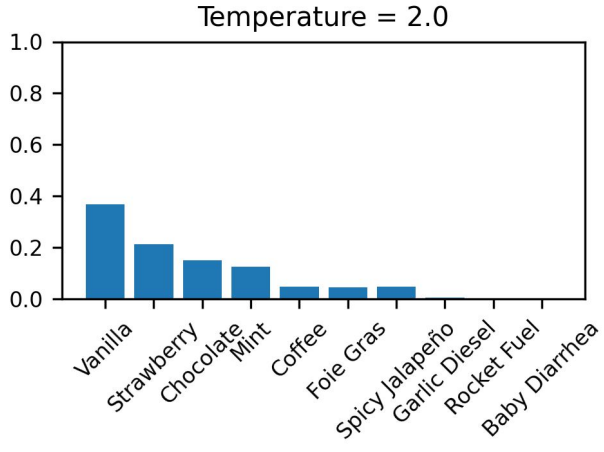


- Greedy: highest probability
- Sample: sample from distribution

```
model.generate(**ids, do_sample=True)
```

From probabilities to next token id





- Greedy: highest probability
- Sample: sample from distribution

Temperature can help you sample different outputs, per input.

```
model.generate(**ids, do_sample=True, temperature=0.7)
```

From probabilities to next token id

Probabilities over vocabulary

0.5 0.2 0.3 ... 0.01

Token ID

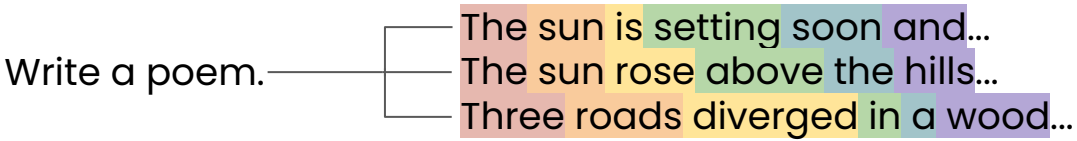
0.5 → ID 0

0.3 → ID 2

- Greedy: highest probability
- Sample: sample from distribution
- Beam search: track N different sequences

```
model.generate(**ids, num_beams=2)
```

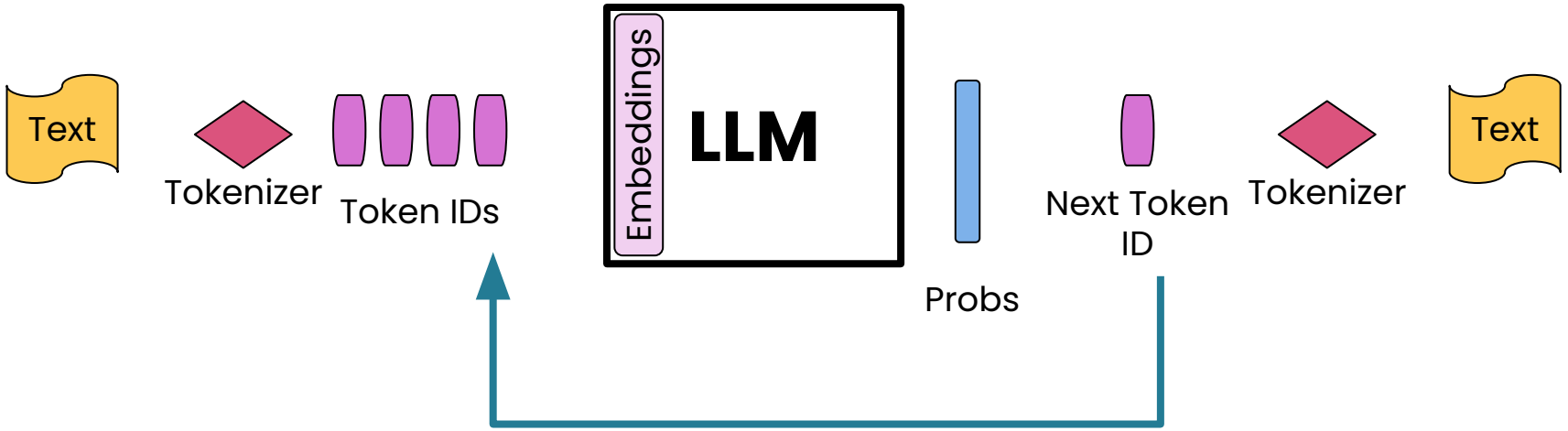
From probabilities to next token id



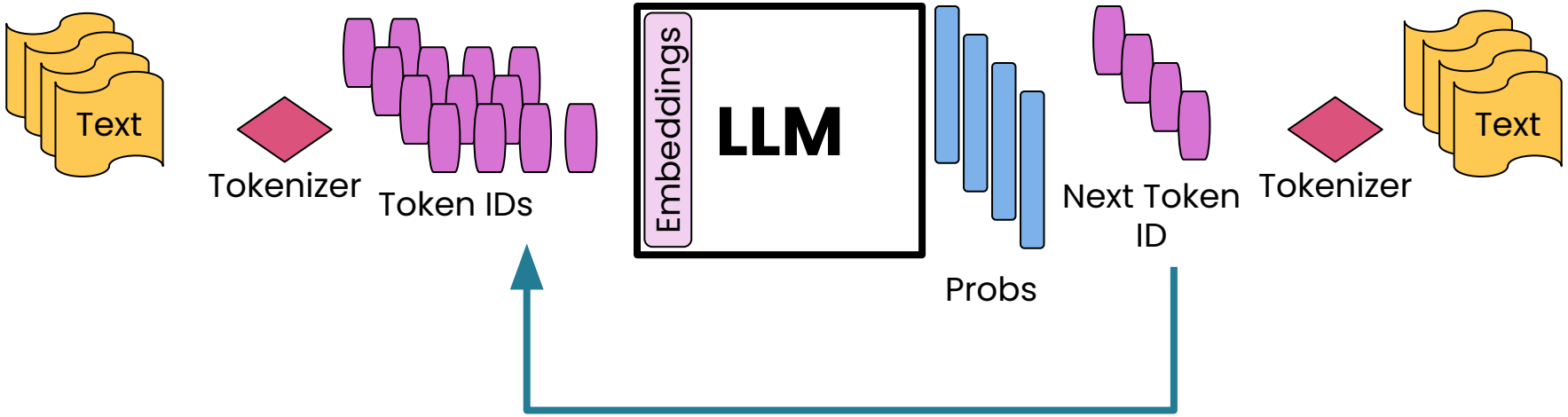
- Greedy: highest probability
- Sample: sample from distribution
- Beam search: track N different sequences

```
model.generate(**ids, num_beams=3)
```

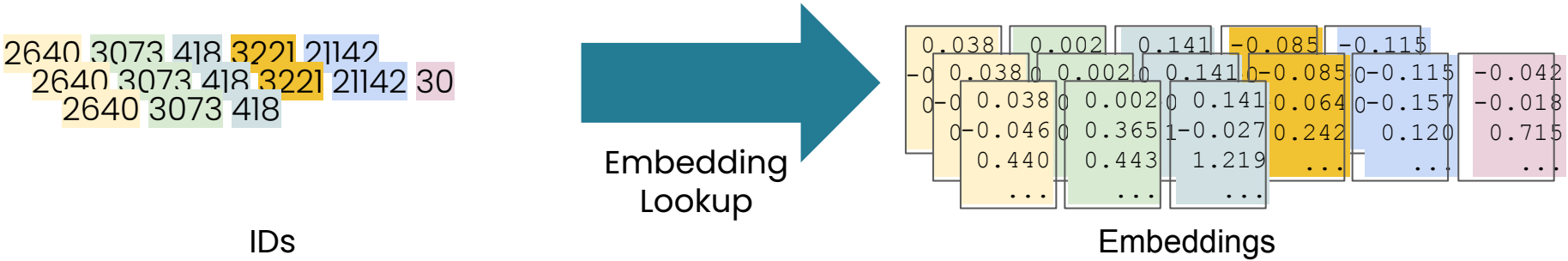
How tokens fit in



How tokens fit in

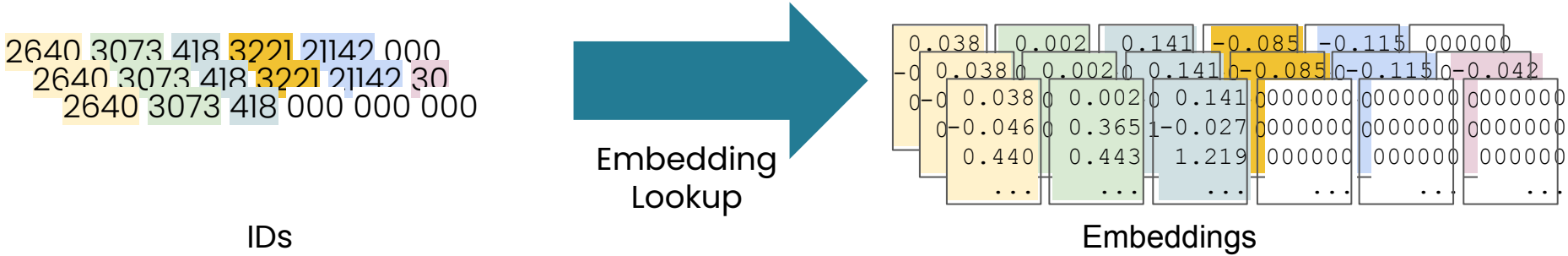


Batching


2640 3073 418 3221 21142
2640 3073 418 3221 21142 30
2640 3073 418
Embedding Lookup
IDs
Embeddings

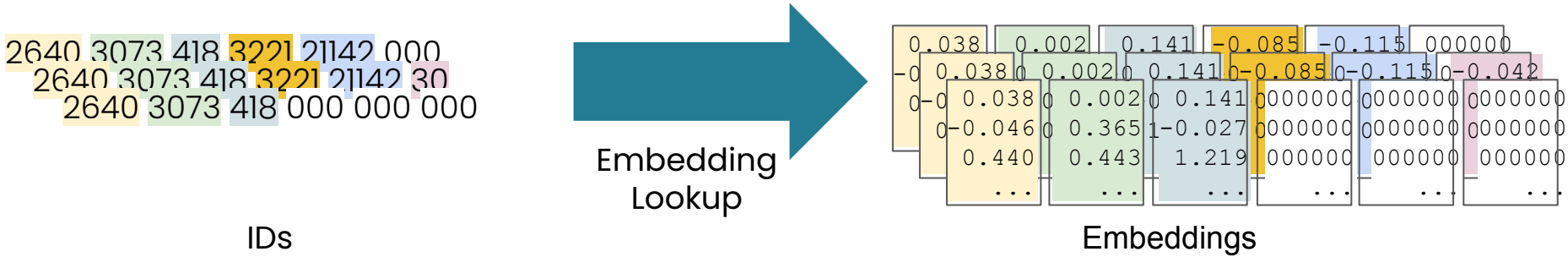
Padding tokens in a batch

"Padding" tokens make inputs same length



Padding tokens in a batch

"Padding" tokens make inputs same length




```
AutoTokenizer.from_pretrained(model_name)(prompt_batch, padding=True)['input_ids']
tensor([[0,      0,   2640,   3073,    418,   3221,  21142],
        [0,      0,   3073,    418,   3221,  21142,     30],
        [0,      0,      0,      0,   2640,   3073,    418]])
```

Models are tied to their tokenizers

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

Compare different tokenizers

Many tokenizers use the transformers library from HuggingFace

```
from transformers import AutoTokenizer

prompt = 'Using    Huggingface  is pretty manageable   '

AutoTokenizer.from_pretrained('bert-base-uncased').tokenize(prompt)
['using', 'hugging', '##face', 'is', 'pretty', 'manage', '##able']
```

Compare different tokenizers

Many tokenizers use the transformers library from HuggingFace

```
from transformers import AutoTokenizer

prompt = 'Using    Huggingface  is pretty manageable   '

AutoTokenizer.from_pretrained('t5-small').tokenize(prompt)
['_', 'Using', '_Hug', 'ging', 'face', '_is', '_pretty', '_manage', 'able']
```

Compare different tokenizers

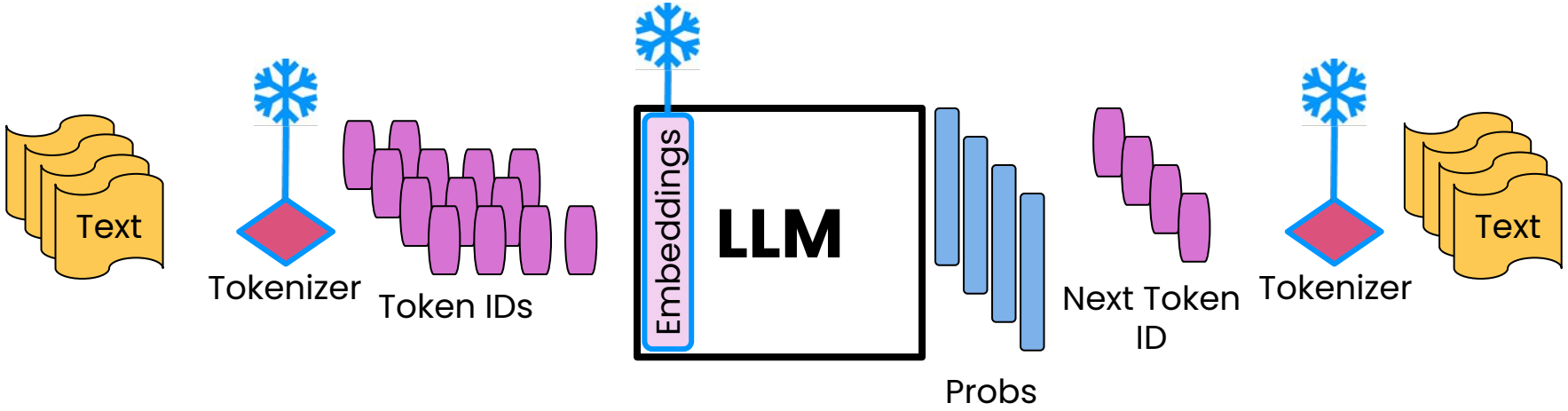
Many tokenizers use the transformers library from HuggingFace

```
from transformers import AutoTokenizer

prompt = 'Using    Huggingface  is pretty manageable   '

AutoTokenizer.from_pretrained('DeepSeek-ai/DeepSeek-math-7b-base').tokenize
(prompt)
['Using', 'ĠĠĠ', 'ĠHug', 'ging', 'face', 'Ġ', 'Ġis', 'Ġpretty',
'Ġmanageable', 'ĠĠĠ']
```

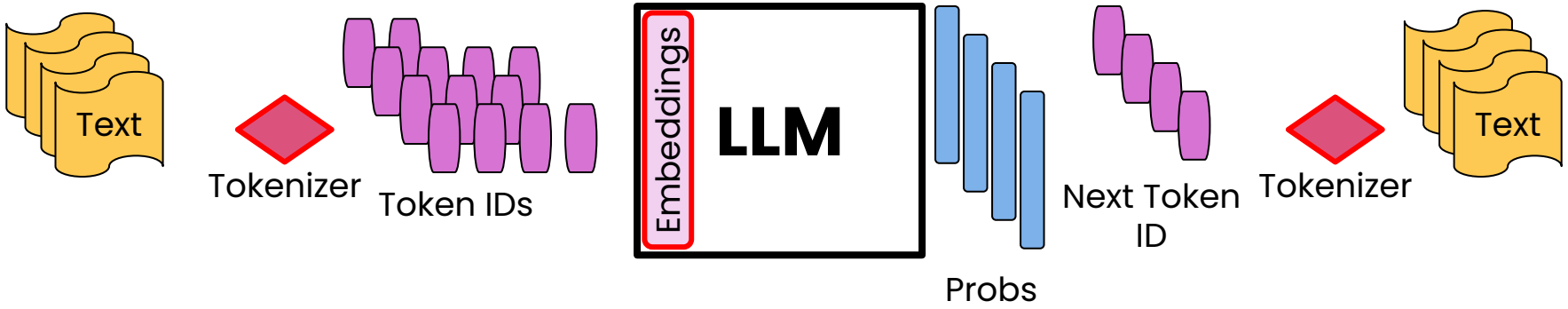
Tokenization in post-training



Post-training:

- When making <u>small</u> changes → freeze embeddings
- When keeping same vocab (e.g. RL) → freeze tokenizer

Tokenization in post-training



Post-training:

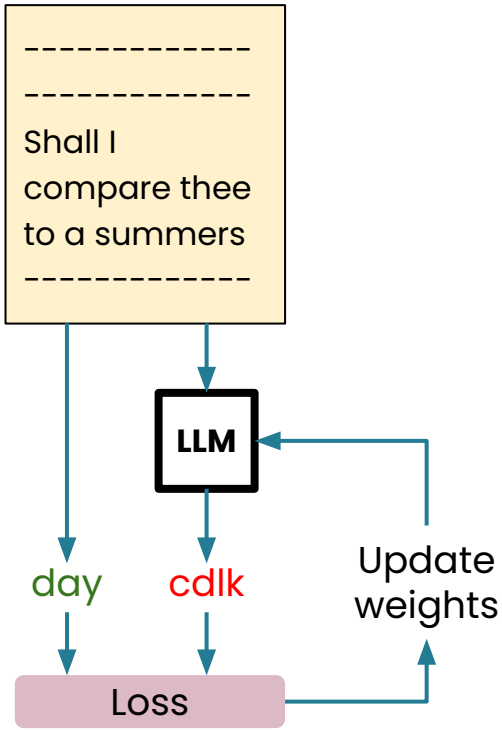
- When making large changes → train embeddings
- When adding new terms or special tags → train tokenizer
  - Special tags, e.g. <|THINK>, <|THINGY>

DeepLearning.AI

Core Techniques in Fine-tuning and RL

Fine-tuning math: Loss, gradients, weight updates (Part 1)

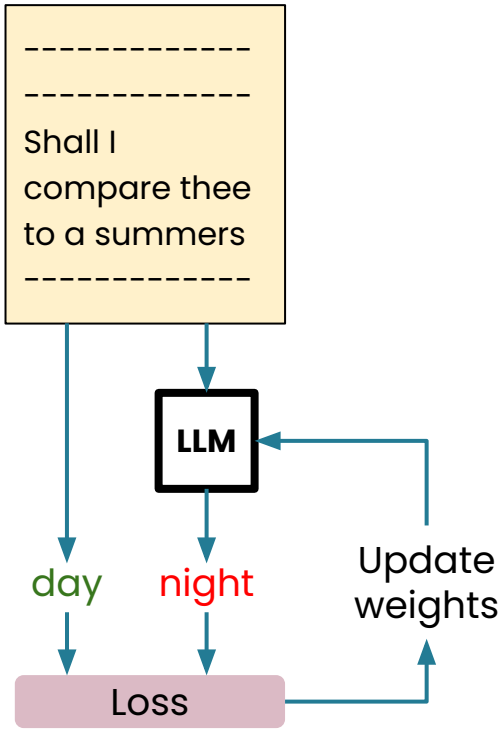
Refresher: Training a neural network



What's going on?

- Add training data
- Calculate loss
- Backprop through model
- Update weights

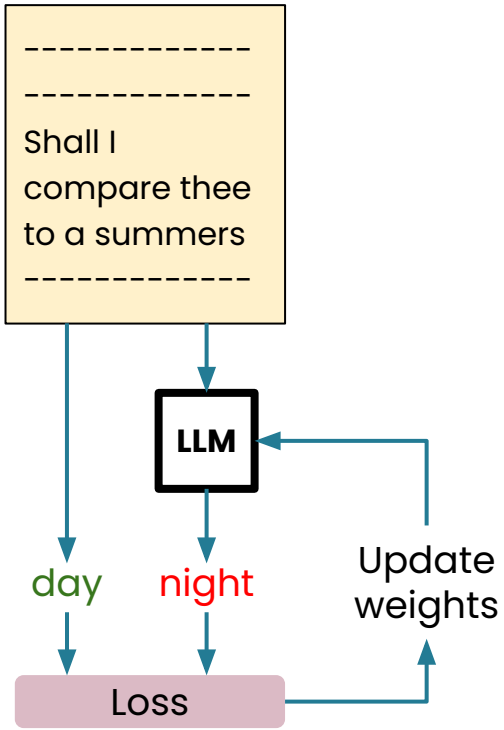
Fine-tuning is a type of training



Similar steps

- Add training data {input, target output}
- Calculate loss on the outputs
- Backprop through model
- Update weights

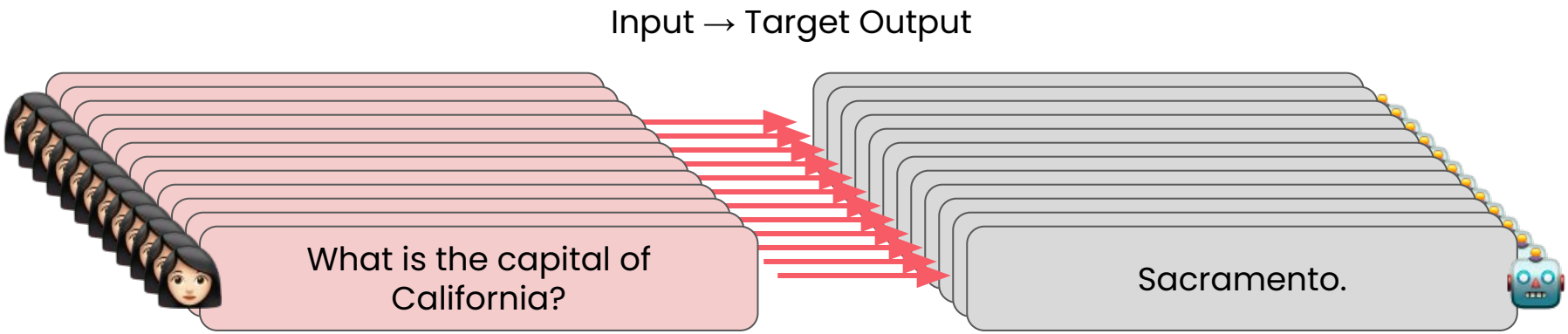
Fine-tuning is a type of training



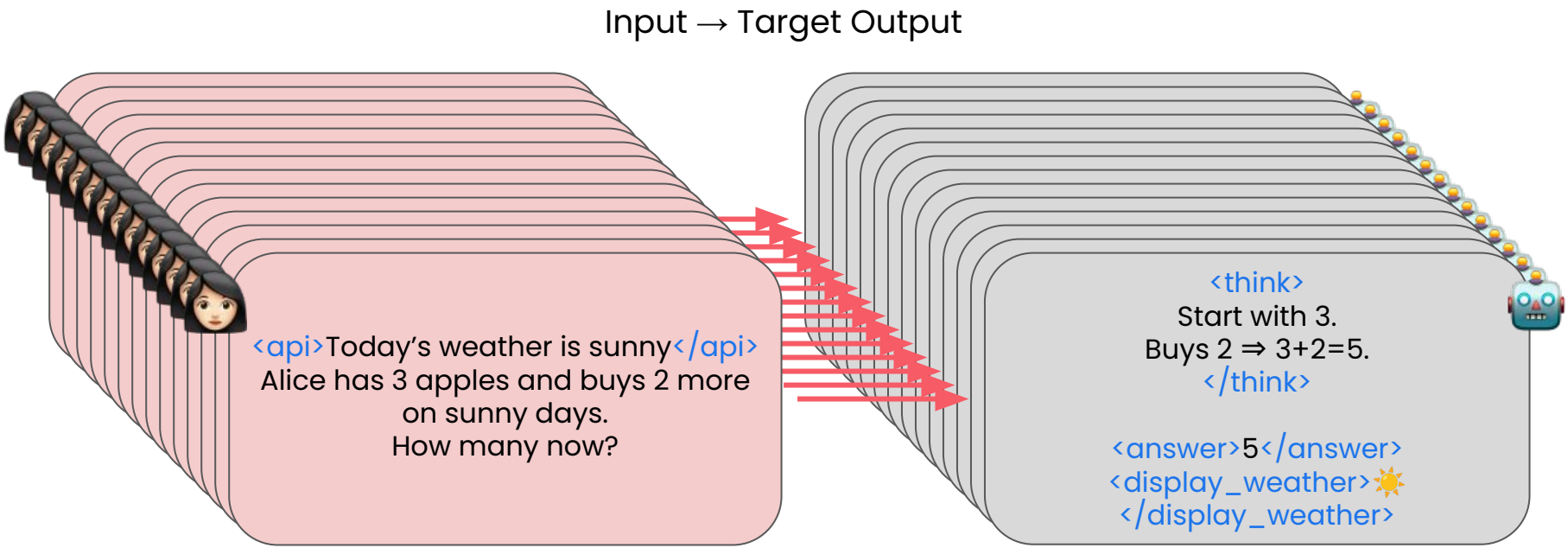
Similar steps

- Add training data {input, target output}
- Calculate loss on the outputs
- Backprop through model
- Update weights

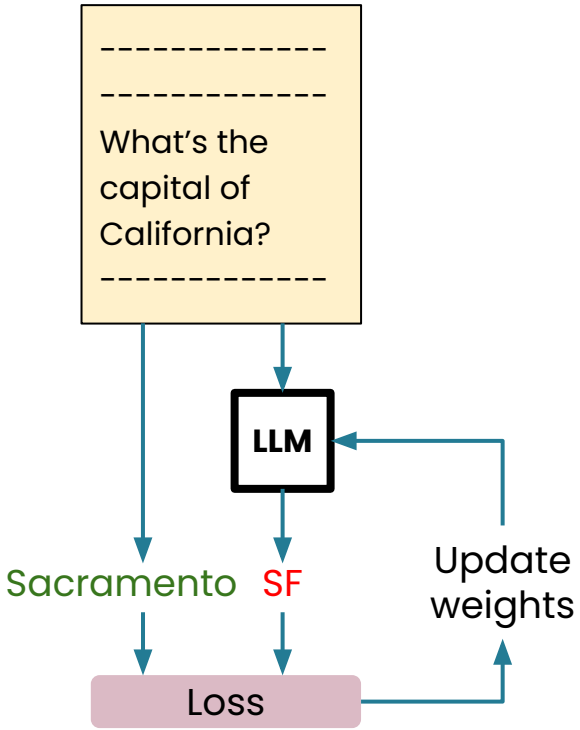
Input-target output pairs



Input-target output pairs



Calculating loss only on the model outputs

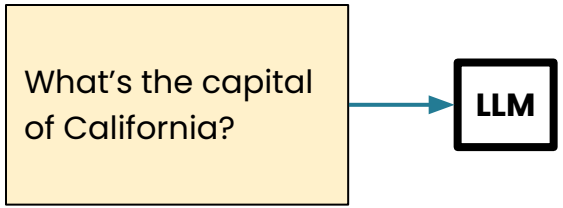


Similar steps

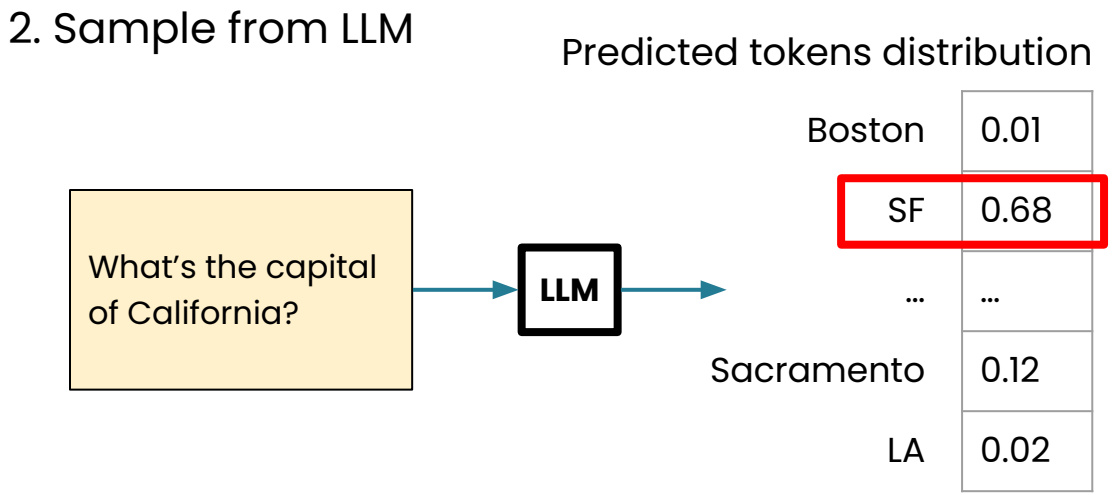
- Add training data {input, target output}
- Calculate loss on the outputs
- Backprop through model
- Update weights

Calculating loss only on the model outputs

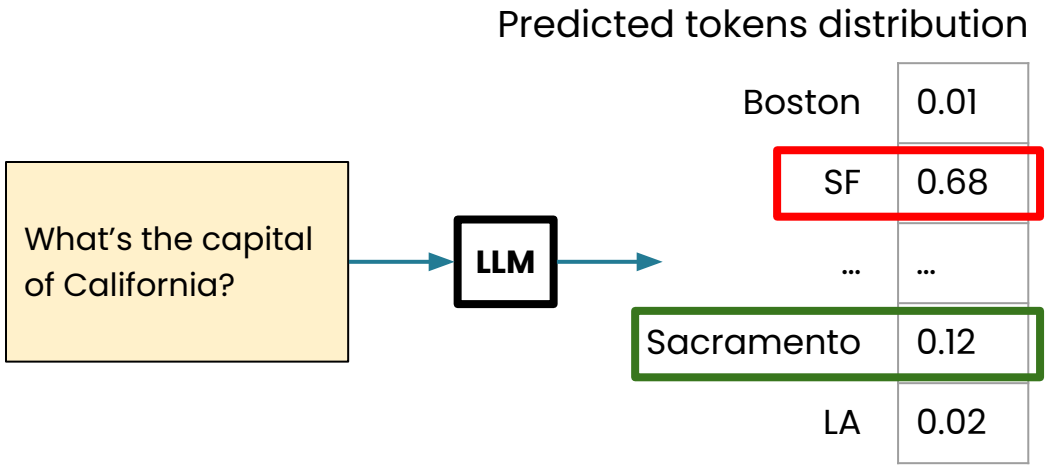
1. Given an input...



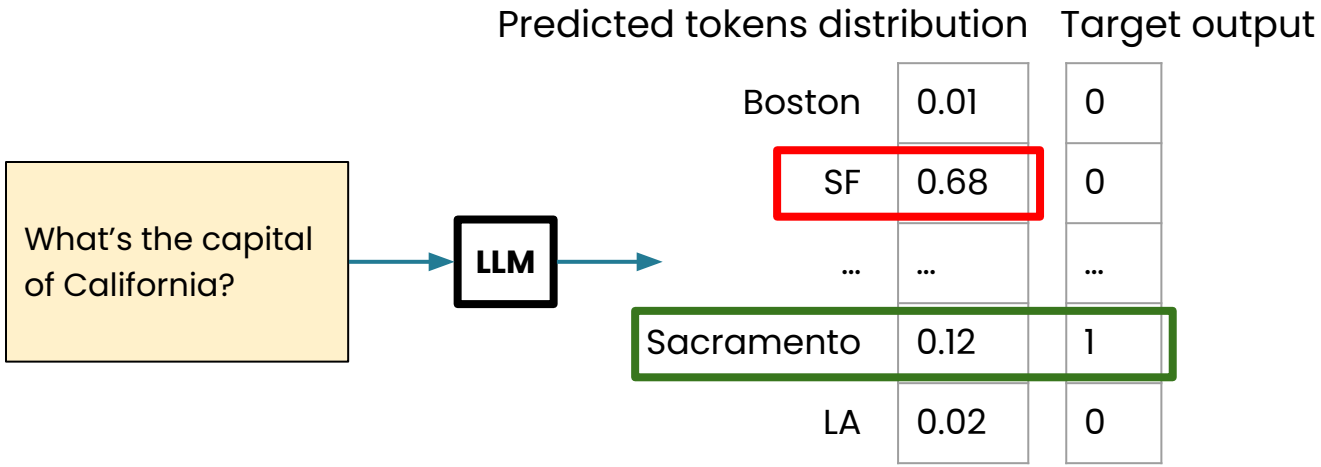
Calculating loss only on the model outputs



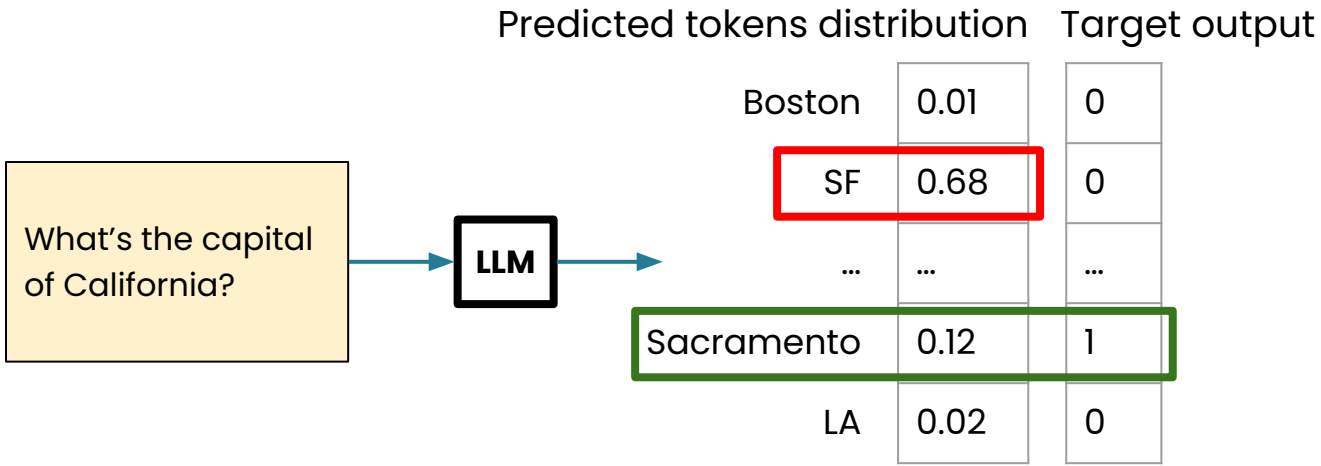
Calculating loss only on the model outputs



Calculating loss only on the model outputs

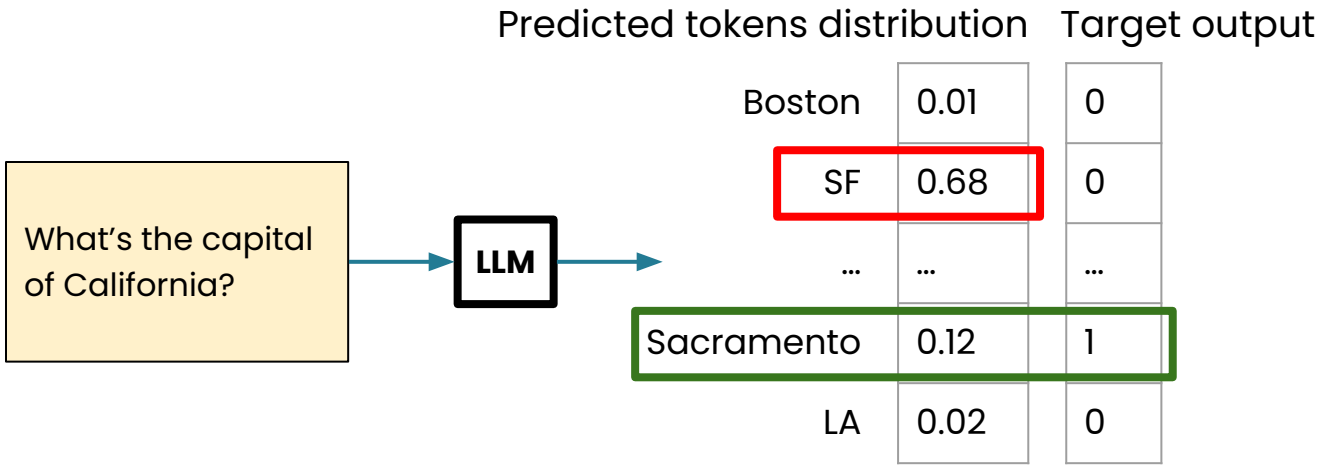


Calculating loss only on the model outputs



$$\text{Token loss} = -\log(P_{\text{correct}})$$

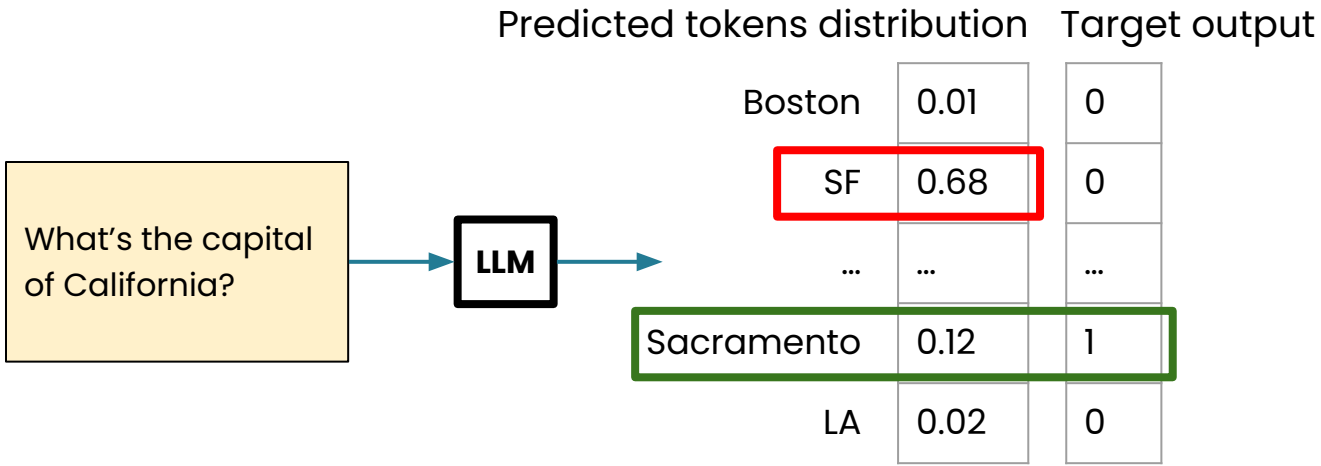
Calculating loss only on the model outputs



Cross Entropy (CE) Loss: maximize prob of correct token

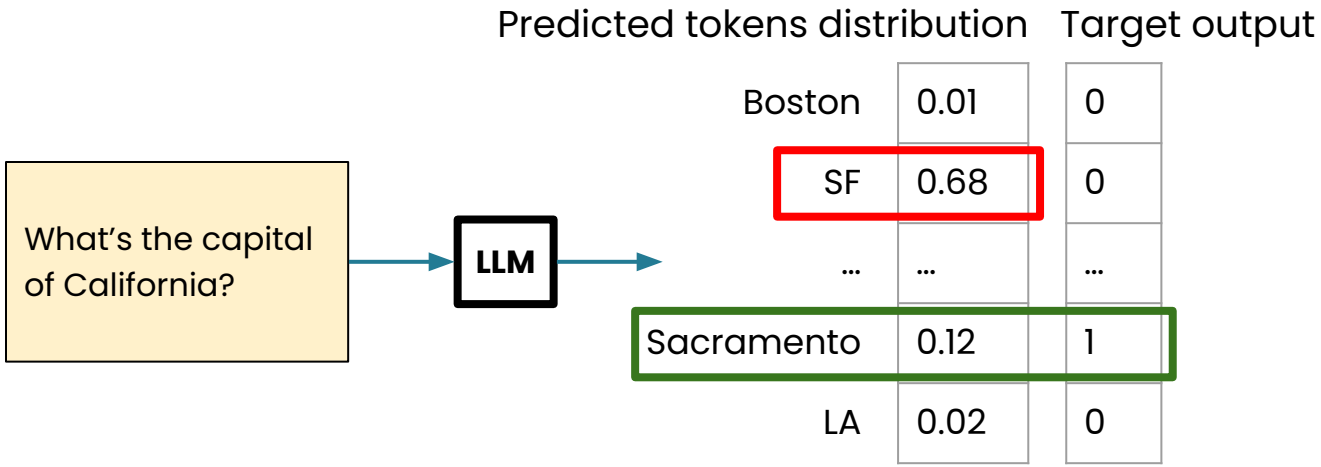
$$\text{Token loss} = -\log(P_{\text{correct}})$$

Calculating loss only on the model outputs



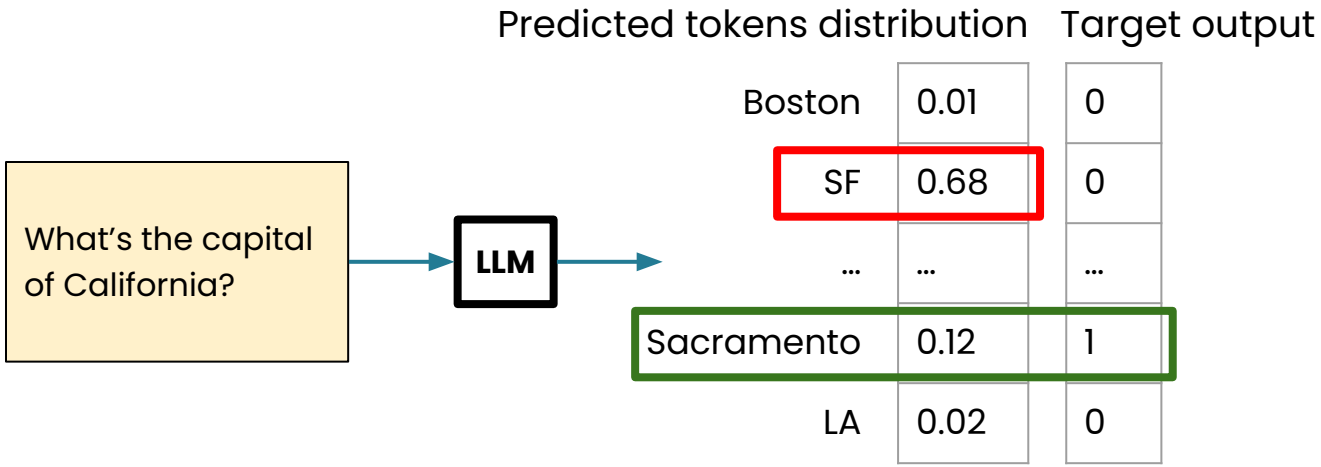
$$\text{Token loss} = -\log(P_{\text{correct}})$$

Calculating loss only on the model outputs



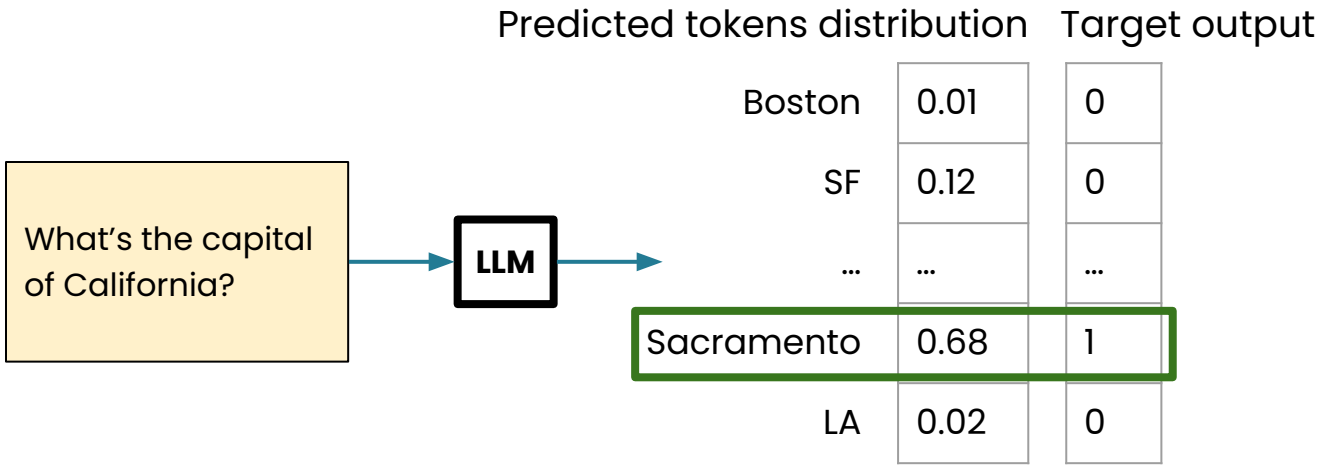
Token loss = -log(0.12)

Calculating loss only on the model outputs



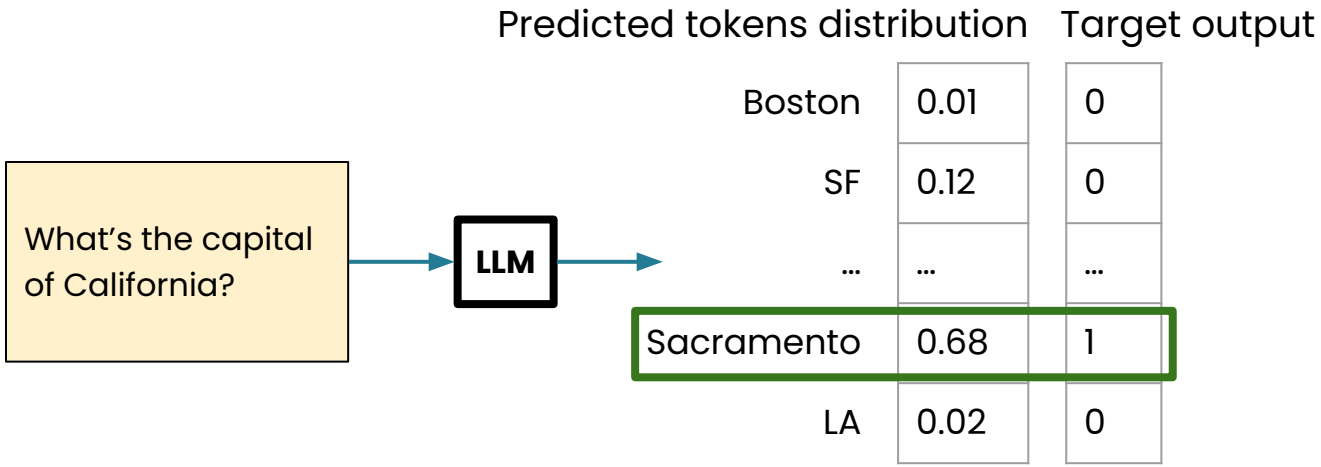
Token loss ~= 2.12

Calculating loss only on the model outputs



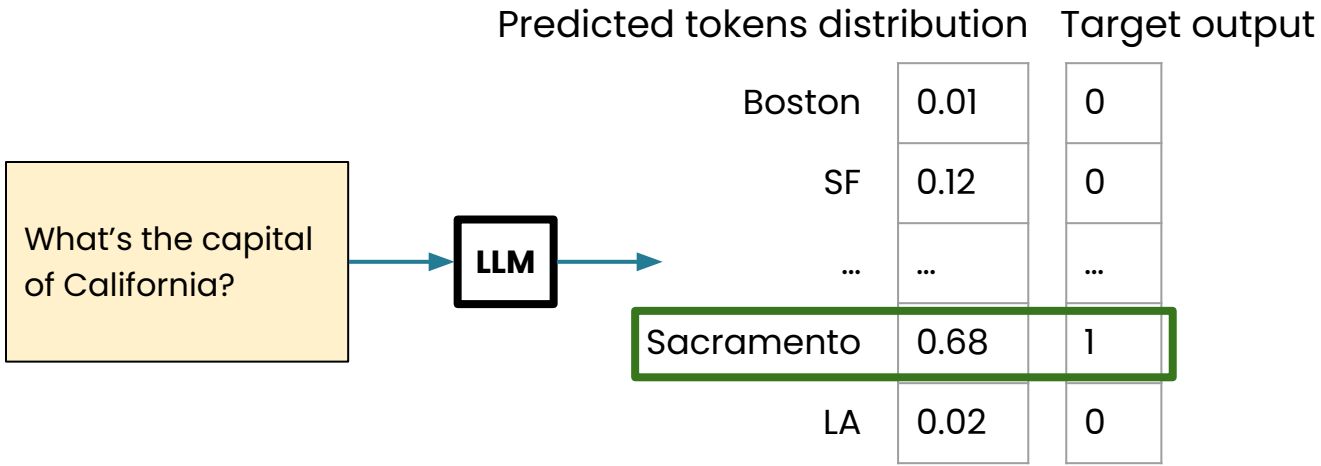
Token loss = -log(0.68)

Calculating loss only on the model outputs



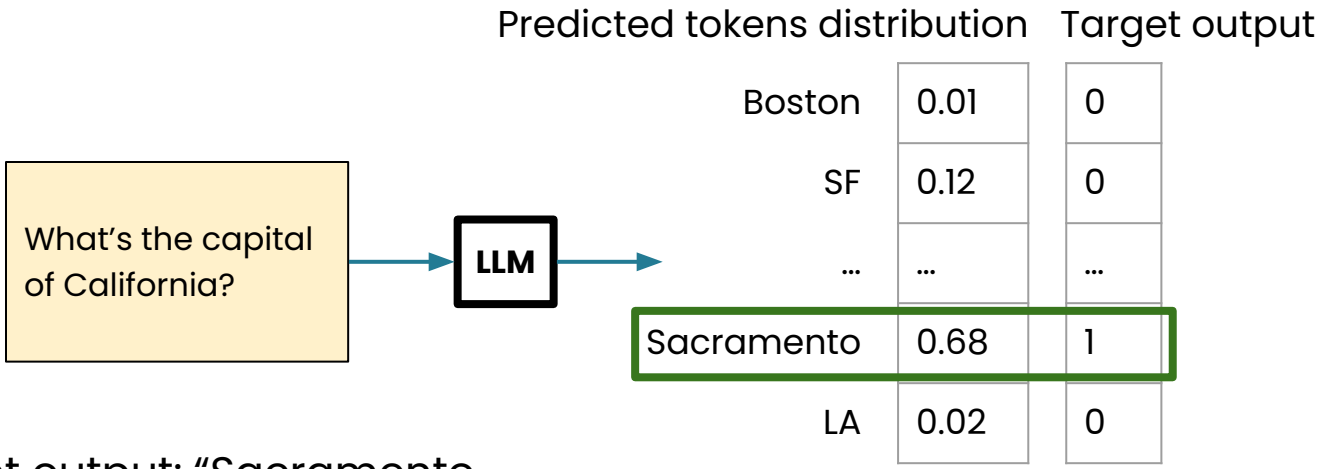
Token loss ~= 0.39

Calculating loss only on the model outputs



Token loss ~= 0.39 << 2.12

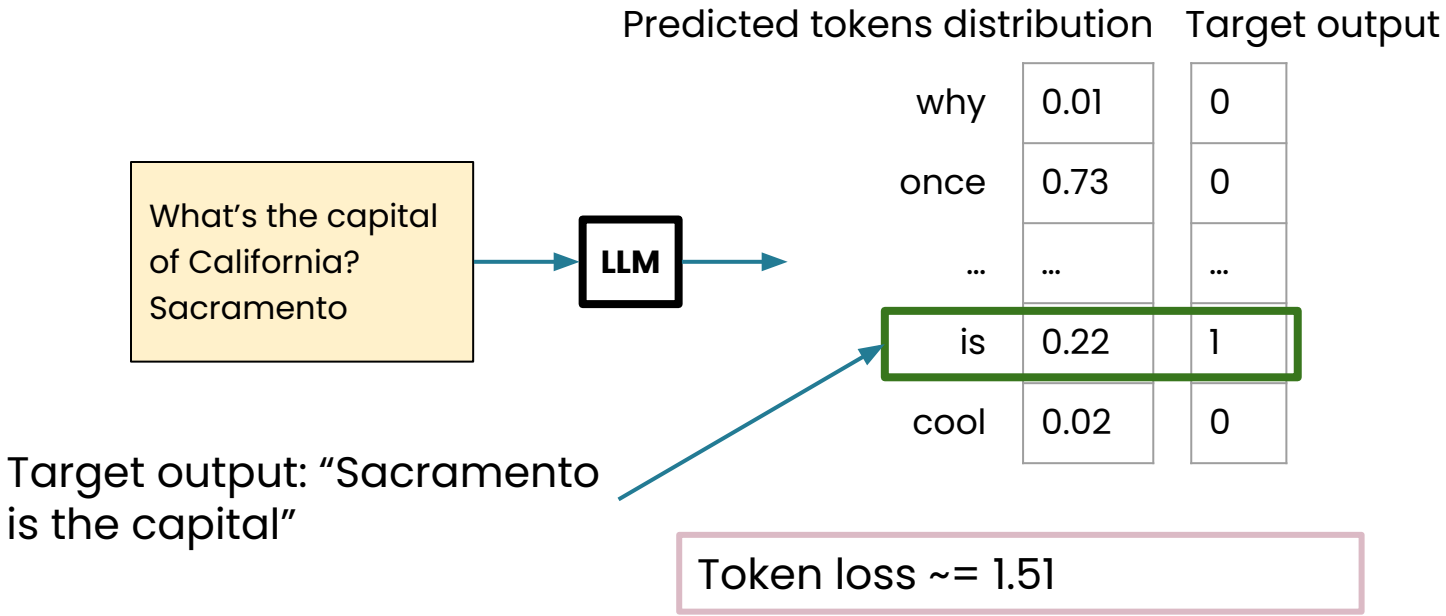
Calculating loss only on the model outputs



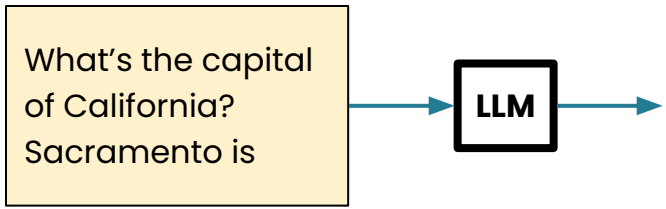
Target output: "Sacramento is the capital"

Token loss ~= 0.39 << 2.12

Calculating loss only on the model outputs



Calculating loss only on the model outputs

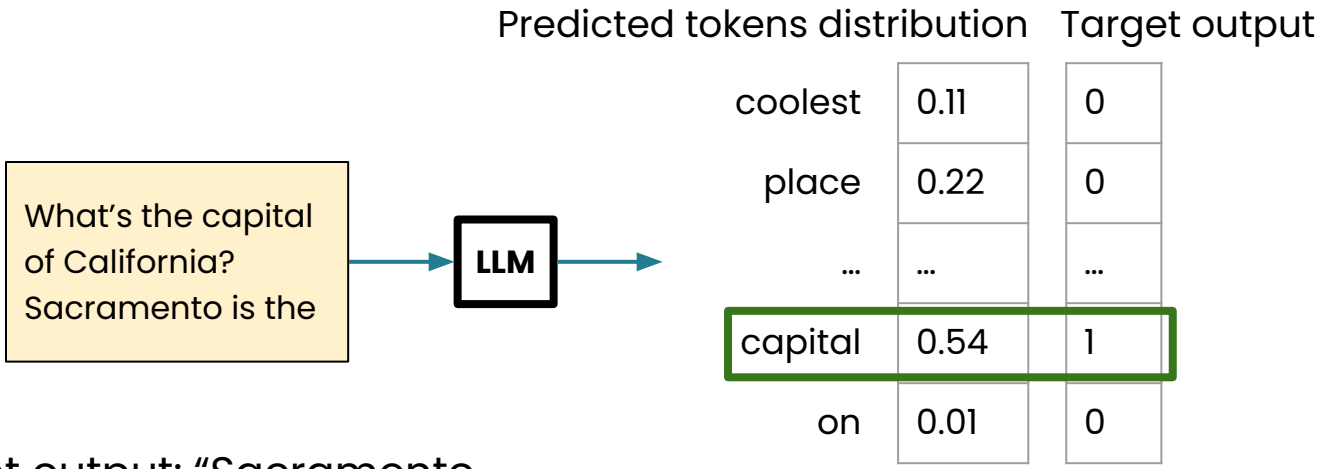


	Predicted tokens distribution	Target output
red	0.12	0
a	0.04	0
...	...	...
the	0.72	1
hot	0.04	0

Target output: "Sacramento is the capital"

Token loss ~= 0.33

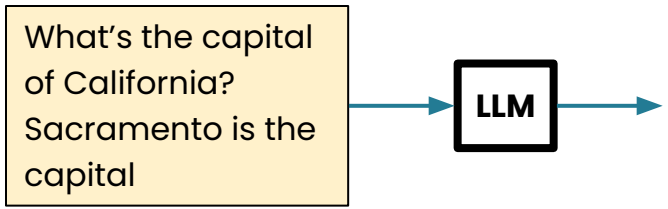
Calculating loss only on the model outputs



Target output: "Sacramento is the capital"

Token loss ~= 0.62

Calculating loss only on the model outputs

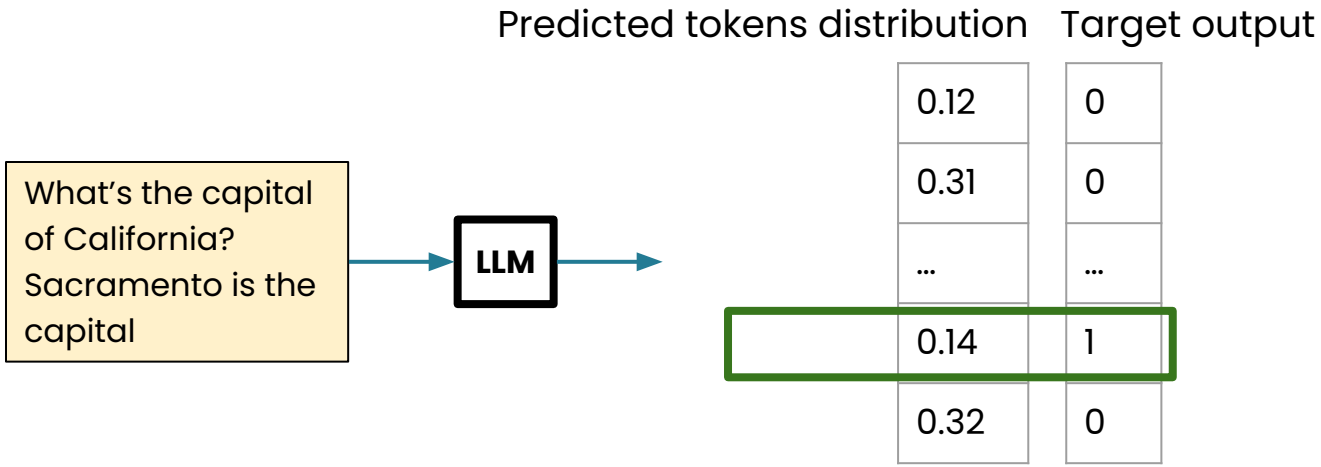


	Predicted tokens distribution	Target output
city	0.12	0
.	0.31	0
...	...	...
<STOP>	0.14	1
of	0.32	0

Target output: "Sacramento is the capital"

Token loss ~= 1.97

Calculating loss only on the model outputs



Cross Entropy (CE) Loss sums over all output token losses

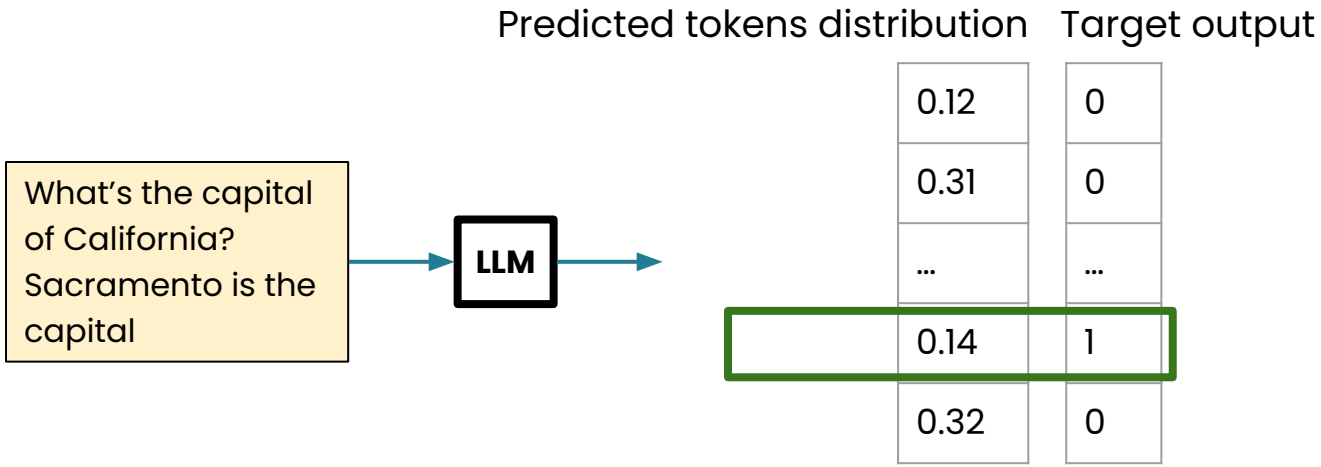
Loss ~= 0.39 + 1.51 + 0.33 + 0.62 + 1.97

DeepLearning.AI

Core Techniques in Fine-tuning and RL

Fine-tuning math: Loss, gradients, weight updates (Part 2)

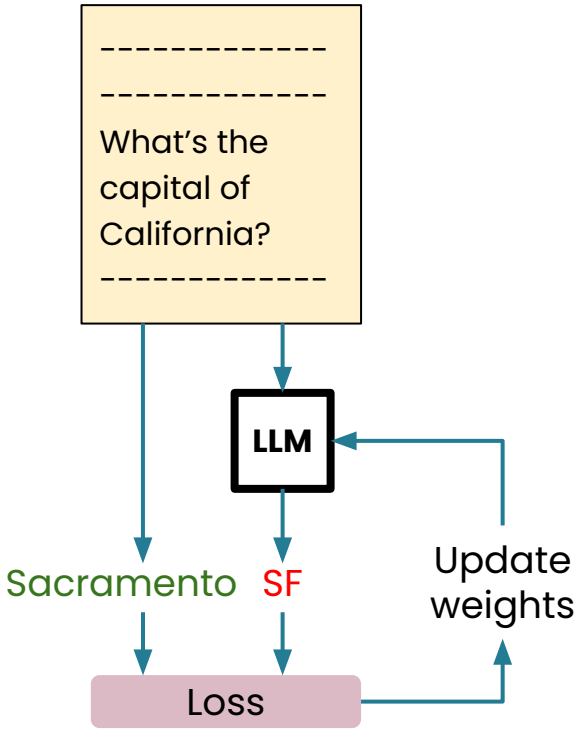
Calculating loss only on the model outputs



Goal: update the weights to minimize this term.

Loss ~= 0.39 + 1.51 + 0.33 + 0.62 + 1.97

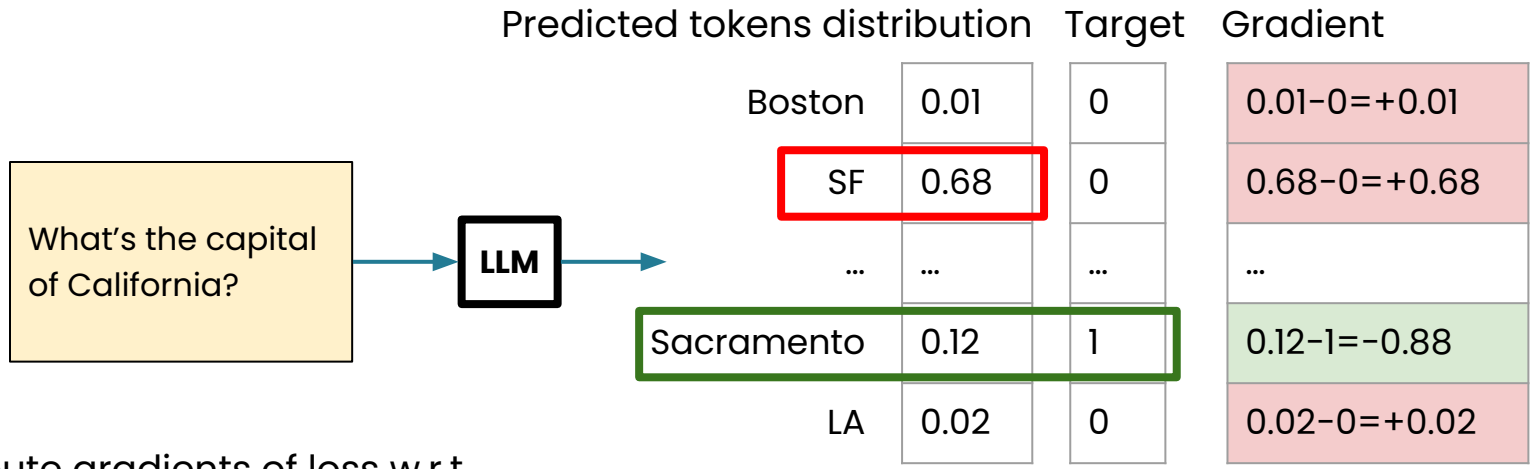
Backpropagation computes gradients



Similar steps

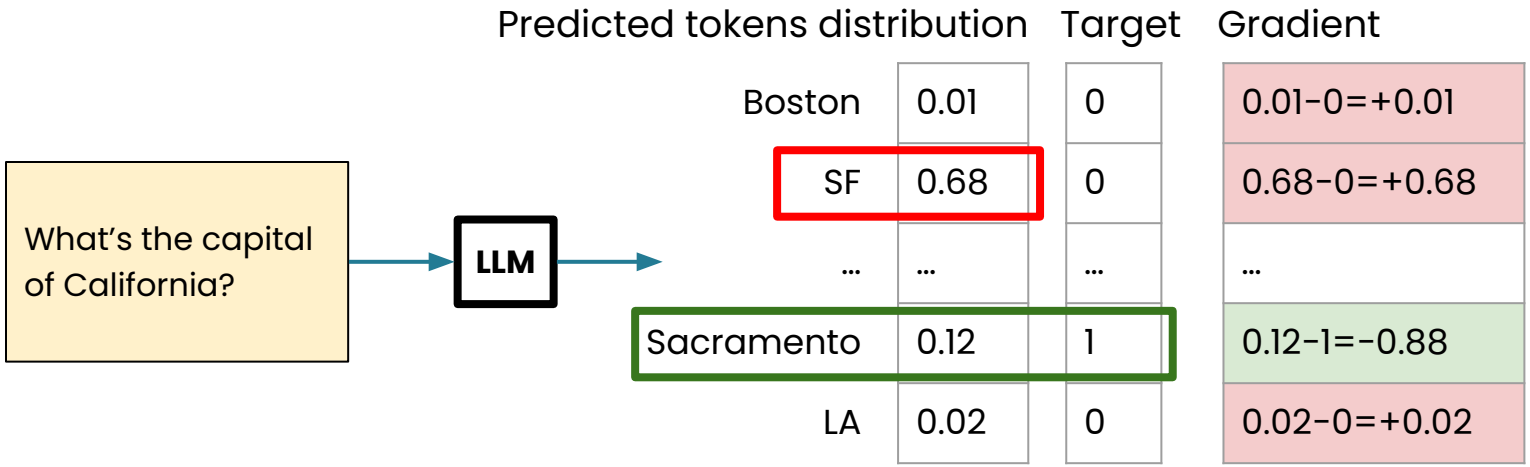
- Add training data (input-output pairs)
- Calculate loss on the outputs
- Backprop through model
- Update weights

Backpropagation computes gradients



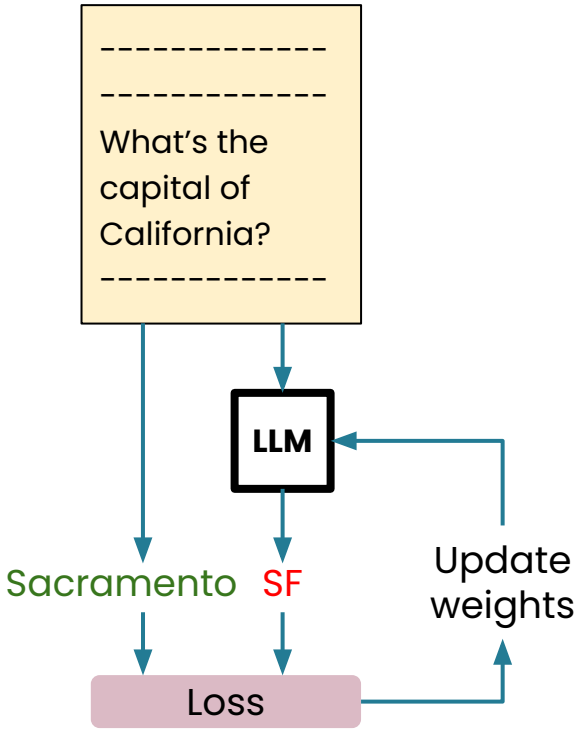
Compute gradients of loss w.r.t. every weight.

Backpropagation computes gradients



Gradient per token = predicted prob - target
(Derivative from CE loss)

Update weights with your gradients



Similar steps

- Add training data (input-output pairs)
- Calculate loss on the outputs
- Backprop through model
- Update weights

Update weights with your gradients

	Output gradients
Boston	0.01-0=+0.01
SF	0.68-0=+0.68
...	...
Sacramento	0.12-1=-0.88
LA	0.02-0=+0.02

Update weights with your gradients

	Output gradients
Boston	0.01-0=+0.01
SF	0.68-0=+0.68
...	...
Sacramento	0.12-1=-0.88
LA	0.02-0=+0.02

Hidden state

$$\begin{bmatrix} 0.5 \\ -1.0 \end{bmatrix}$$

Update weights with your gradients

	Output gradients	Gradient w.r.t. weights
Boston	0.01-0=+0.01	(+0.01)[0.5,-1.0]=[+0.005,-0.01]
SF	0.68-0=+0.68	(+0.68)[0.5,-1.0]=[+0.34,-0.68]
...	...	...
Sacramento	0.12-1=-0.88	(-0.88)[0.5,-1.0]=[-0.44,0.88]
LA	0.02-0=+0.02	(+0.02)[0.5,-1.0]=[+0.01,-0.02]

Hidden state

$$\begin{bmatrix} 0.5 \\ -1.0 \end{bmatrix}$$

Update weights with your gradients

	Output gradients	Gradient w.r.t. weights	Weight update w/ optimizer
Boston	0.01-0=+0.01	(+0.01)[0.5,-1.0]=[+0.005,-0.01]	...
SF	0.68-0=+0.68	(+0.68)[0.5,-1.0]=[+0.34,-0.68]	...
...	...	...	...
Sacramento	0.12-1=-0.88	(-0.88)[0.5,-1.0]=[-0.44,0.88]	$W_{row} \leftarrow W_{row} - 0.1[-0.44,+0.88]$
LA	0.02-0=+0.02	(+0.02)[0.5,-1.0]=[+0.01,-0.02]	...

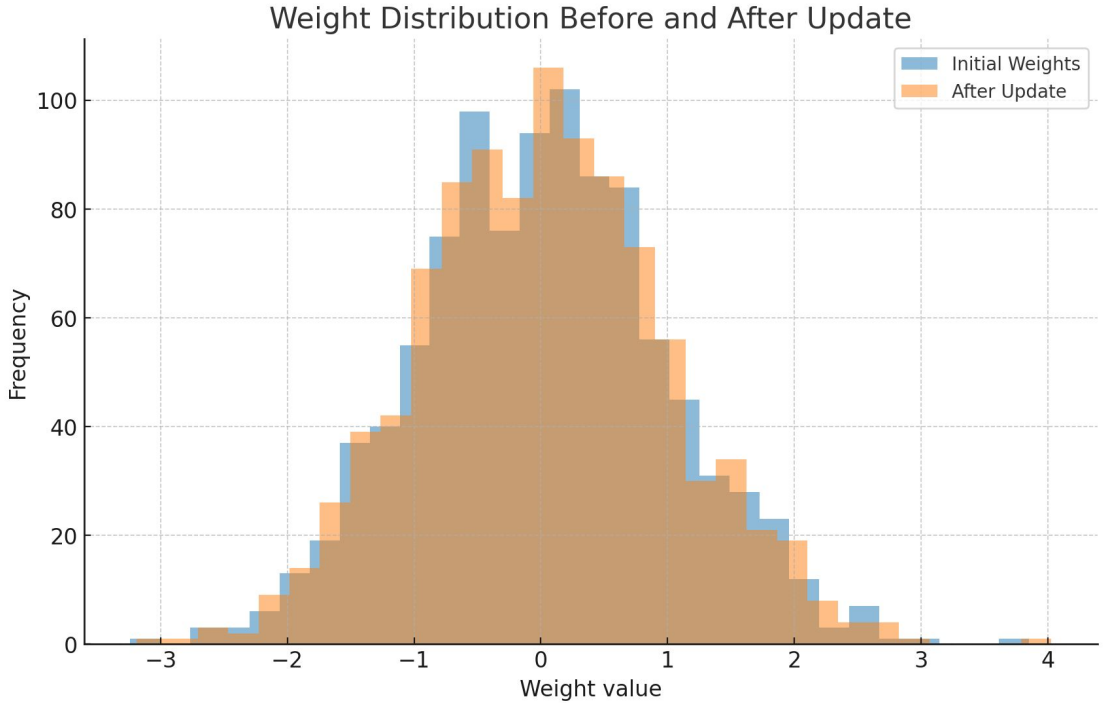
Hidden state

$$\begin{bmatrix} 0.5 \\ -1.0 \end{bmatrix}$$

$\eta = 0.1$

Optimizer: algorithm that use gradients to update weights.

Weight changes - visualized!



In your code

```
trainer = SFTTrainer(
    model=model_name,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
)
```

SFT stands for Supervised Fine-Tuning, which is the type of fine-tuning you're doing

In your code

```
training_args = SFTConfig(completion_only_loss=True, ...)

trainer = SFTTrainer(
    model=model_name,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
)

trainer.train()
```

Parameter to calculate loss only on the outputs (aka. “completions”)

In your code

```
training_args = SFTConfig(completion_only_loss=True, ...)

trainer = SFTTrainer(
    model=model_name,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
)

trainer.train()
```

Take a look inside...

Dropping down to pytorch

```
for epoch in range(num_epochs):
    for batch in train_dataloader:
        outputs = model(**batch)
        loss = outputs.loss
        loss.backward()
        optimizer.step()
```

Dropping down to pytorch

```
for epoch in range(num_epochs):
    for batch in train_dataloader:
        outputs = model(**batch)
        loss = outputs.loss
        loss.backward()
        optimizer.step()
```

For each training epoch

Dropping down to pytorch

```
for epoch in range(num_epochs):
    for batch in train_dataloader:
        outputs = model(**batch)
        loss = outputs.loss
        loss.backward()
        optimizer.step()
```

For each training epoch

For each batch of {input, target output} pairs

Dropping down to pytorch

```
for epoch in range(num_epochs):
    for batch in train_dataloader:
        outputs = model(**batch)
        loss = outputs.loss
        loss.backward()
        optimizer.step()
```

For each training epoch

For each batch of {input, target output} pairs

1. Model predicts outputs

Dropping down to pytorch

```
for epoch in range(num_epochs):
    for batch in train_dataloader:
        outputs = model(**batch)
        loss = outputs.loss
        loss.backward()
        optimizer.step()
```

For each training epoch

For each batch of {input, target output} pairs

1. Model predicts outputs
2. Calculate loss on the outputs (vs. target outputs)

Dropping down to pytorch

```
for epoch in range(num_epochs):
    for batch in train_dataloader:
        outputs = model(**batch)
        loss = outputs.loss
        loss.backward()
        optimizer.step()
```

For each training epoch

For each batch of {input, target output} pairs

1. Model predicts outputs
2. Calculate loss on the outputs (vs. target outputs)
3. Backprop computes the gradients

Dropping down to pytorch

```
for epoch in range(num_epochs):
    for batch in train_dataloader:
        outputs = model(**batch)
        loss = outputs.loss
        loss.backward()
        optimizer.step()
```

For each training epoch

For each batch of {input, target output} pairs

1. Model predicts outputs
2. Calculate loss on the outputs (vs. target outputs)
3. Backprop computes the gradients
4. Update weights using gradients

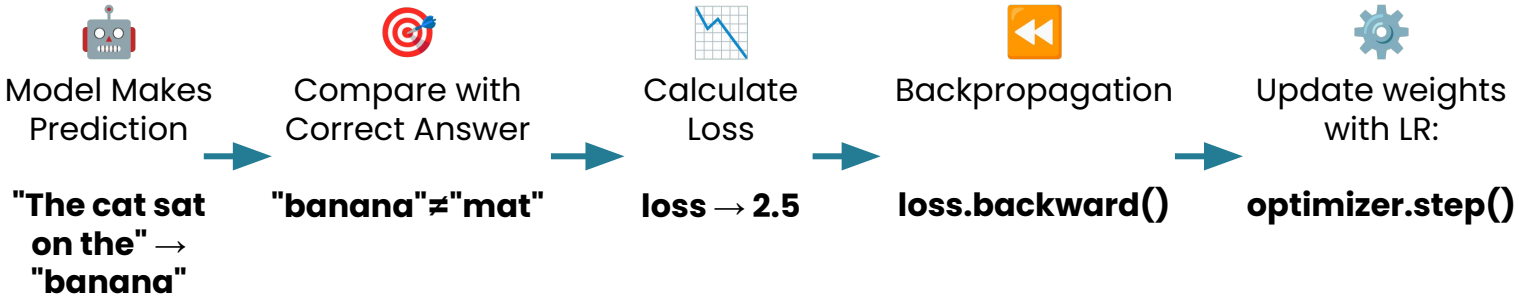
Optimizer: algorithm that use gradients to update weights.

DeepLearning.AI

Core Techniques in Fine-tuning and RL

Fine-tuning: Hyperparameters & hyperparameter tuning (Part 1)

Learning rate



How to choose a good LR?

High LR
Model "forgets" what it learned, **oscillates** wildly

Good LR
Model **steadily** improves, learns efficiently

Low LR
Model barely learns from mistakes, very **slow** progress

Learning rate

Learning rate

🌋Too High LR

Console Output

Loss: NaN
Loss: inf
RuntimeError

Learning rate

The cat sat on

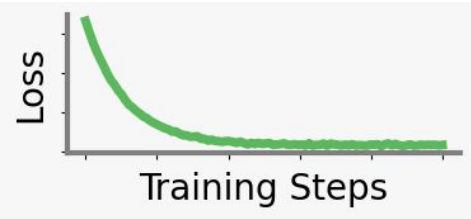
🌋Too High LR

*#&/jw2@

Console Output

Loss: NaN
Loss: inf
RuntimeError

✅Perfect LR



the warm blanket

Console Output

Epoch 1: 2.15
Epoch 2: 1.89
Epoch 3: 1.67

Learning rate

The cat sat on

🌋Too High LR

*#&/jw2@

Console Output
Loss: NaN
Loss: inf
RuntimeError

✅Perfect LR

the warm blanket

Console Output
Epoch 1: 2.15
Epoch 2: 1.89
Epoch 3: 1.67

🐌Too Low LR

chair window door

Console Output
Epoch 1: 2.15
Epoch 2: 2.14
Epoch 3: 2.14

Learning rate

Learning rate (LR) schedulers

Cosine Annealing
Smooth decay, good final performance

Linear Decay with Warmup
Simple, predictable, widely used

Learning rate

Learning rate (lr) defaults in frameworks

Transformers
Default optimizer

AdamW: lr=5e-5

PyTorch
Popular optimizers

SGD: lr=0.01
Adam: lr=0.001

Number of epochs

Your Training Dataset

"The cat sat on the mat"
"I love sunny days"
"Today is Monday"
"Dogs are loyal pets"
... 1000 total examples

During 1 Epoch

Step 1 Process "The cat sat on the mat"
Step 2 Process "I love sunny days"
Step 3 Process "Today is Monday"
... keeps going until...
Final Process last example #1000

✅ 1 Epoch Complete = Model has seen every training example once

Number of epochs

Your Training Dataset

"The cat sat on the mat"
"I love sunny days"
"Today is Monday"
"Dogs are loyal pets"
... 1000 total examples

During 1 Epoch

Step 1 Process "The cat sat on the mat"
Step 2 Process "I love sunny days"
Step 3 Process "Today is Monday"
... keeps going until...
Final Process last example #1000

✅ 1 Epoch Complete = Model has seen every training example once

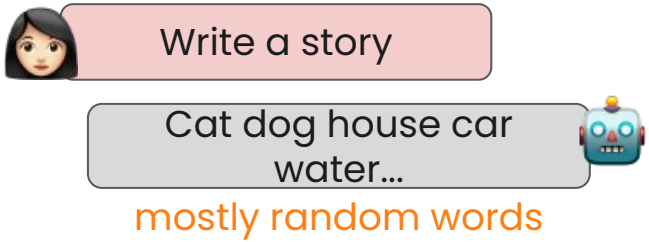
What Happens with Multiple Epochs?

Epoch 1: First time seeing all data

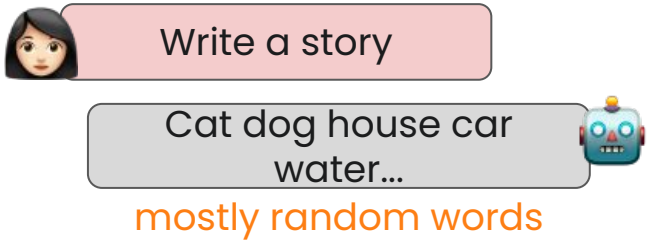
Epoch 2: Second time seeing same data

Epoch 3: Third time... gets better each time

Number of epochs



Number of epochs



Number of epochs

Write a story

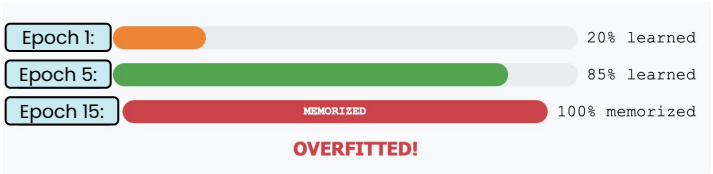
Cat dog house car water...

mostly random words

Write a story

Once upon a time, in a small village ...

coherent, creative text

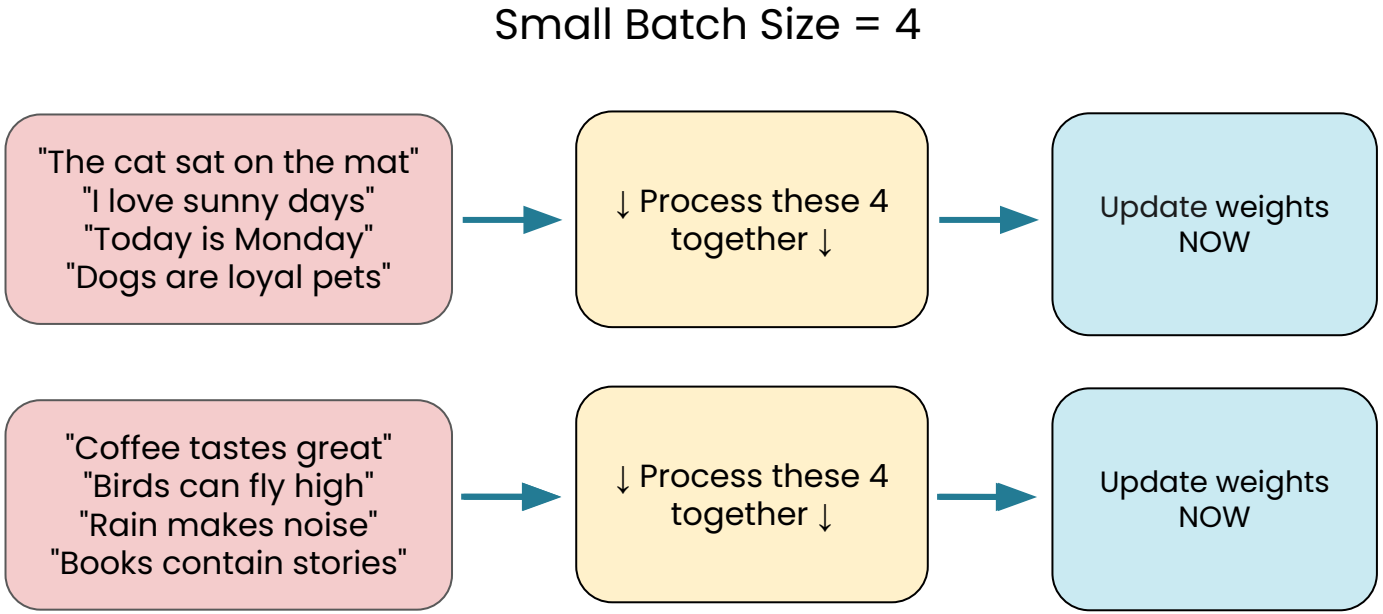


Write a story

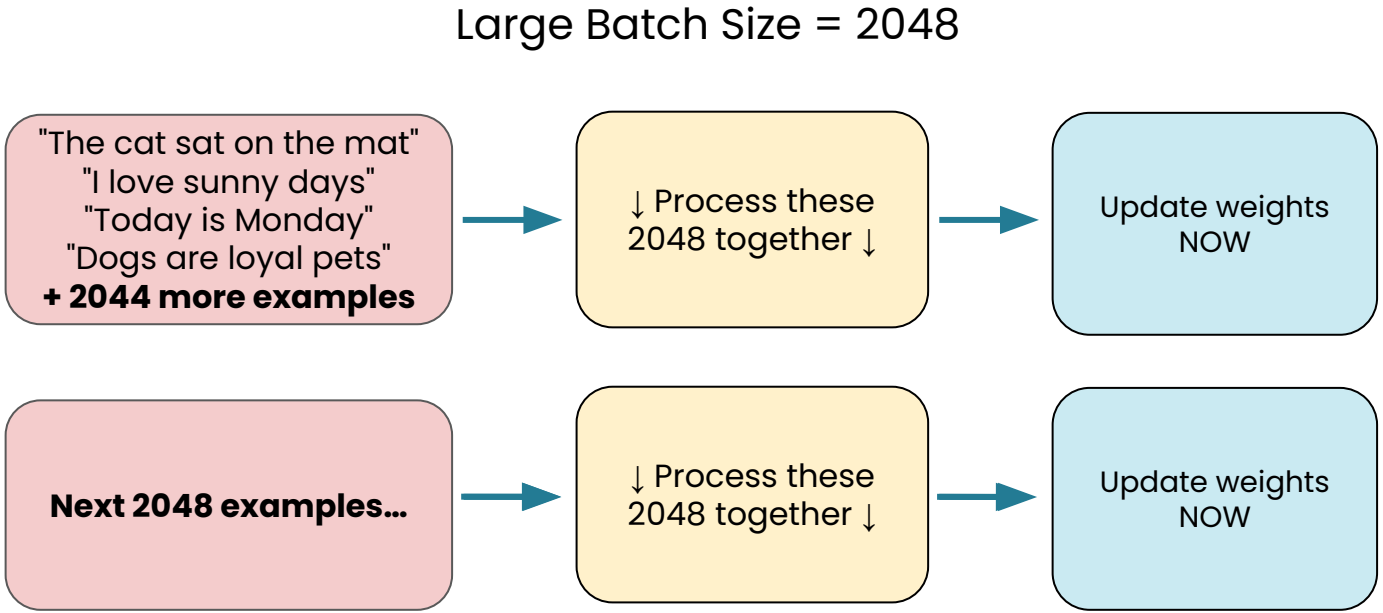
pne feuwe ndjdwxo

memorized training, fails on new text

Batch size


Small Batch Size = 4
"The cat sat on the mat"
"I love sunny days"
"Today is Monday"
"Dogs are loyal pets"
↓ Process these 4 together ↓
Update weights NOW
"Coffee tastes great"
"Birds can fly high"
"Rain makes noise"
"Books contain stories"
↓ Process these 4 together ↓
Update weights NOW
Many frequent weight updates

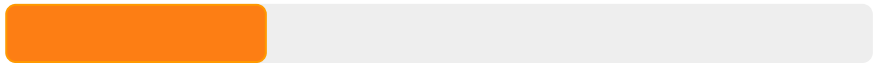
Batch size


Large Batch Size = 2048
"The cat sat on the mat"
"I love sunny days"
"Today is Monday"
"Dogs are loyal pets"
+ 2044 more examples
↓ Process these 2048 together ↓
Update weights NOW
Next 2048 examples...
↓ Process these 2048 together ↓
Update weights NOW
Fewer, more stable updates

Batch size

Small Batch (16)

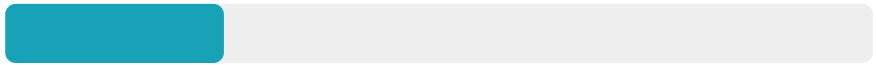
1. Training Speed



Slower

Many small updates = more time per epoch

2. GPU Memory



Few GB used

3. Console Output

```
Batch 1/500: Loss 2.15
Batch 2/500: Loss 2.12
Batch 3/500: Loss 2.18
Training: 45 min/epoch
```

Good for: Limited GPU memory, fine-tuning

Batch size

Large Batch (512)

1. Training Speed

Faster

Fewer, larger updates = less time per epoch

2. GPU Memory

Medium GB used

3. Console Output

Batch 1/31: Loss 2.15
Batch 2/31: Loss 2.11
Batch 3/31: Loss 2.09
Training: 15 min/epoch

Good for: Large datasets

Batch size

Too Large (2048)

1. Training Speed

Crashed

No training - out of memory

2. GPU Memory

Too many GB used

3. Console Output

Starting training...
RuntimeError: Out of memory!
Tried to allocate 24.50 GiB
Training terminated

Fix: Reduce batch size

In your code

```
training_args = TrainingArguments(
        learning_rate=2e-6,
        weight_decay=0.01,
        per_device_train_batch_size=20,
        per_device_eval_batch_size=20,
        num_train_epochs=3,
)
```

DeepLearning.AI

Core Techniques in Fine-tuning and RL

Fine-tuning: Hyperparameters & hyperparameter tuning (Part 2)

Hyperparameter tuning

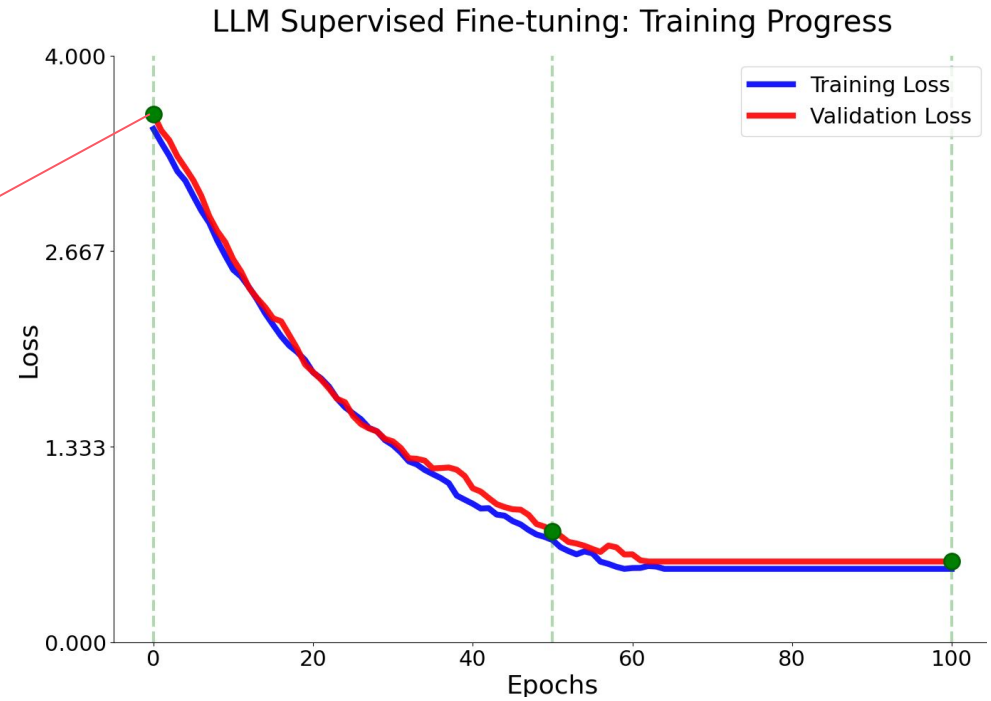
Testing held out examples

Input

Explain quantum computing briefly

Output

Quantum computing is computers quantum mechanics use.



Testing held out examples

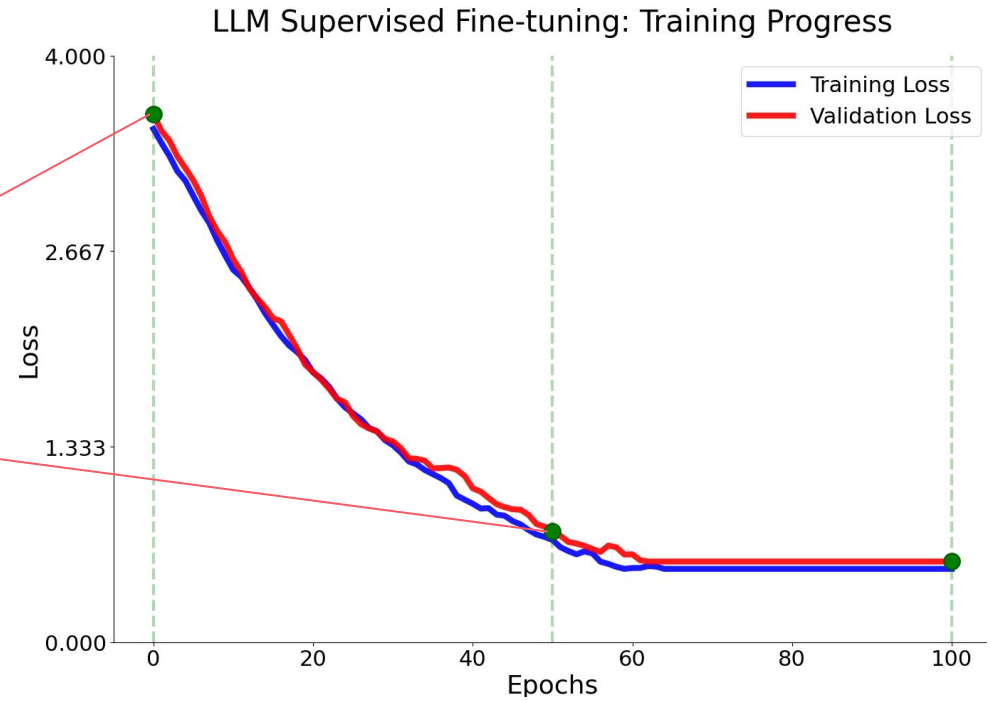
Input

Explain quantum computing briefly

Output

Quantum computing is computers quantum mechanics use.

Quantum computing uses quantum mechanics principles like superposition for computation.



Testing held out examples

Input

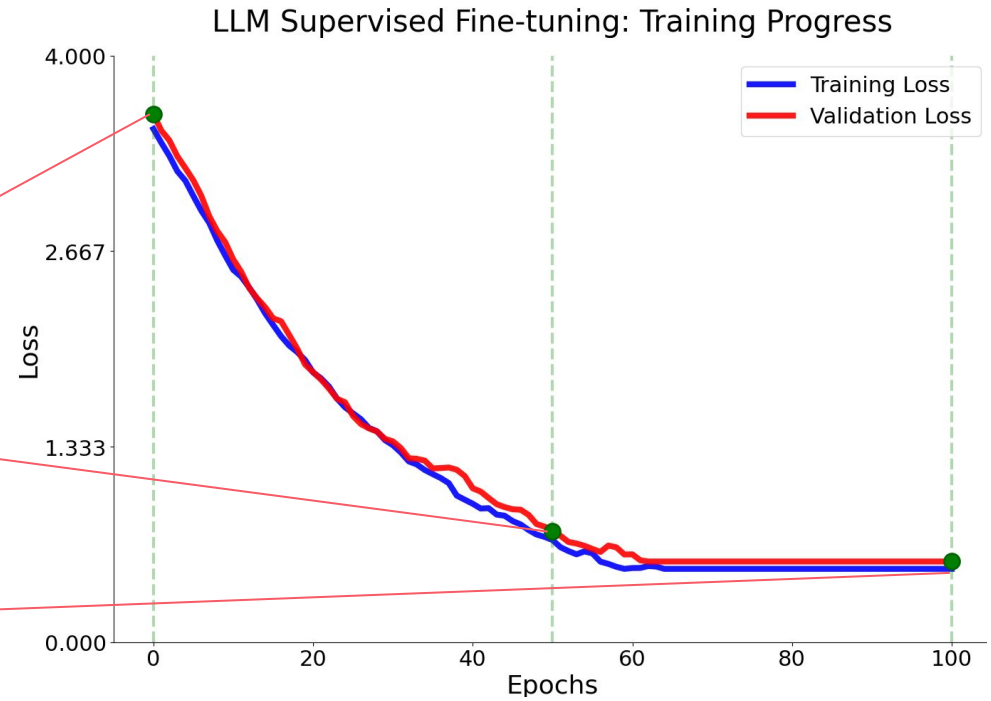
Explain quantum computing briefly

Output

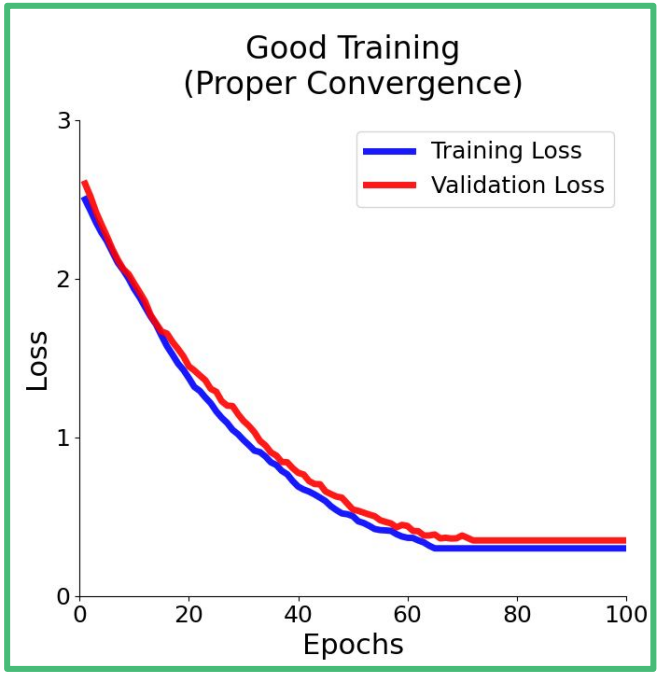
Quantum computing is computers quantum mechanics use.

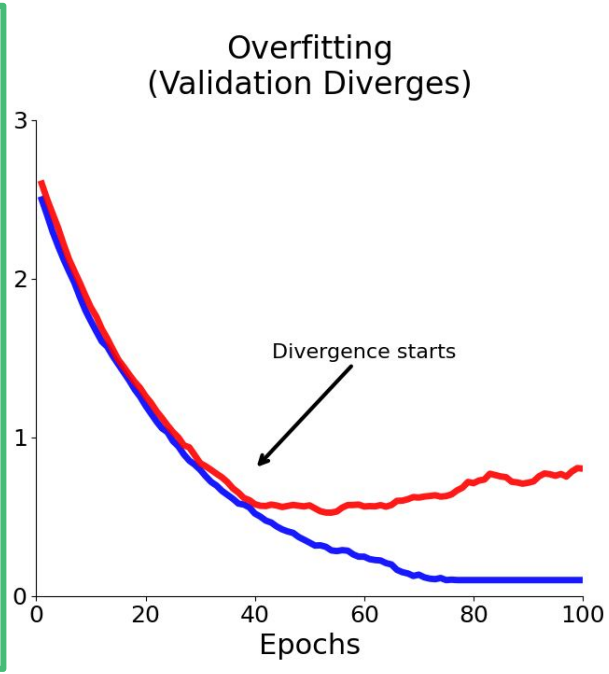
Quantum computing uses quantum mechanics principles like superposition for computation.

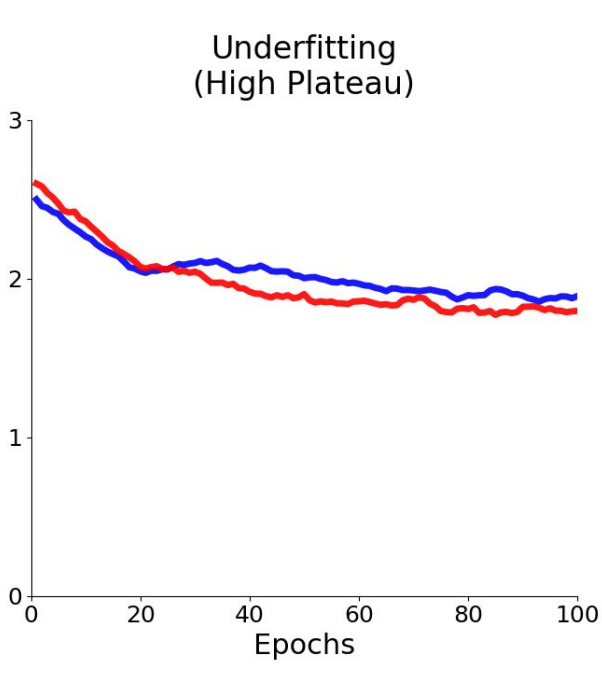
Quantum computing utilizes quantum mechanics principles, employing qubits that exist in superposition states.



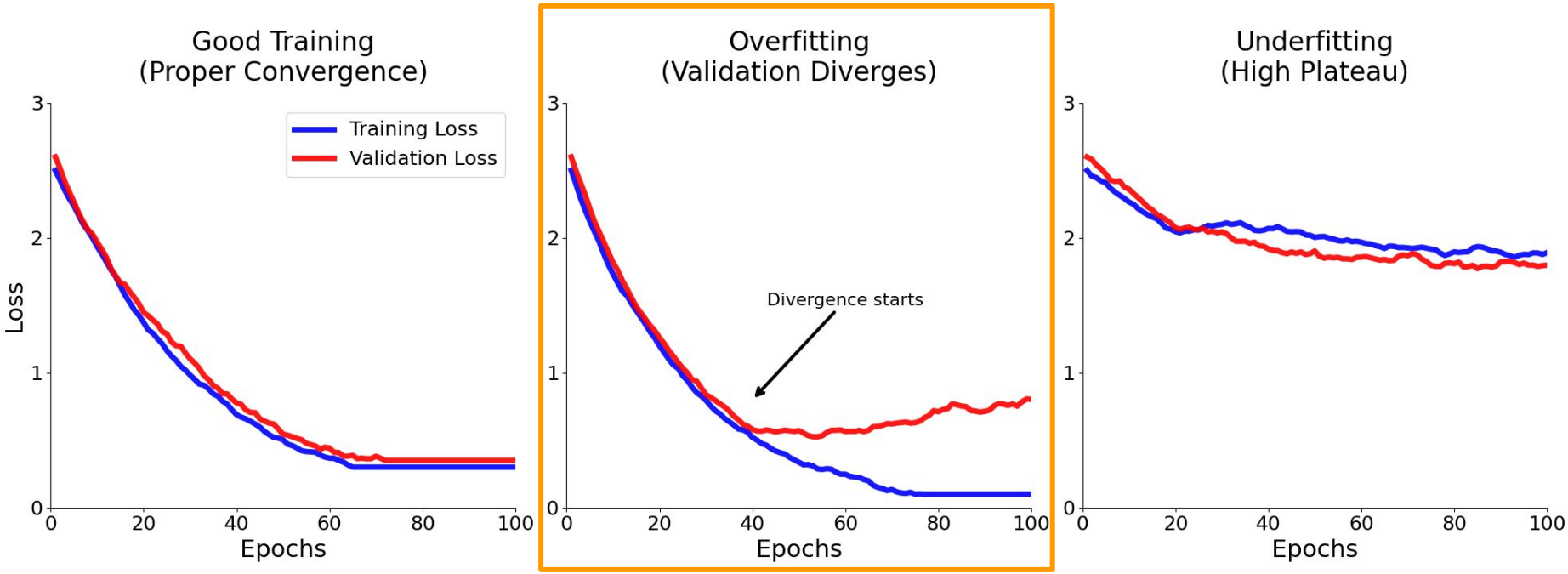
Training vs validation loss



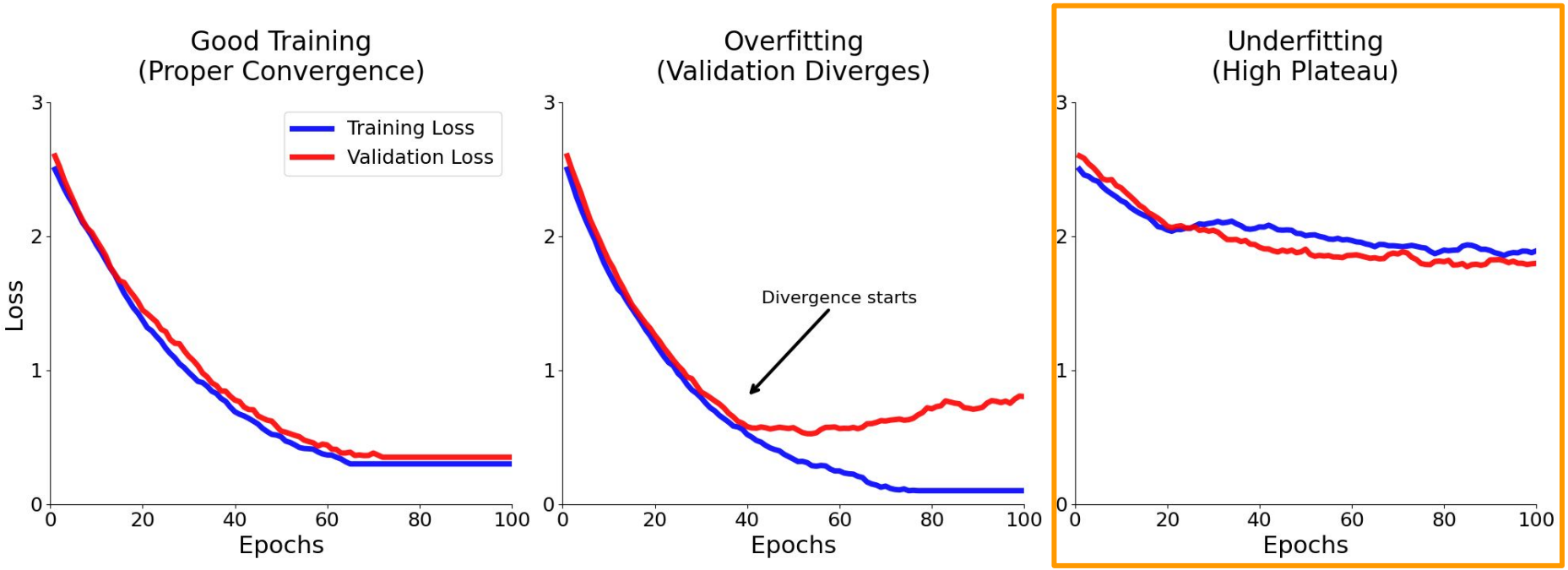




Training vs validation loss



Training vs validation loss



Plots in your lab

With a lot of experiments, reproducibility is important: use random seeds

Model 1

Accuracy 86%

Model 2

Accuracy 91%

Which is better?

With a lot of experiments, reproducibility is important: use random seeds

Which is better?

Without random seeds,
could just be random variation

With a lot of experiments, reproducibility is important: use random seeds

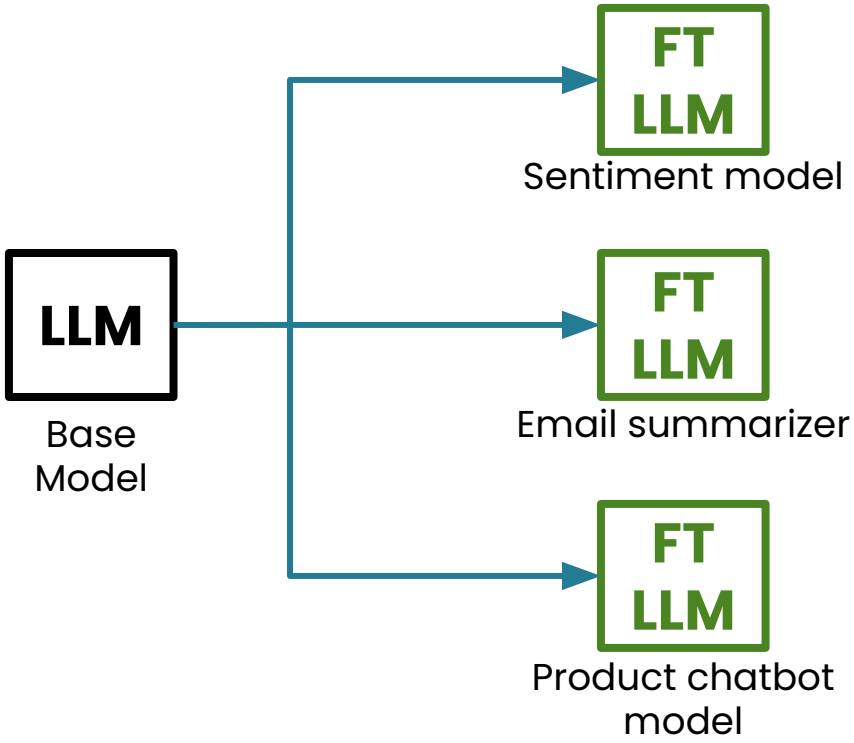

```
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
```

DeepLearning.AI

Core Techniques in Fine-tuning and RL

Fine-tuning: Parameter efficient fine-tuning (PEFT)

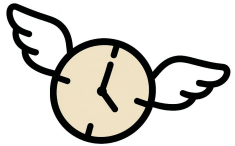
Fine-tuning your whole model is expensive



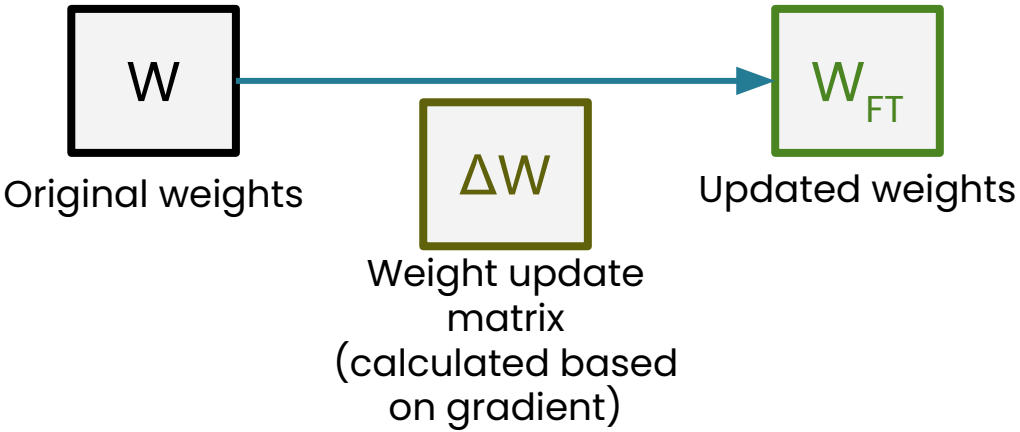
Lots of compute + memory to

- Change all model weights
- Serve different models

Is there a better way?



Weight updates in full fine-tuning have lots of redundancy

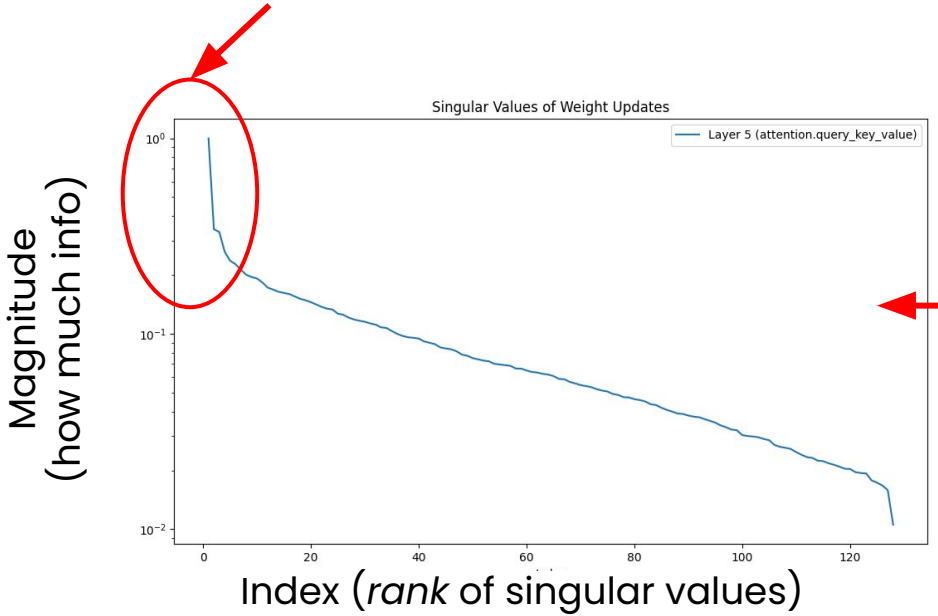


The gradient matrix ΔW has lots of redundancy!

ΔW is big, but a much smaller matrix could contain approximately all information

Empirical evidence ΔW could be smaller

Sharp drop-off → much lower rank matrix could capture most info



SVD

ΔW

Weight update matrix

Making it more efficient: An analogy

Multiple models

One model,
multiple adapters

Low-rank decomposition

A large, low-rank matrix can be ~2 smaller matrices. Rank = 2:

1000 x 1000

1000 x 1000 = 1,000,000

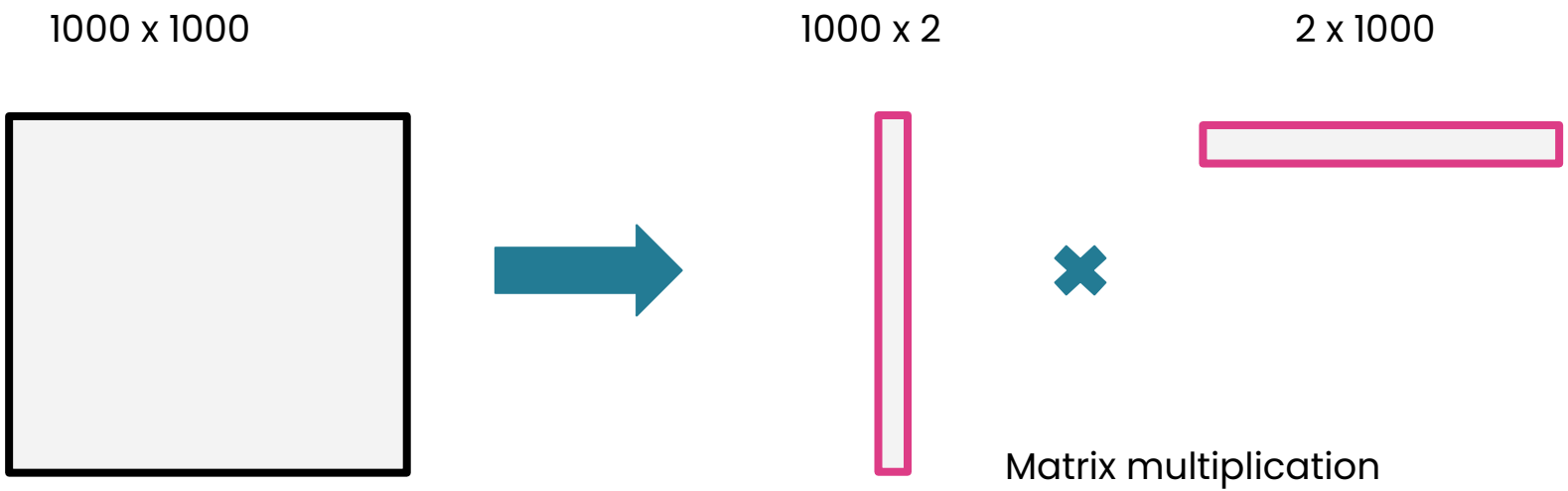
1000 x 2

2 x 1000

Matrix multiplication

1000 x 2 + 1000 x 2 = 4000

LLM matrices are huge - big savings!



1000 x 1000 = 1,000,000

1000 x 2 + 1000 x 2 = 4000

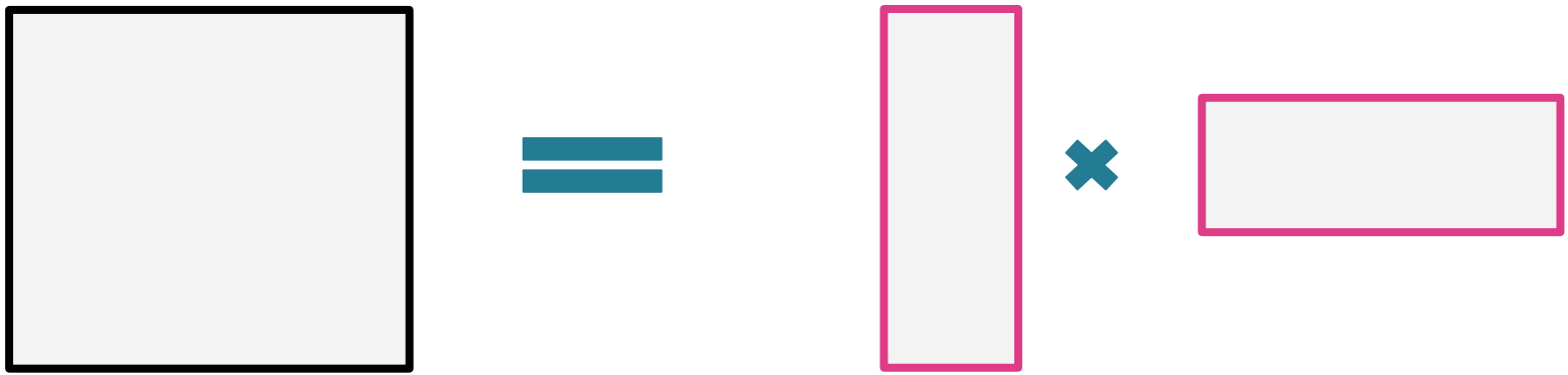
250X few parameters to train/store in memory

Let’s generalize this to rank r…

1000 x 1000

1000 x r

r x 1000



1000 x 1000 = 1,000,000

1000 x r + 1000 x r

Low rank: r << original matrix dimensions

What does this mean for fine-tuning?

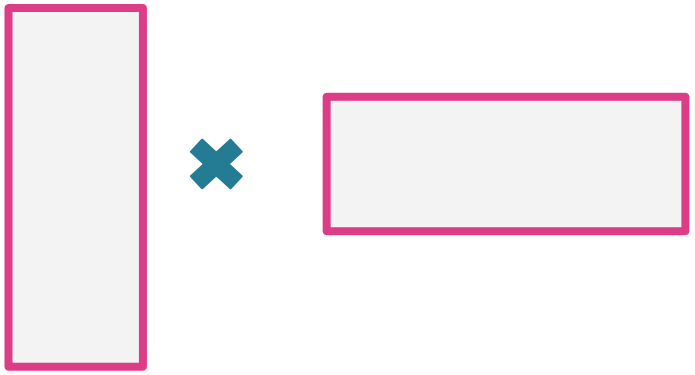
1000 x 1000

1000 x 1000 = 1,000,000

- Huge on disk (GB)
- Many params to update

1000 x r

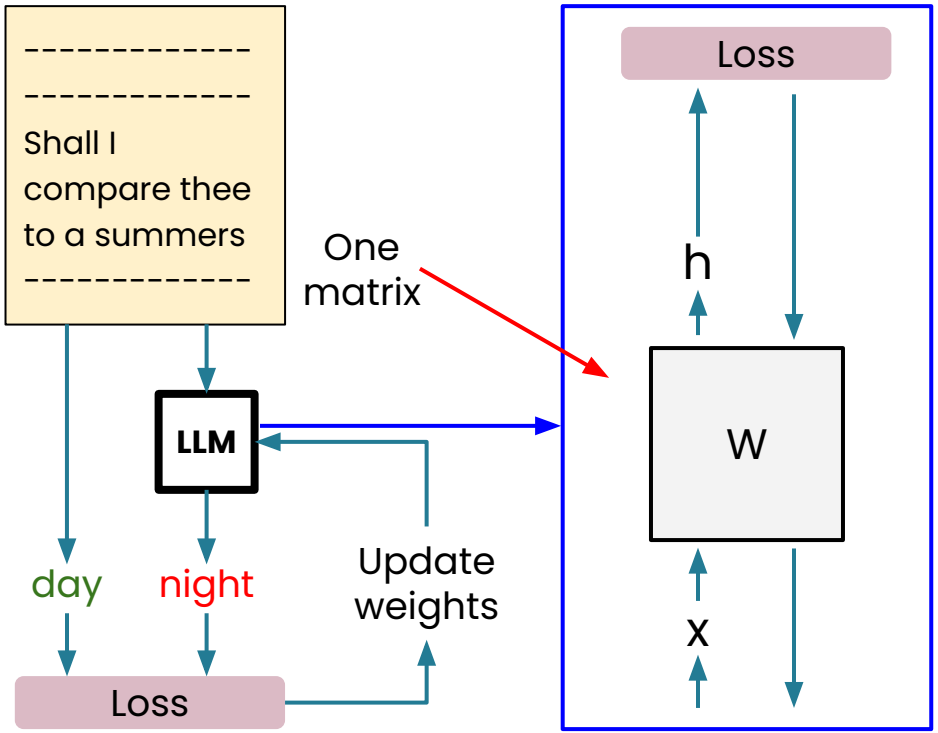
r x 1000



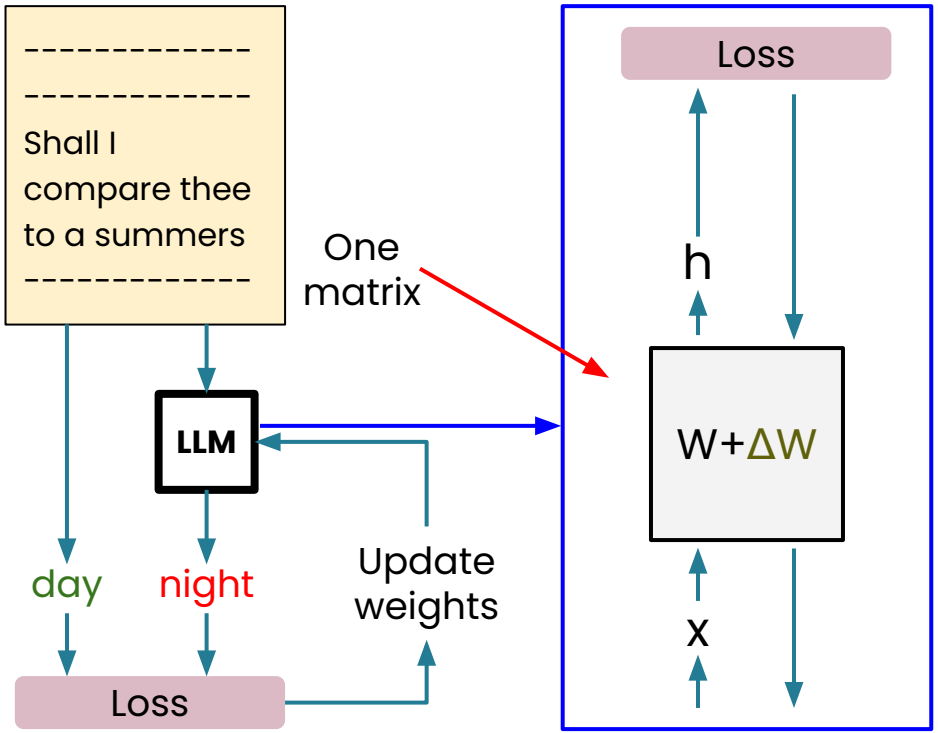
1000 x r + 1000 x r

- Small on disk (MB)
- Fewer params to update

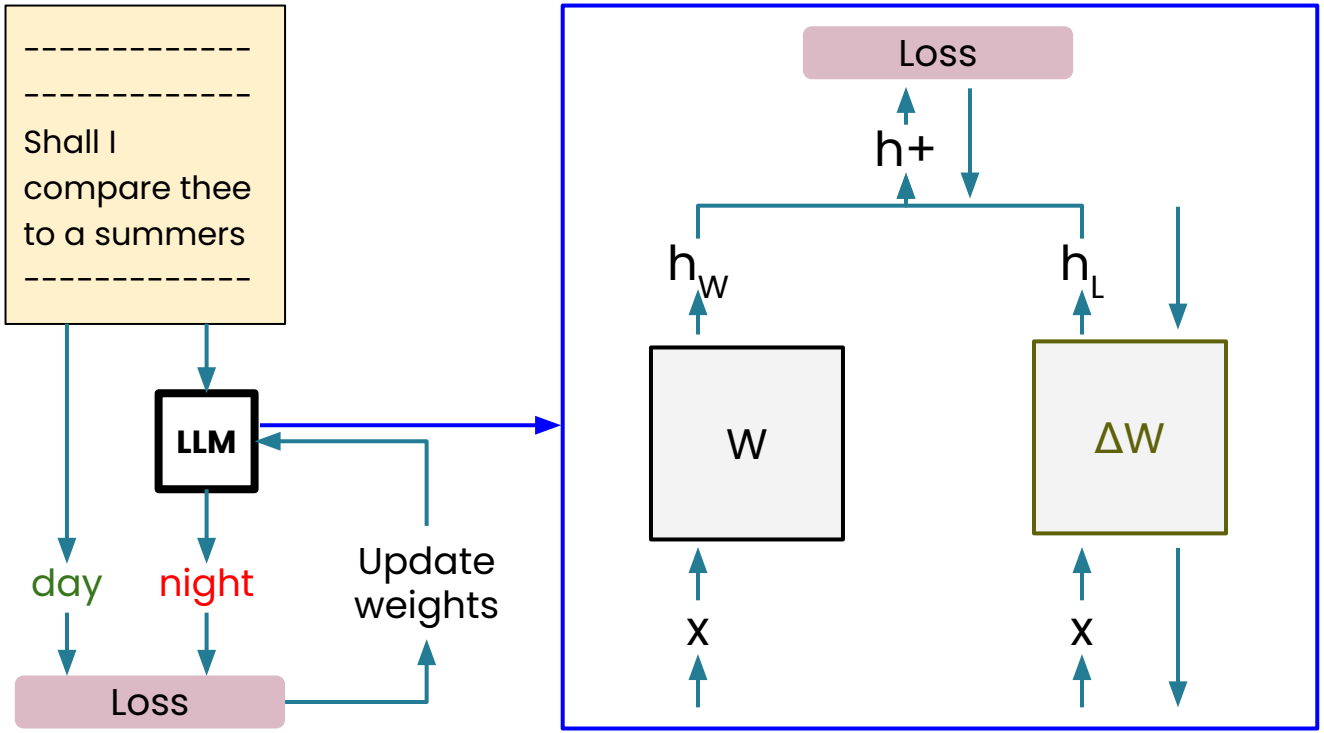
Weight updates during full fine-tuning



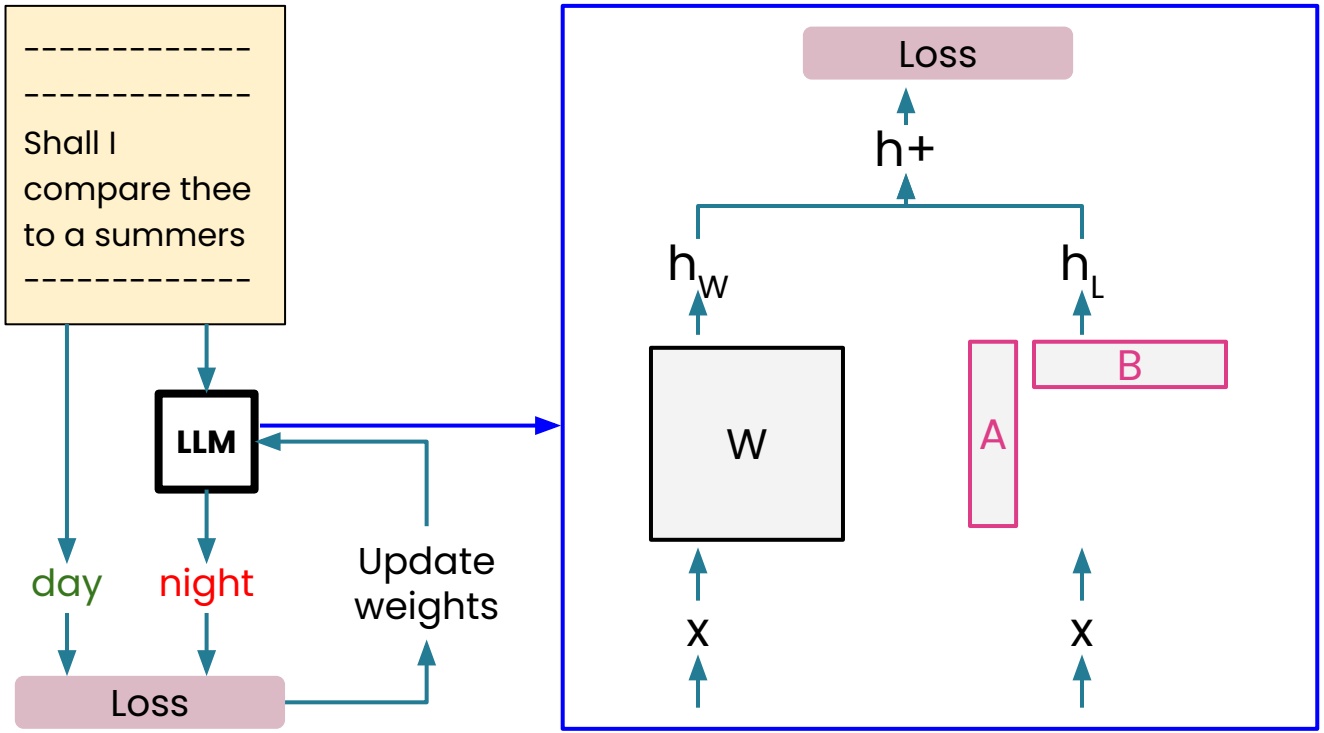
Weight updates during full fine-tuning



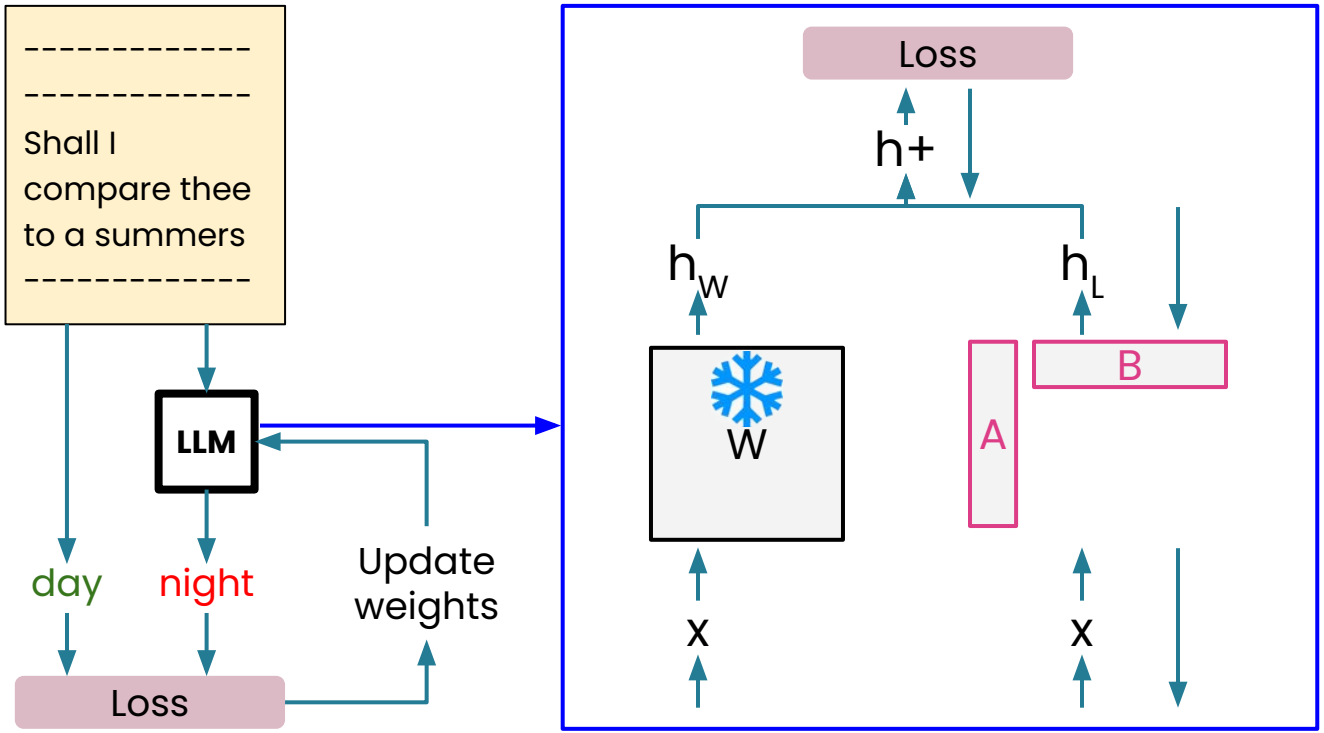
Weight updates during Low-Rank Adaption (LoRA) fine-tuning



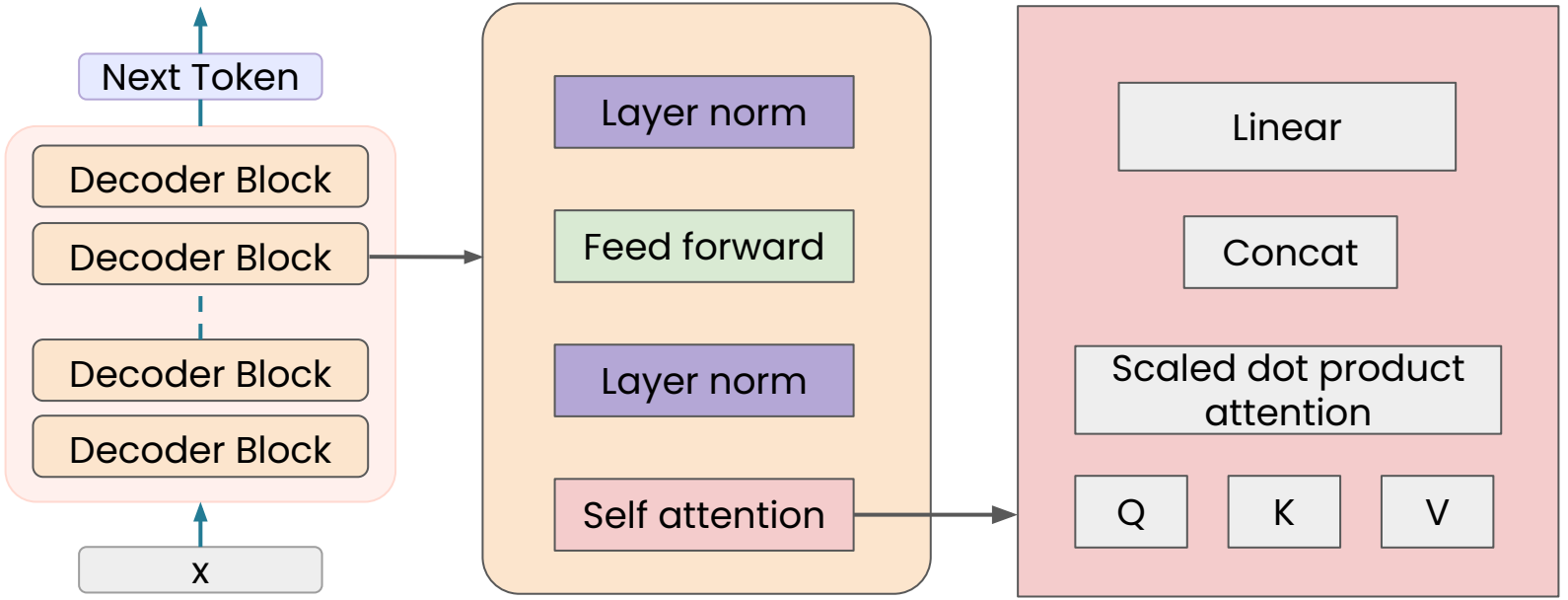
Weight updates during Low-Rank Adaption (LoRA) fine-tuning



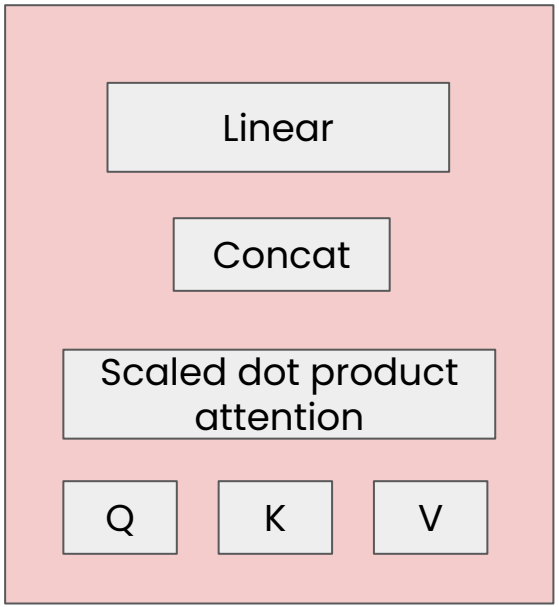
Weight updates during Low-Rank Adaption (LoRA) fine-tuning

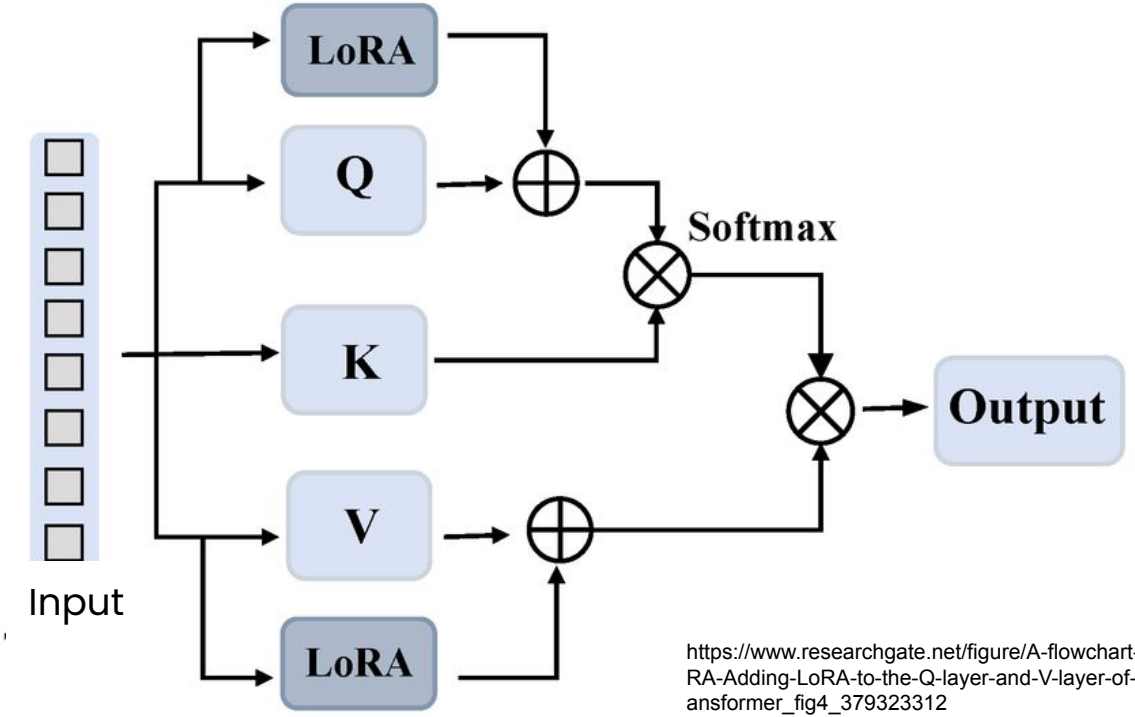


Which LLM matrices get LoRAs?



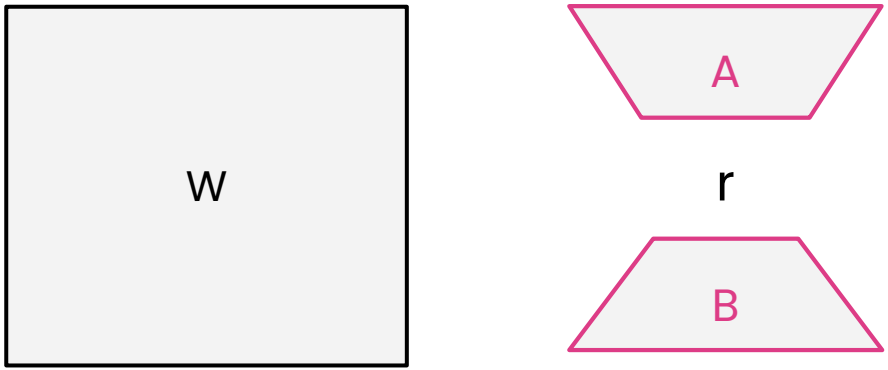
Which LLM matrices get LoRAs?





https://www.researchgate.net/figure/A-flowchart-of-LoRA-Adding-LoRA-to-the-Q-layer-and-V-layer-of-the-transformer_fig4_379323312

Often represented like this



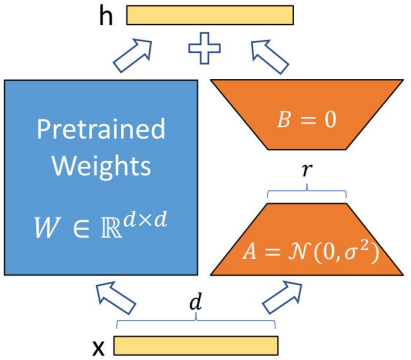
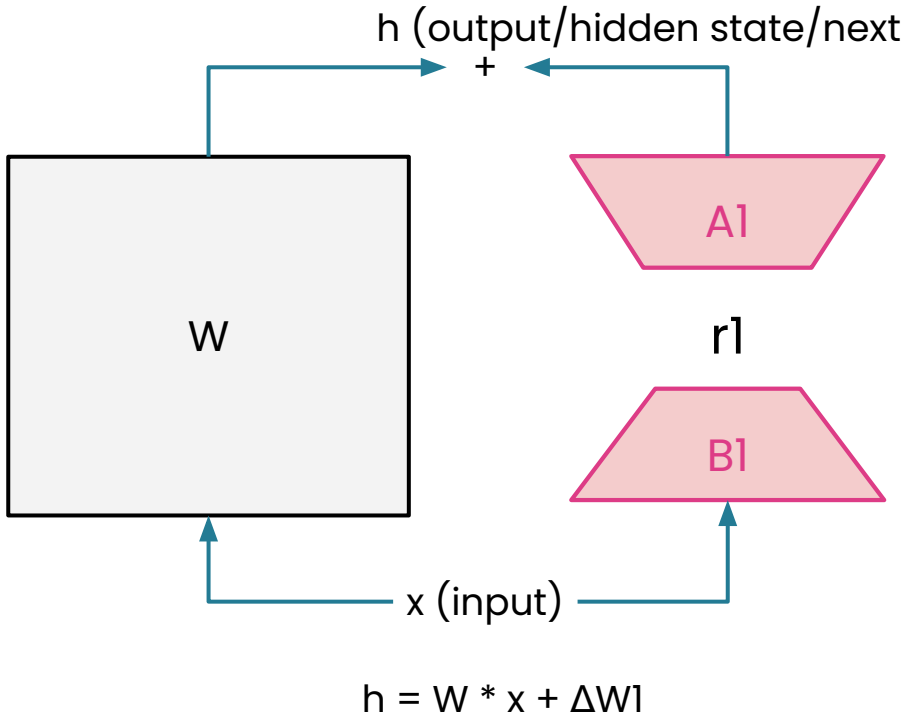


Figure 1: Our reparametrization. We only train A and B.

From Hu et al. 2021

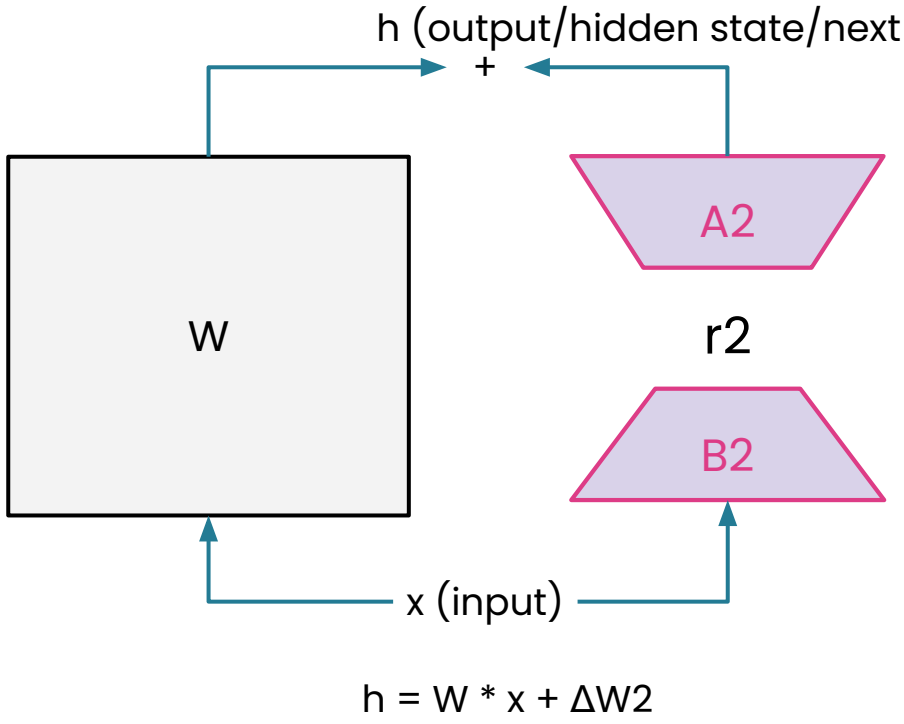
Portability: hot-swap different LoRAs



LoRA with task 1

Fine-tuned on custom code generation

Portability: hot-swap different LoRAs



LoRA with task 2

Fine-tuned on custom unit tests

Mo’ hyperparams

α scales how much LoRAs vs. the original W weights matter

$$\mathbf{W}_{FT} = \mathbf{W} + \frac{\alpha}{r}\Delta\mathbf{W}$$

Less memory usage = lowers GPU barriers!

Total Memory Usage (PyTorch) - Apply LoRA

LLM Fine-tuned

Total_memory = model_memory

LLM Fine-tuned (with LoRA)

Total_memory(Apply LoRA) = model_memory

Reference: https://xiaosean5408.medium.com/fine-tuning-LLMs-made-easy-with-LoRA-and-generative-ai-stable-diffusion-LoRA-39ff27480fda

Less memory usage = lowers GPU barriers!

Total Memory Usage (PyTorch) - Apply LoRA

LLM Fine-tuned

Total_memory = model_memory

LLM Fine-tuned (with LoRA)

Total_memory(Apply LoRA) = model_memory + LoRA Memory (Trainable Parameters)

Trainable parameters can be as small as 0.1%

Reference: https://xiaosean5408.medium.com/fine-tuning-LLMs-made-easy-with-LoRA-and-generative-ai-stable-diffusion-LoRA-39ff27480fda

Less memory usage = lowers GPU barriers!

Total Memory Usage (PyTorch) - Apply LoRA

LLM Fine-tuned

Total_memory = model_memory + gradient_memory

LLM Fine-tuned (with LoRA)

Total_memory(Apply LoRA) = model_memory + LoRA Memory (Trainable Parameters) + gradient_memory (Only consider trainable parameters)

Trainable parameters can be as small as 0.1%

Reference: https://xiaosean5408.medium.com/fine-tuning-LLMs-made-easy-with-LoRA-and-generative-ai-stable-diffusion-LoRA-39ff27480fda

Less memory usage = lowers GPU barriers!

Total Memory Usage (PyTorch) - Apply LoRA

LLM Fine-tuned

Total_memory = model_memory + gradient_memory + Optimization state (Depends on Optimizer, SGD:0, RMSProp:1,Adam:2) *gradient_memory

LLM Fine-tuned (with LoRA)

Total_memory(Apply LoRA) = model_memory + LoRA Memory (Trainable Parameters) + gradient_memory (Only consider trainable parameters) + Optimization state (Depends on Optimizer, SGD:0, RMSProp:1, Adam:2) * gradient_memory

Trainable parameters can be as small as 0.1%

Only consider trainable parameters

Reference: https://xiaosean5408.medium.com/fine-tuning-LLMs-made-easy-with-LoRA-and-generative-ai-stable-diffusion-LoRA-39ff27480fda

Less memory usage = lowers GPU barriers!

Total Memory Usage (PyTorch) - Apply LoRA

LLM Fine-tuned

Total_memory = model_memory + gradient_memory + Optimization state o (Depends on Optimizer, SGD:0, RMSProp:1,Adam:2) *gradient_memory + forward_pass_memory

LLM Fine-tuned (with LoRA)

Total_memory(Apply LoRA) = model_memory + LoRA Memory (Trainable Parameters) + gradient_memory (Only consider trainable parameters) + Optimization state o (Depends on Optimizer, SGD:0, RMSProp:1, Adam:2) * gradient_memory + forward_pass_memory (Including LoRA)

Trainable parameters can be as small as 0.1%

Only consider trainable parameters

Extra LoRA compute if not merged into base.

Reference: https://xiaosean5408.medium.com/fine-tuning-LLMs-made-easy-with-LoRA-and-generative-ai-stable-diffusion-LoRA-39ff27480fda

Less memory usage = lowers GPU barriers!

Total Memory Usage (PyTorch) - Apply LoRA

LLM Fine-tuned

Total_memory = model_memory + gradient_memory + Optimization state o (Depends on Optimizer, SGD:0, RMSProp:1,Adam:2) *gradient_memory + forward_pass_memory

LLM Fine-tuned (with LoRA)

Total_memory(Apply LoRA) = model_memory + LoRA Memory (Trainable Parameters) + gradient_memory (Only consider trainable parameters) + Optimization state o (Depends on Optimizer, SGD:0, RMSProp:1, Adam:2) * gradient_memory + forward_pass_memory (Including LoRA)

Trainable parameters can be as small as 0.1%

Only consider trainable parameters

Reference: https://xiaosean5408.medium.com/fine-tuning-LLMs-made-easy-with-LoRA-and-generative-ai-stable-diffusion-LoRA-39ff27480fda

Open source frameworks and LoRA

Out of the box on popular GPUs:

- PEFT in Hugging Face
- Unsloth
- Llama Factory

Hugging Face

unsloth

Pros

- Get started quickly
- Can train locally

Cons (of LoRA in general)

- Hyperparameters hard to optimize
- Local training only small models

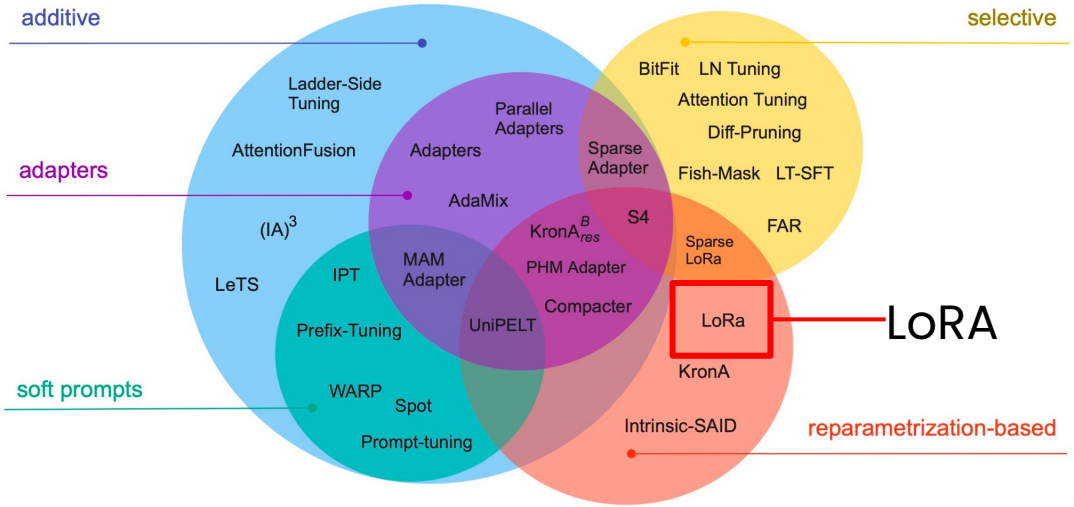
Huggingface example

```
from transformers import AutoModelForSeq2SeqLM
from peft import LoraModel, LoraConfig
config = LoraConfig(
    task_type="SEQ_2_SEQ_LM",
    r=8,
    lora_alpha=32,
    target_modules=["q", "v"]
    )
model = AutoModelForSeq2SeqLM.from_pretrained("t5-base")
lora_model = LoraModel(model, config, "default")
```

https://huggingface.co/docs/PEFT/en/package_reference/LoRA

PEFT: Parameter Efficient Fine-Tuning

PEFT is a set of techniques to make fine-tuning more computationally efficient & lightweight in storage



Scaling Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning (https://arxiv.org/abs/2303.15647)

DeepLearning.AI

Core Techniques in Fine-tuning and RL

RL: Rewards & preference learning

Reward is a key difference between RL and fine-tuning

RL loop

1. Get RL data {input, model output, reward}
2. Train LLM with RL, to maximize reward

3. Get RL data {input, model output, reward}
4. Train LLM with RL, to maximize reward

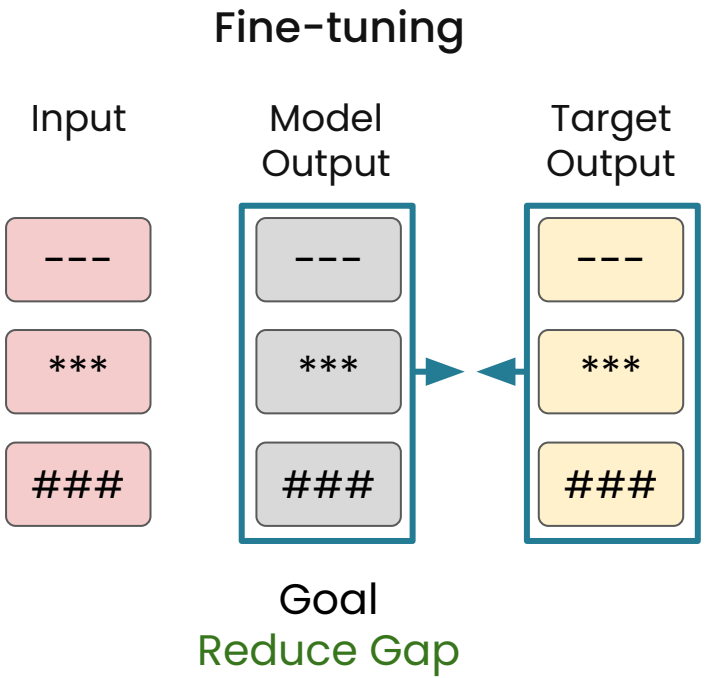
5. Get RL data {input, model output, reward}
6. Train LLM with RL, to maximize reward

...

Fine-tuning

1. Get data {input, target output}
2. Fine-tune LLM to minimize loss

Reward

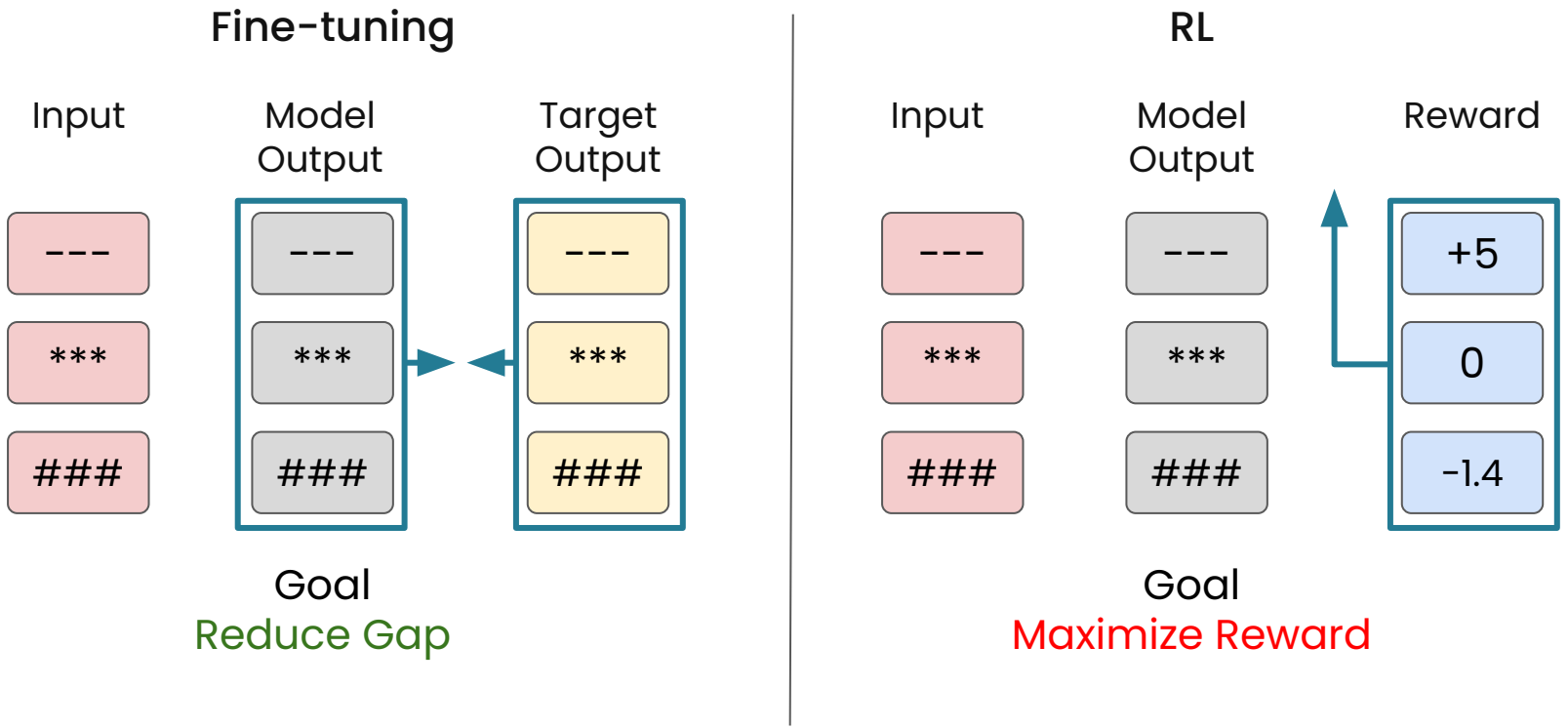

Fine-tuning
Input
Model Output
Target Output

***
###

***
###

***
###
Goal
Reduce Gap

Reward


Fine-tuning
Input
Model Output
Target Output

***
###

***
###

***
###
Goal
Reduce Gap
RL
Input
Model Output
Reward

***
###

***
###
+5
0
-1.4
Goal
Maximize Reward

Reward is scalar

"What does GPU stand for?"

Good Response

"Graphics Processing Unit."

+1

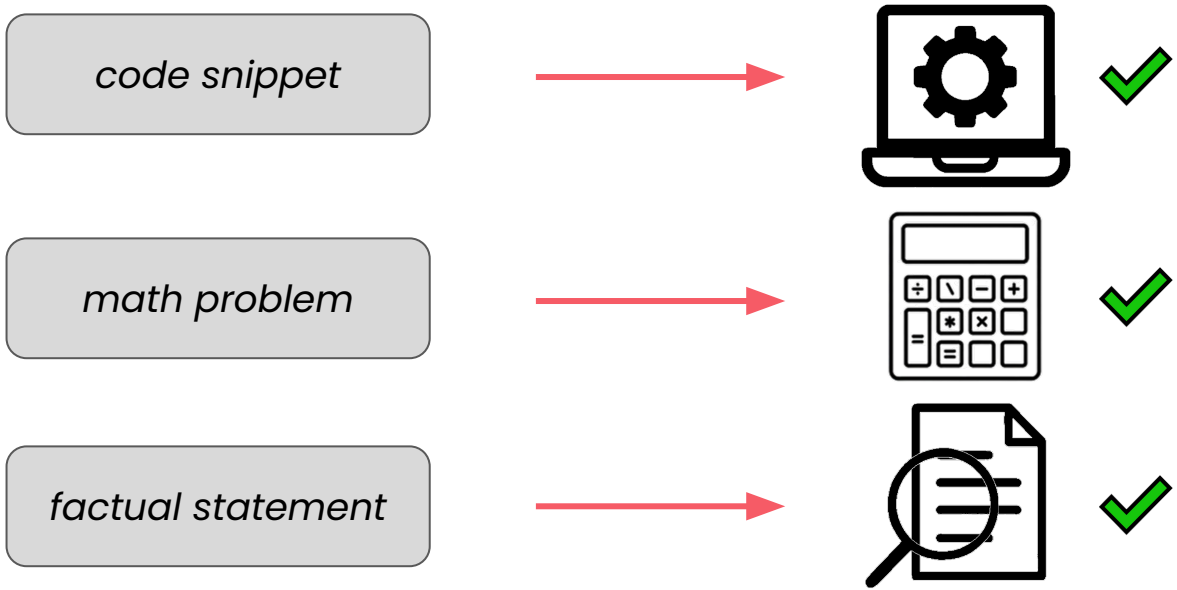
Bad Response

"Some kind of computer thing. Not sure really."

-1

Verifiable reward signals

Objective, checkable signals, codified in a function or logic. Not learned.



Verifiers in GRPO and DeepSeek models

```
def math_reward(response, correct_answer):
  if (extract_final_answer(response) ==
      correct_answer):
    return 1.0
  else:
    return 0.0
```

DeepSeek R1-Zero: reasoning must be in <think> tags

```
def format_reward(response):
  if ("<think>" in response and
      "</think>" in response):
    return 1.0
  else:
    return 0.0
```

```
def language_consistency_reward(response):
  if consistent_language(response):
    return 1.0
  else:
    return -0.5
```

DeepSeek R1: don't mix English and Chinese

Verifiers in GRPO and DeepSeek models

```
def combined_reward(response, correct_answer):
  accuracy = math_reward(response, correct_answer)
  format_check = format_reward(response)
  language_check = language_consistency_reward(response)
  return accuracy + format_check + language_check
```

In your code

```
if predicted == correct_answer:
    reward = 1.0
```

In your code

```
if predicted == correct_answer:
    reward = 1.0
else:
    relative_error = abs(predicted - correct_answer) / correct_answer

    if relative_error < 0.01:
        reward = 1.0
    elif relative_error < 0.1:
        reward = 0.8
    elif relative_error < 0.3:
        reward = 0.5
```

Verifiers are not always possible

Input

Carly has 8 apples and buys 2 more, but then sells 5 to the local baker.
How does Carly feel?

Grader

Math checker

🤯🤯🤯🤯🤯🤯

Verifiers are not always possible

Input

Carly has 8 apples and buys 2 more, but then sells 5 to the local baker.
How does Carly feel?

Grader

LLM or another model to output reward

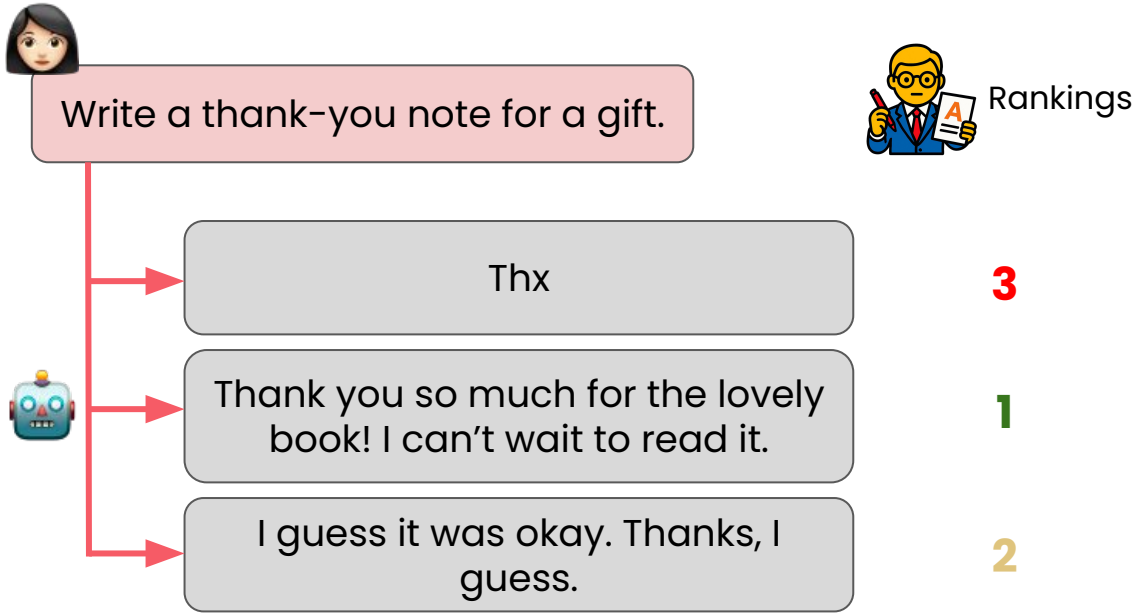
Instead: Train a reward model!

Reward model behavior

Input		Carly has 8 apples and buys 2 more, but then sells 5 to the local baker. **How does Carly feel?**
Model Output		Carly feels productive!
Reward Model Output		+1.3

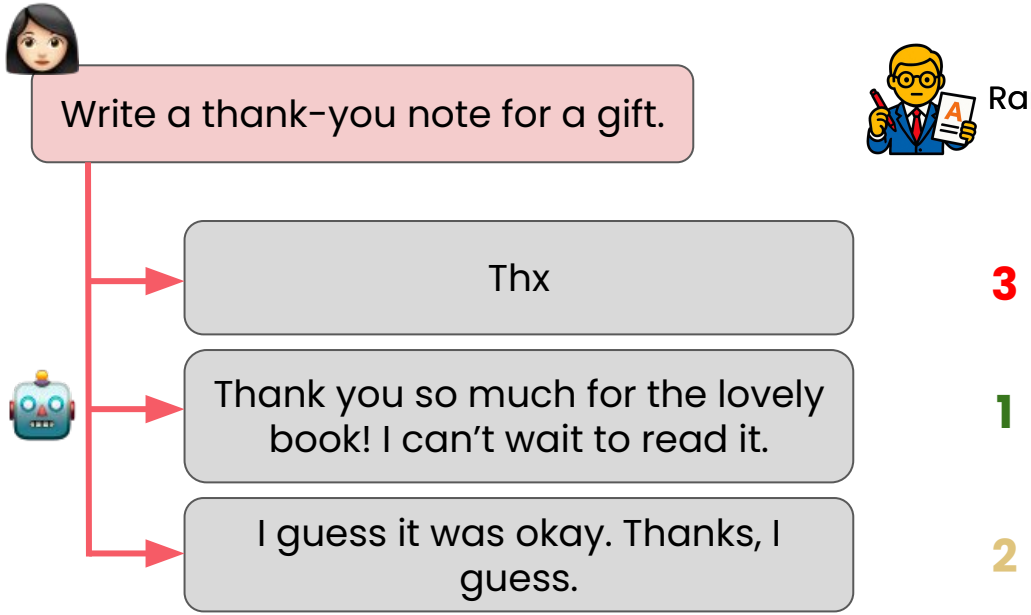
Data to train a reward model

Get reward model to learn human preferences from human preference data



Reward model needs to output a scalar reward

Convert human preference rankings into a loss function for scalar rewards



Rankings

The reward model learns...

Rankings → loss & scalar rewards

Preference learning: Train a reward model

Rankings

The reward model learns...

Rankings → loss & scalar rewards

3

1

2

Ranked preferences

Preference learning: Train a reward model

Rankings

The reward model learns...

Rankings → loss & scalar rewards

3

1

2

Ranked preferences

Pairs
A >> B

1 >> **3**
1 >> **2**
1 >> **3**

Preference pairs

Preference learning: Train a reward model

Rankings

The reward model learns...

Rankings → loss & scalar rewards

3

1

2

Ranked preferences

Predict r(A) & r(B)

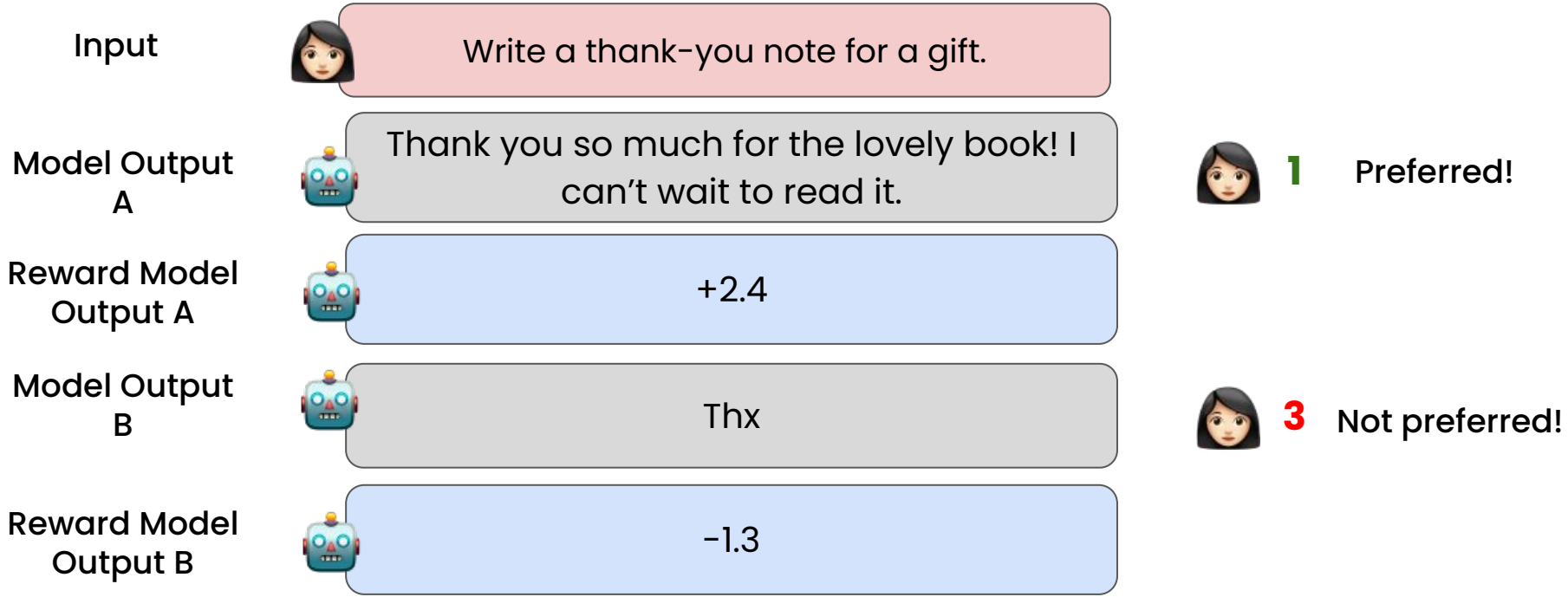
Pairs
A >> B

1 >> **3**
1 >> **2**
1 >> **3**

Preference pairs

Preference Learning

Reward model behavior



Preference learning: Train a reward model

Rankings

The reward model learns...

Rankings → loss & scalar rewards

3

1

2

Ranked preferences

Pairs
A >> B

1 >> 3
1 >> 2
1 >> 3

Preference pairs

Predict r(A) & r(B)

difference = r(A)-r(B)

Preference Learning

DeepLearning.AI

AMD

Preference learning: Train a reward model

Rankings

The reward model learns...

Rankings → loss & scalar rewards

3

1

2

Ranked preferences

Pairs
A >> B

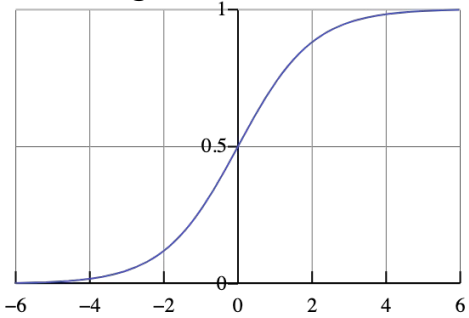
1 >> **3**
1 >> **2**
1 >> **3**

Predict r(A) & r(B)

difference = r(A)-r(B)

σ(difference): p(A >> B)
1 if r(A) >> r(B)
0.5 if r(A) = r(B)
0 if r(A) << r(B)

Sigmoid (σ)



Preference pairs

Preference Learning

Preference learning: Train a reward model

Rankings

The reward model learns...

Rankings → loss & scalar rewards

3

1

2

Ranked preferences

Pairs
A >> B

1 >> **3**
1 >> **2**
1 >> **3**

Predict r(A) & r(B)

difference = r(A)-r(B)

σ(difference): p(A >> B)

L=-logσ(difference)

Loss function!

Preference pairs

Preference Learning

DeepLearning.AI

AMD

Preference learning: Train a reward model

Rankings

The reward model learns...

Rankings → loss & scalar rewards

3

1

2

Ranked preferences

Pairs
A >> B

1 >> **3**
1 >> **2**
1 >> **3**

Predict r(A) & r(B)

difference = r(A)-r(B)

σ(difference): p(A >> B)

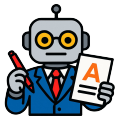
L=-logσ(difference)

Reward model = Preference model

Preference pairs

Preference Learning

Preference learning: Train a reward model



Rankings

LLMs can do rankings too
RLHF → RLAIF

3

1

2

Ranked
preferences

The reward model
learns...

Rankings → loss & scalar rewards

Pairs
A >> B

1 >> **3**
1 >> **2**
1 >> **3**

Predict r(A) & r(B)

difference = r(A)-r(B)

σ(difference): p(A >> B)

L=-logσ(difference)

Reward model =
Preference model

Preference
pairs

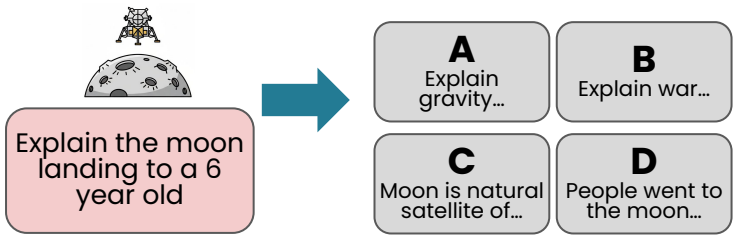
Preference
Learning

DeepLearning.AI

AMD

RLHF: reward model from preference learning

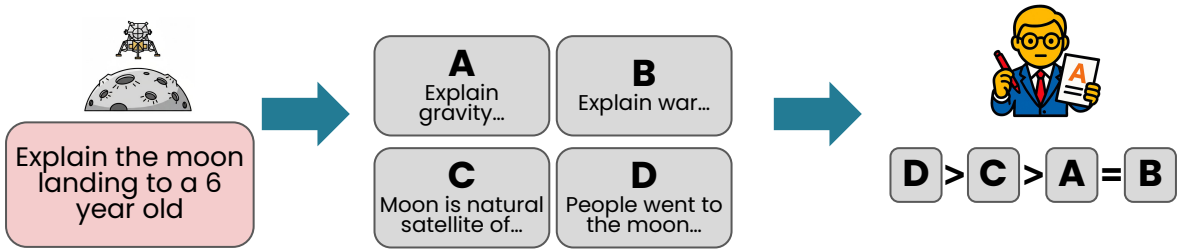
Gather data for reward model



An input, and several model outputs, are sampled.

RLHF: reward model from preference learning

Gather data for reward model

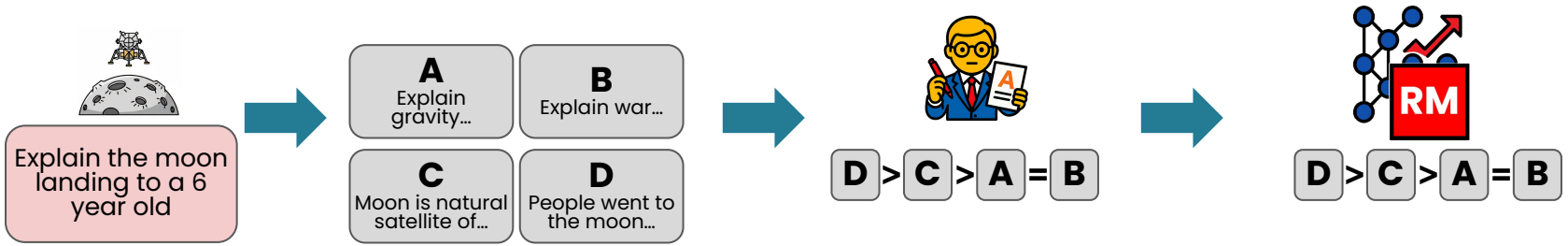


An input, and several model outputs, are sampled.

A labeler ranks the model outputs from best to worst.

RLHF: reward model from preference learning

Train reward model with preference learning



An input, and several model outputs, are sampled.

A labeler ranks the model outputs from best to worst.

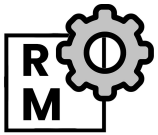
Train reward model on ranked preference data.

Reward model vs. Verifiers (use both!)

Reward model 

Collect preference data 

↓

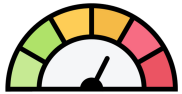
Train reward model 

↓

Generate learned reward signals 

Data can represent your objective

Verifiers 

Define objective metrics 

↓

Implement measurable checks 

↓

Generate verifiable signals 

Functions can represent your objective

Reward model vs. Verifiers (use both!)

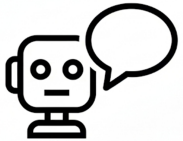
Reward model

RM

Best for:

Aligning with human values/preferences

Improving general dialogue quality



Scaling feedback efficiently

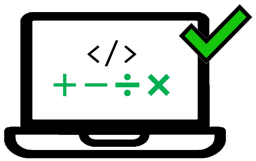
Verifiers

VR

Best for:

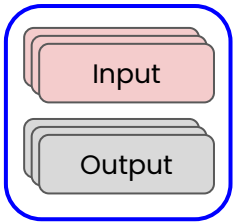
Ensuring factual correctness

Solving math or coding tasks



Getting RL data

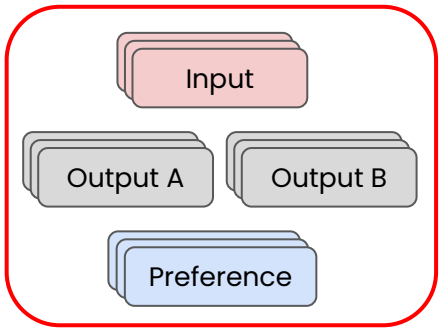
Get {input, model output}
(also called “rollouts”)

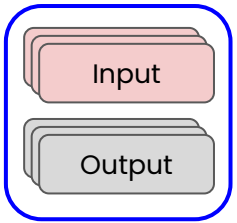


Getting RL data

If using reward model

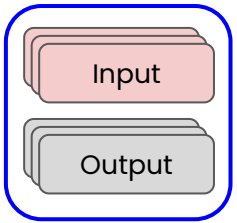
Get preference data: human (or synthetic) preferences → convert to preference pairs.

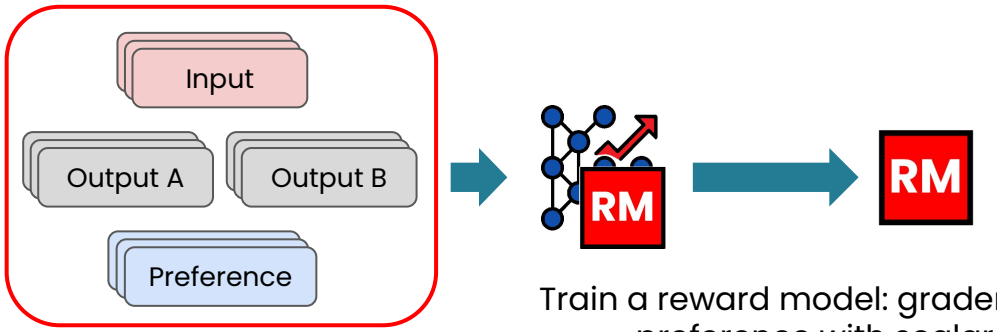




Getting RL data

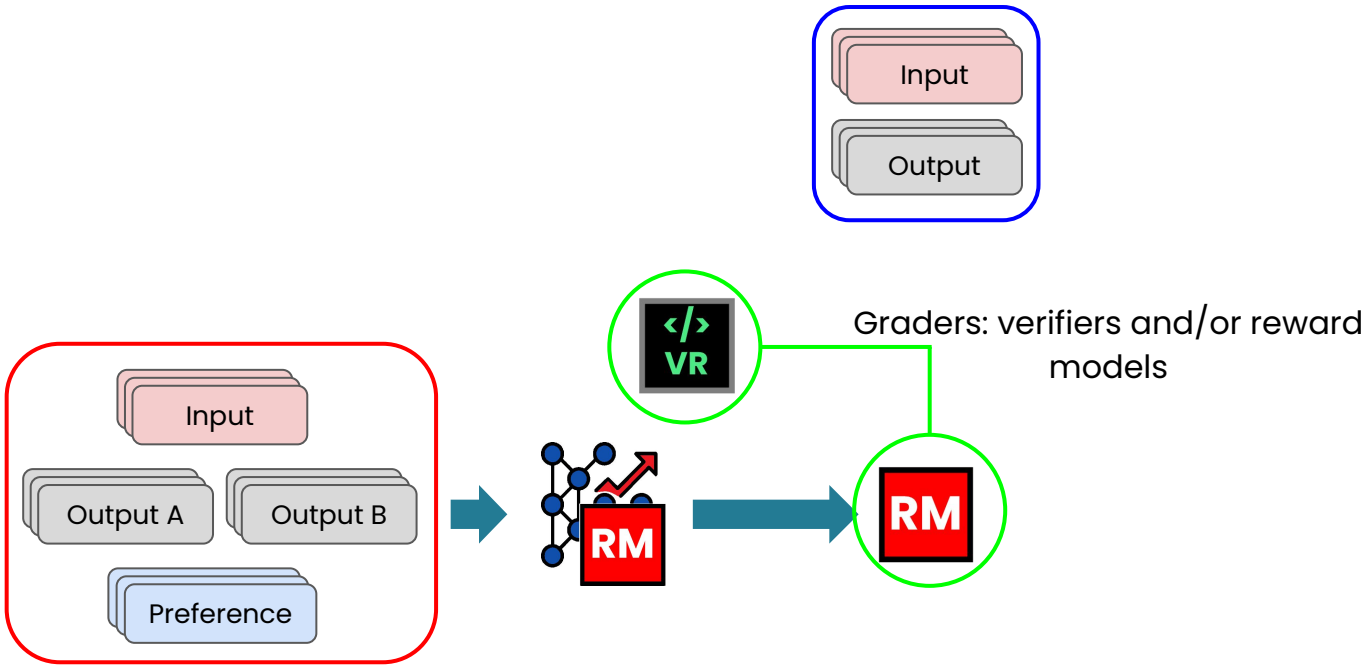
If using reward model



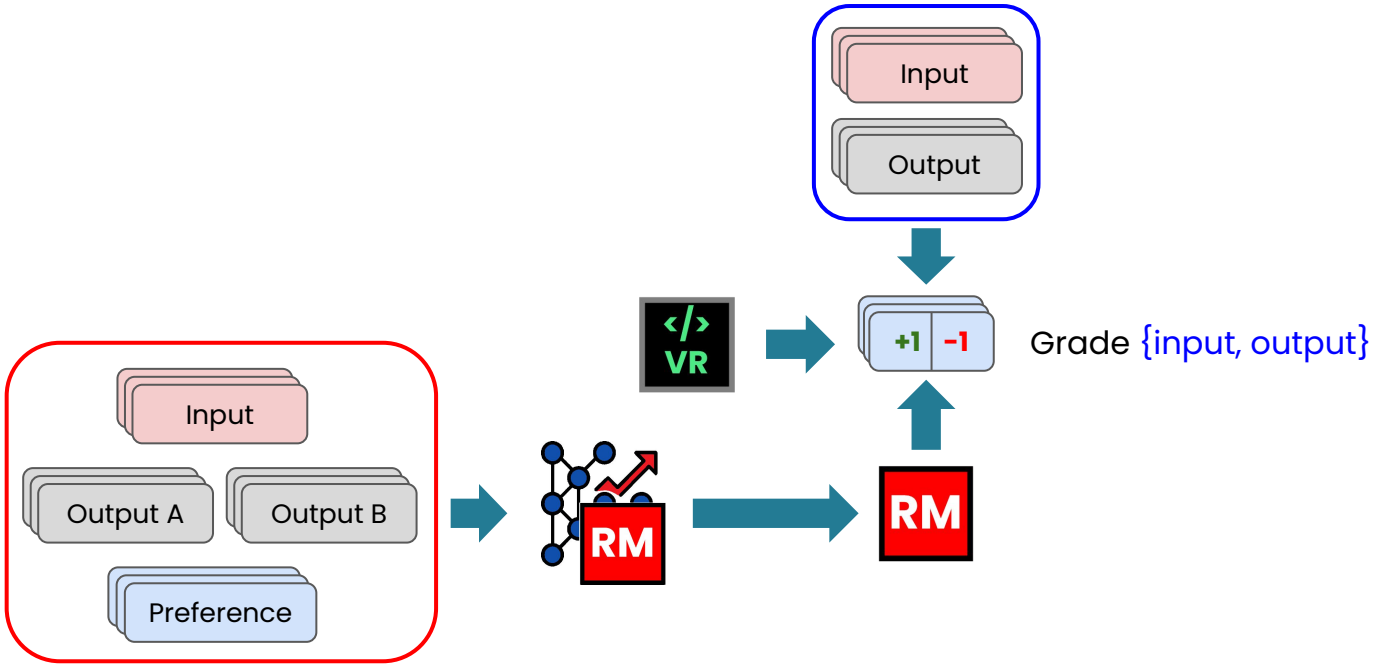


Train a reward model: grader that predicts preference with scalar reward.

Getting RL data

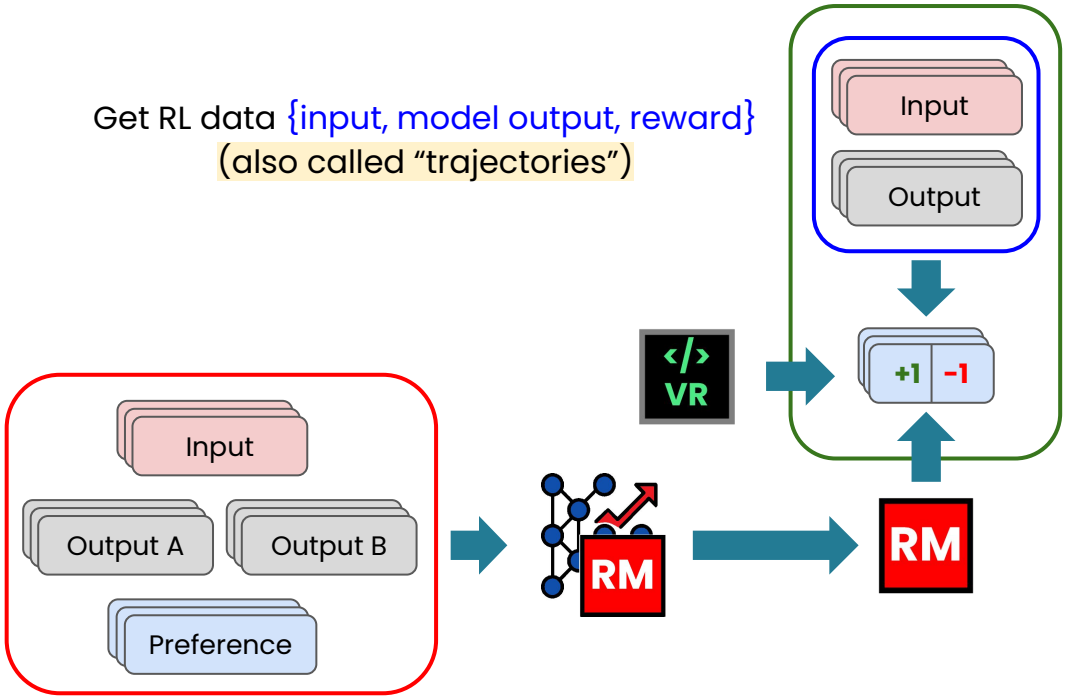


Getting RL data

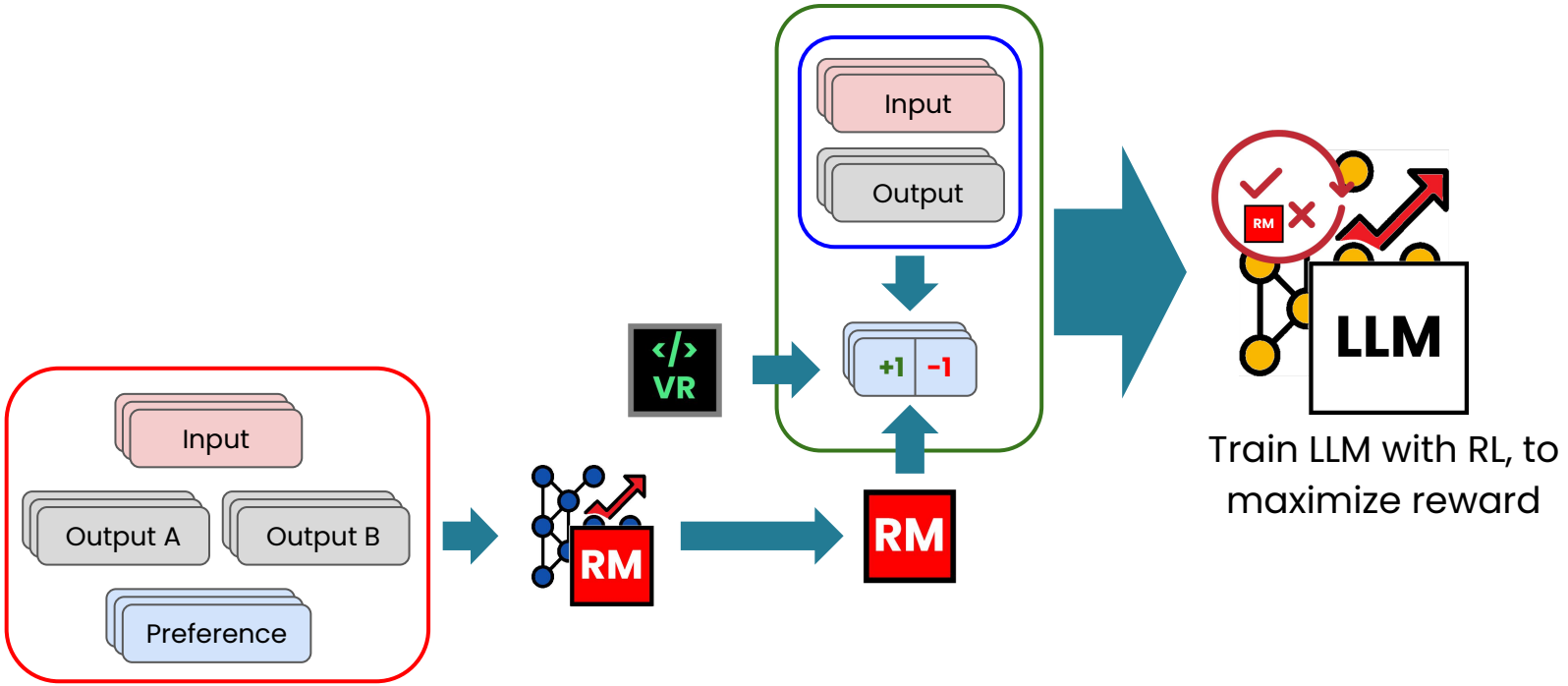


Getting RL data

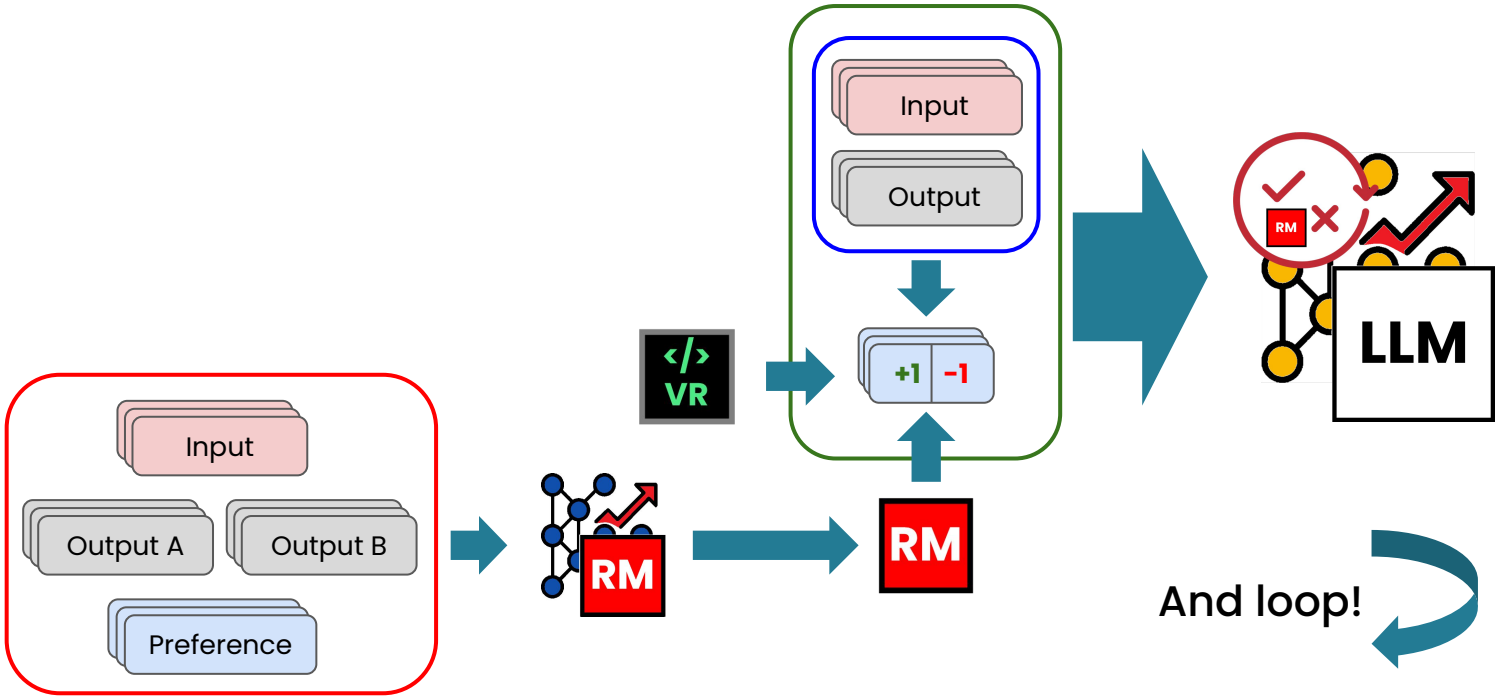
Get RL data {input, model output, reward}
(also called “trajectories”)



Getting RL data



Getting RL data

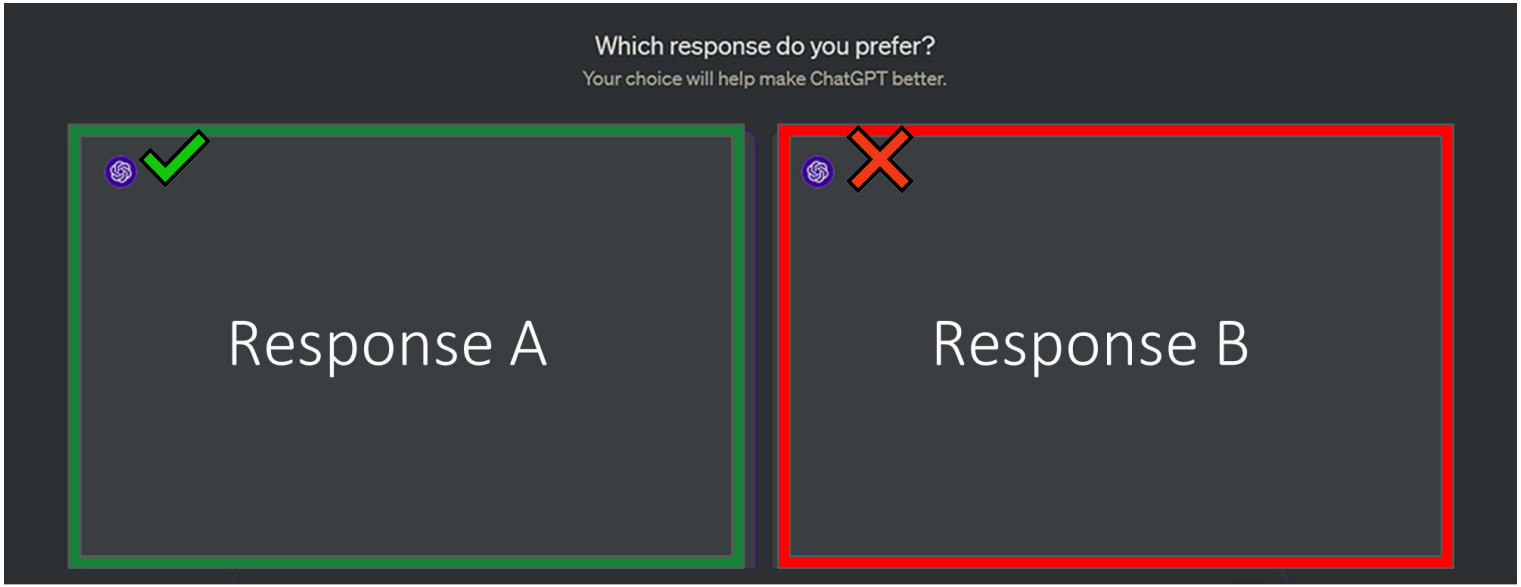


DeepLearning.AI

Core Techniques in Fine-tuning and RL

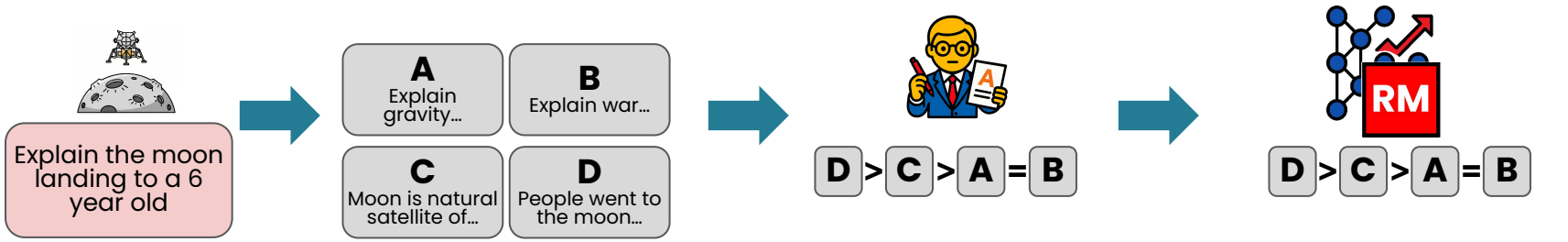
RL: RL training objective and RLHF

ChatGPT uses RL with human feedback (RLHF)



RLHF: Reward model from preference learning

Train reward model with preference learning



An input, and several model outputs, are sampled.

A labeler ranks the model outputs from best to worst.

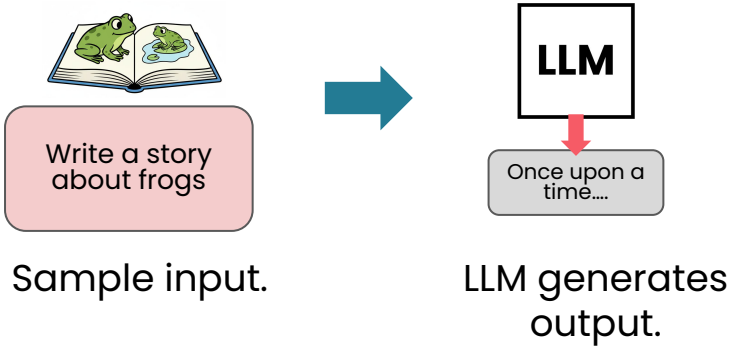
Train reward model with preference learning.

RLHF: Train LLM with reward model + RL

Write a story about frogs

Sample input.

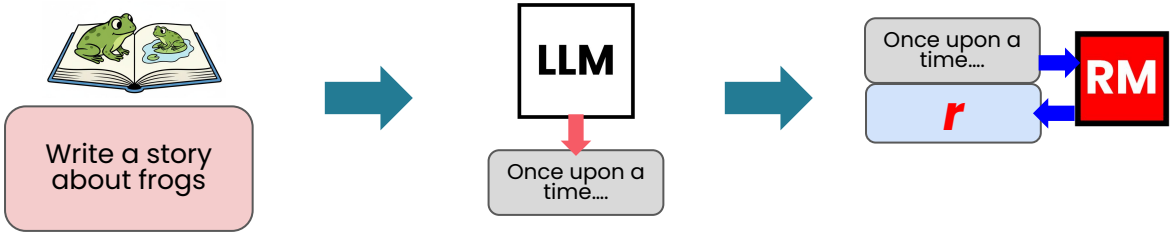
RLHF: Train LLM with reward model + RL



Sample input.

LLM generates output.

RLHF: Train LLM with reward model + RL

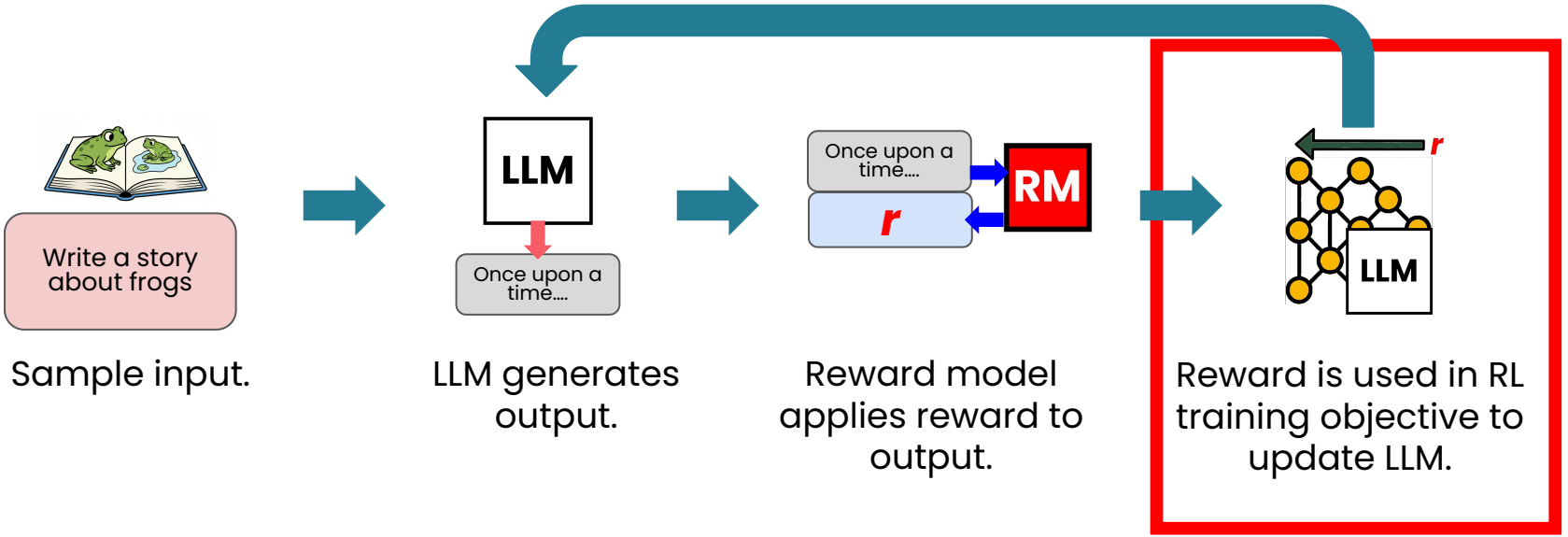


Sample input.

LLM generates output.

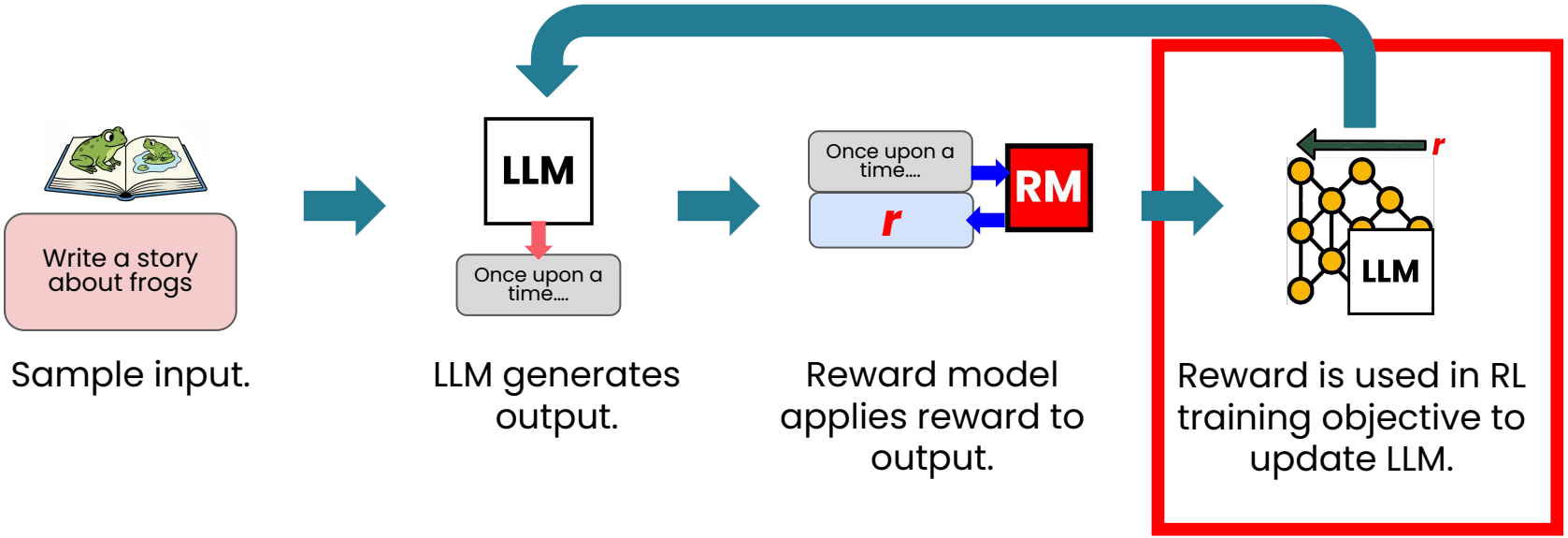
Reward model applies reward to output.

RLHF: Train LLM with reward model + RL



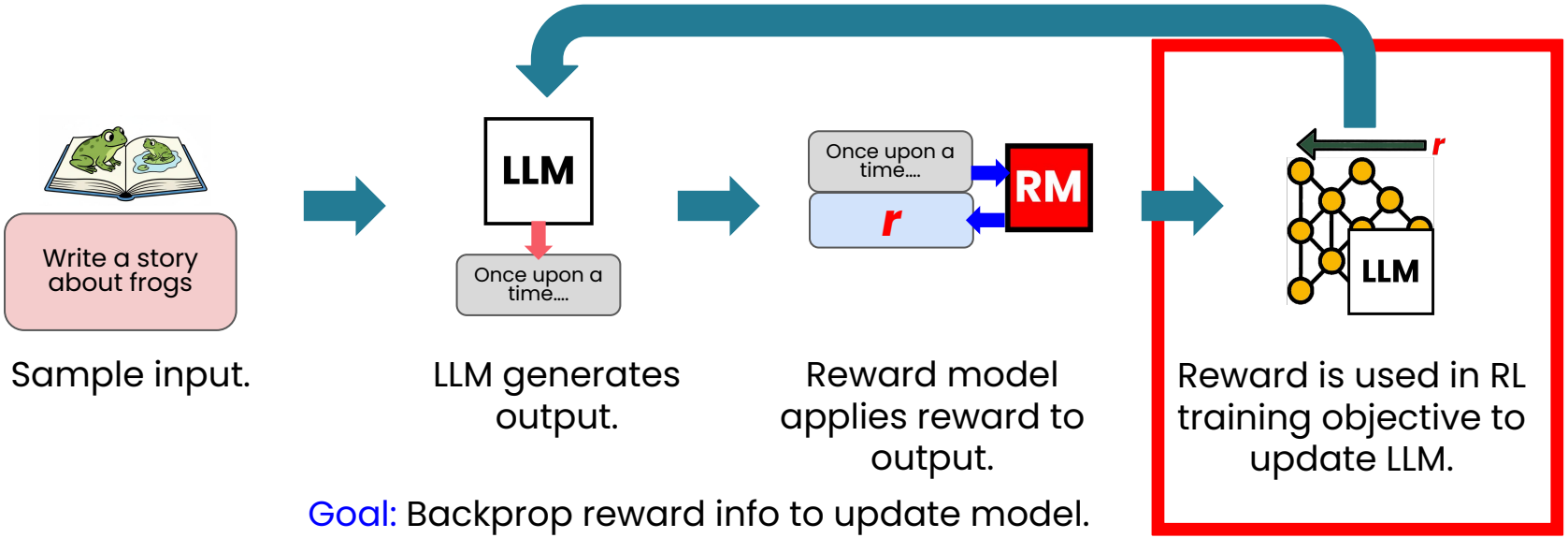
RLHF: Train LLM with reward model + RL

How exactly is the reward used to update the LLM?



RLHF: Train LLM with reward model + RL

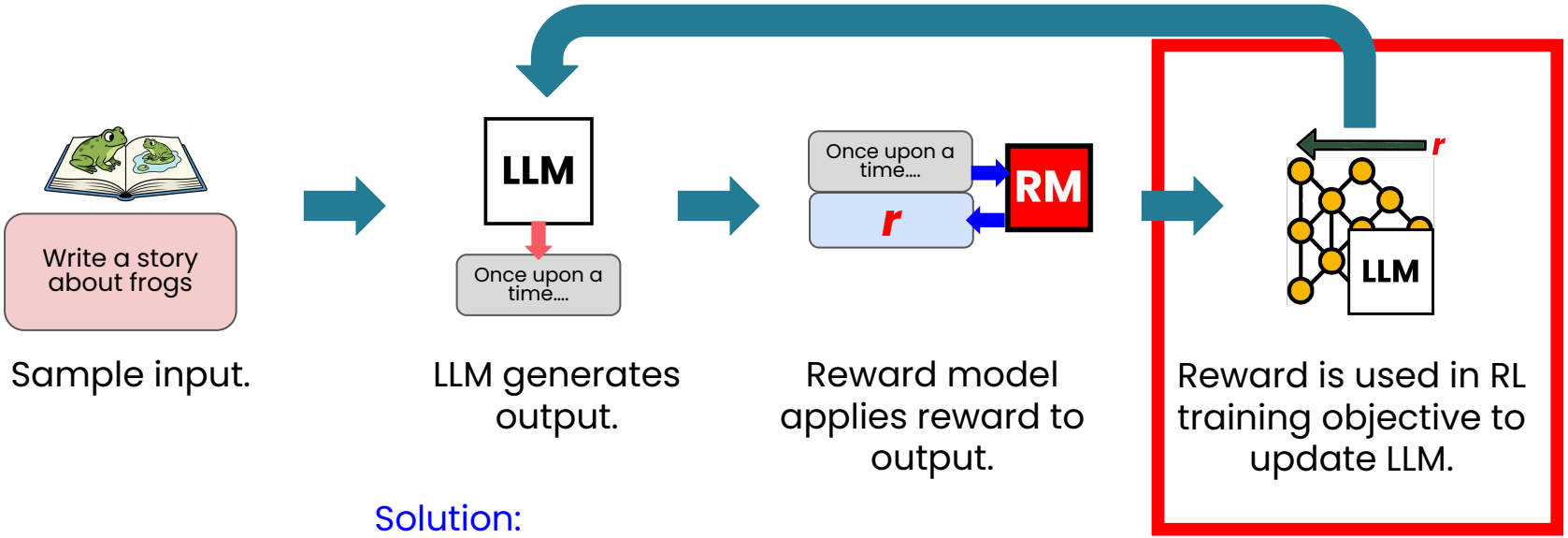
How exactly is the reward used to update the LLM?



Goal: Backprop reward info to update model.
Problem: Reward not directly differentiable.

RLHF: Train LLM with reward model + RL

How exactly is the reward used to update the LLM?



Solution:
Training Objective: Weight LLM output, with info from its reward

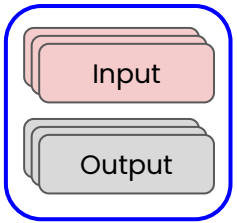
Running RL

RL loop:

1. Get RL data
    a. Get rollouts {input, model output}
    b. Apply reward → trajectories {input, model output, reward}
        i. In RLHF, train a reward model with preference learning
2. Train LLM on trajectories with RL
    a. Define a training objective using rollouts to backprop reward info
    b. Update LLM with training objective, to maximize reward

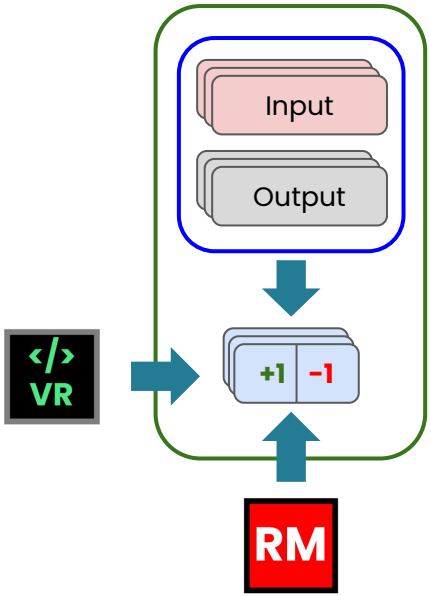
Running RL

Get RL data



Get rollouts {input, model output}

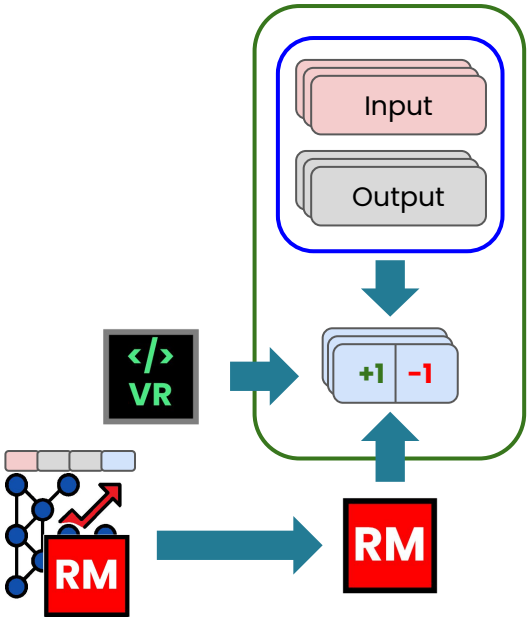
Running RL



Get RL data

Apply reward → trajectories {input, model output, reward}

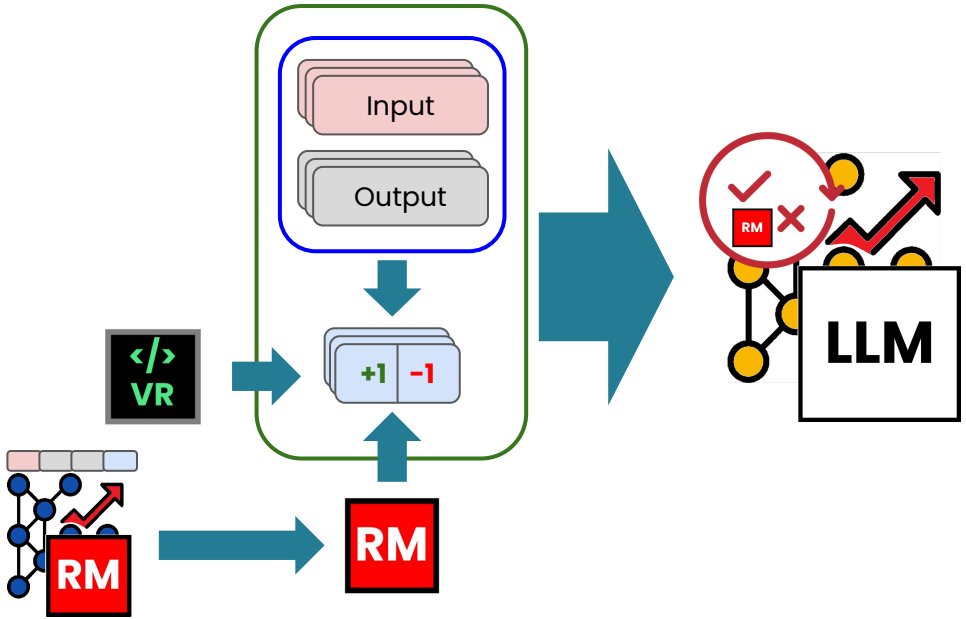
Running RL



In RLHF, train a reward model with preference learning

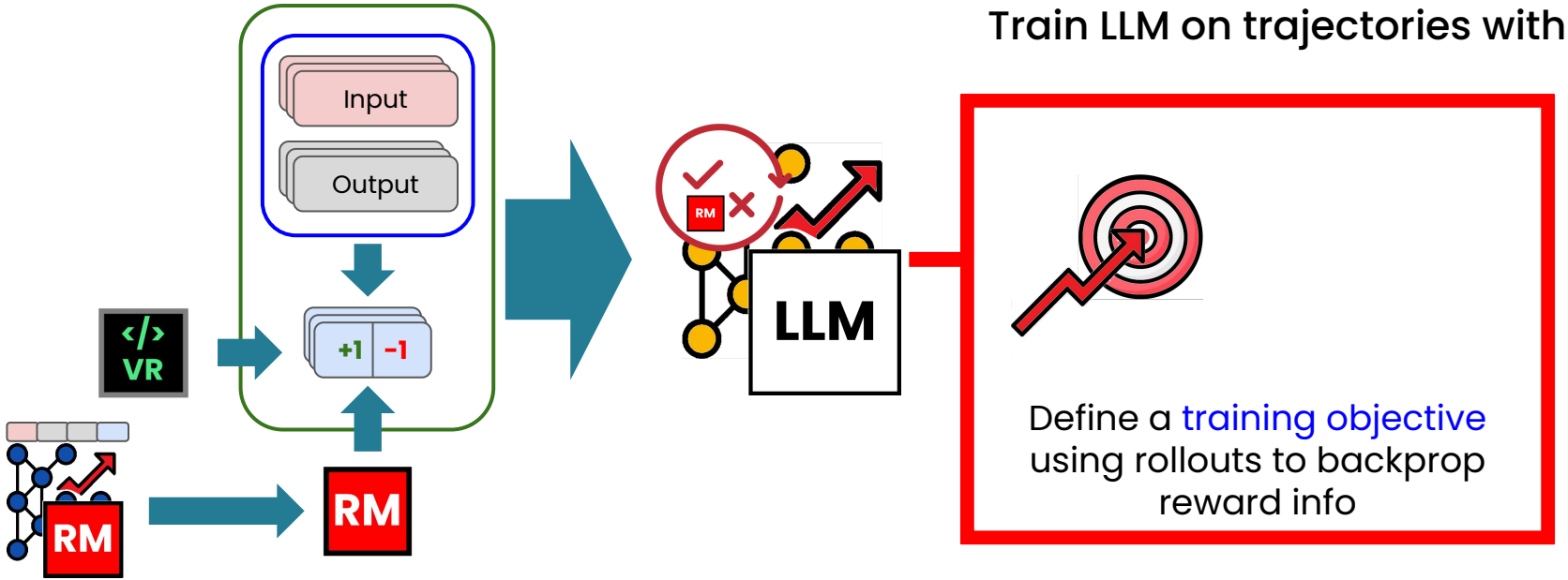
Get RL data

Running RL

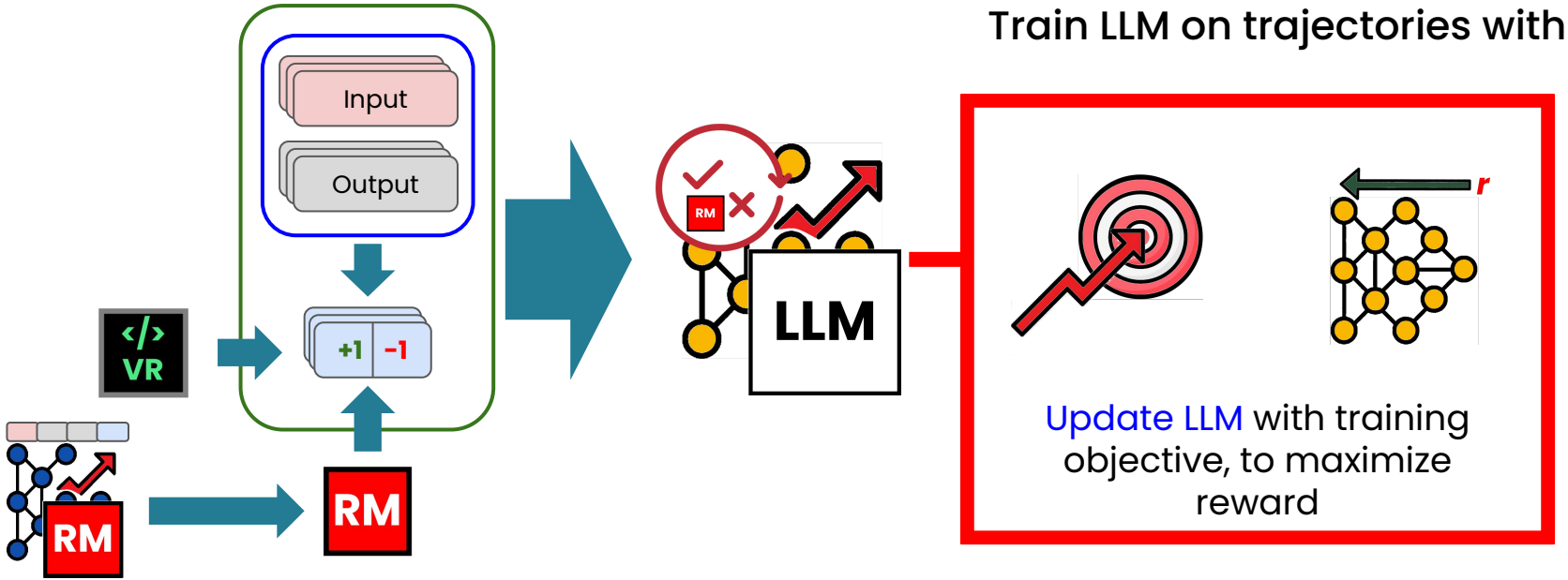


Train LLM on trajectories with RL

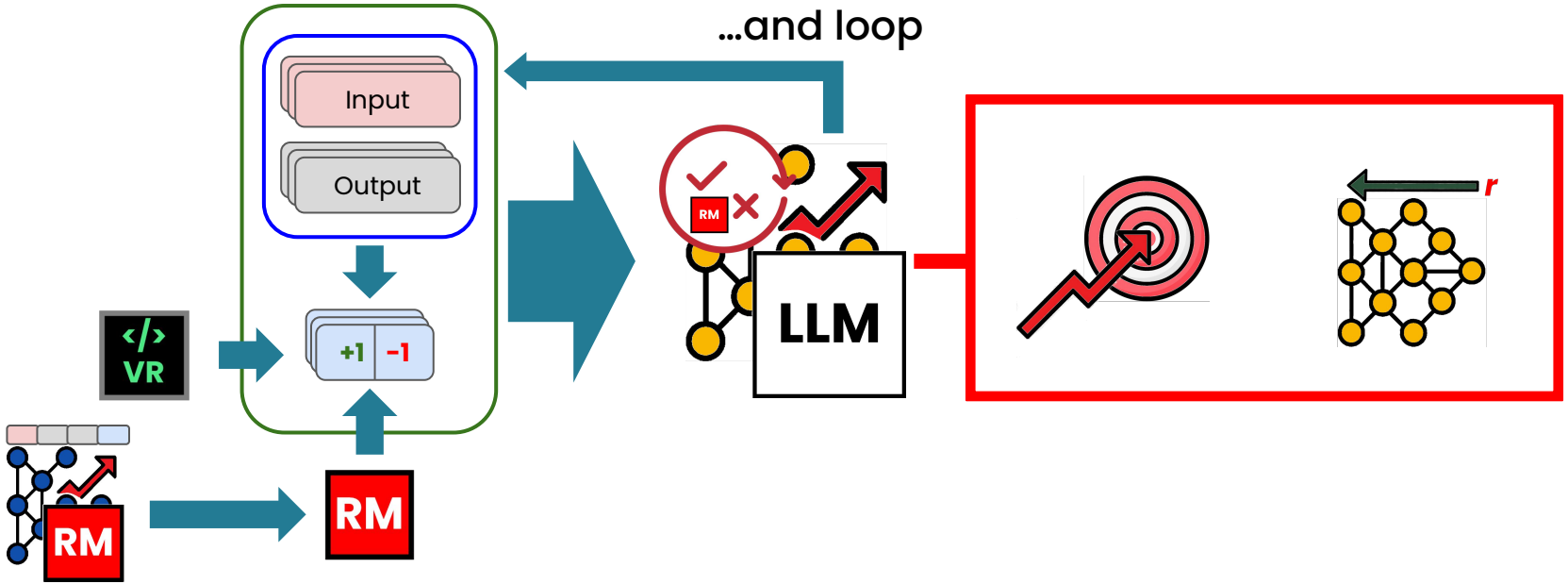
Running RL


Input
Output
+1
-1
</>
VR
RM
RM
RM
LLM
Train LLM on trajectories with RL
Define a training objective using rollouts to backprop reward info

Running RL


Input
Output
+1
-1
</>
VR
RM
RM
RM
LLM
Train LLM on trajectories with RL
r
Update LLM with training objective, to maximize reward

Running RL



RL training objective

Weight LLM output, with its reward

RL training objective

Weight LLM output, with its reward

Weight the probability of an output token, with how good token is

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

RL training objective

Weight LLM output, with its reward

Weight the probability of an output token, with how good token is

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

Based on reward

RL training objective: Calculating token prob

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

$$\mathbf{LLM}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

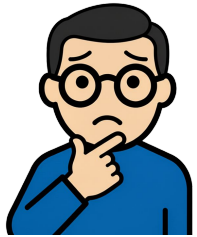
Your LLM, that you're training

LLM(input so far) → next token probability

RL training objective: Calculating token prob

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

$$\mathbf{LLM}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$



During training, your LLM is changing on every batch.

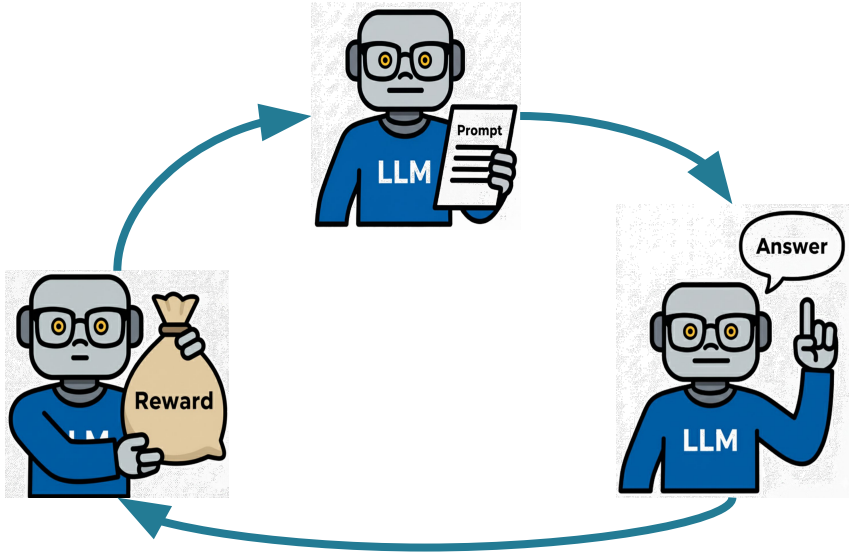
This means that the collected data will diverge from what it'd generate now - from the same prompt.

This is unstable!

RL training objective: Phases

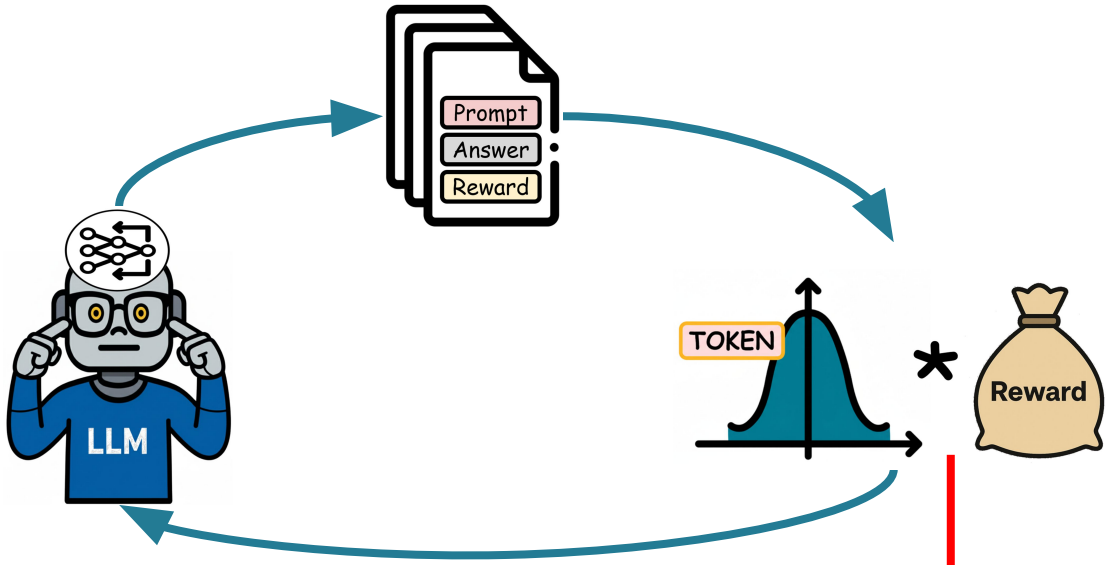
Phase 1: Collect data {input, output, reward}

LLM generates a dataset of rollouts. Apply rewards to get trajectories.



RL training objective: Phases

Phase 2: Train on trajectories



Training Objective

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

RL training objective: Calculating token prob

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

$$\frac{\mathbf{LLM}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}{\mathbf{LLM}_{\text{ref}}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}$$

Using the original old probabilities, weight the new probabilities accordingly.

Your LLM / Reference LLM

Always updating *Collected the original data*

RL training objective: Calculating token prob

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

$$\frac{\mathbf{LLM}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}{\mathbf{LLM}_{\text{ref}}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}$$

Ratio = 1 Same probability to this token
Ratio > 1 Higher probability to this token
Ratio < 1 Lower probability to this token

Your LLM / Reference LLM

Always updating *Collected the original data*

Running RL

RL loop:

1. Get RL data
    a. Get rollouts {input, output} from reference LLM
    b. Apply reward → trajectories {input, output, reward}
        i. In RLHF, train a reward model with preference learning
2. Train your main LLM on trajectories with RL
    a. Define a training objective using trajectories to backprop reward info
    b. Update LLM with training objective, to maximize reward

RL training objective: Calculating advantage

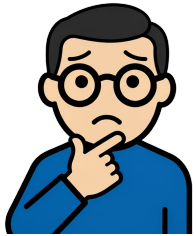
$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

$$r\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

RL training objective: Calculating advantage

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

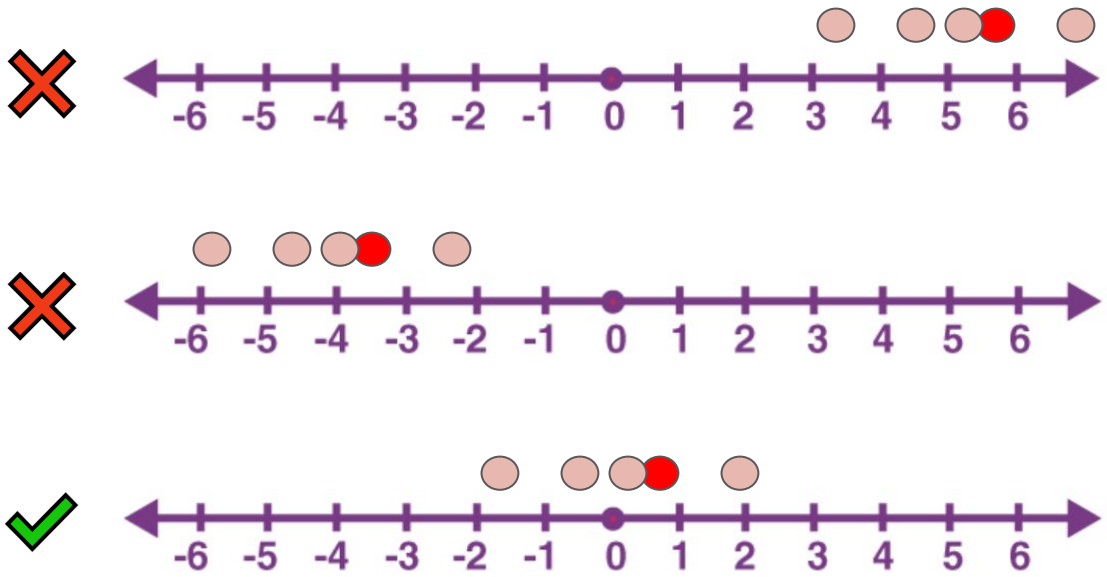
Technically correct, but learning will be slower if rewards are not clustered around zero.



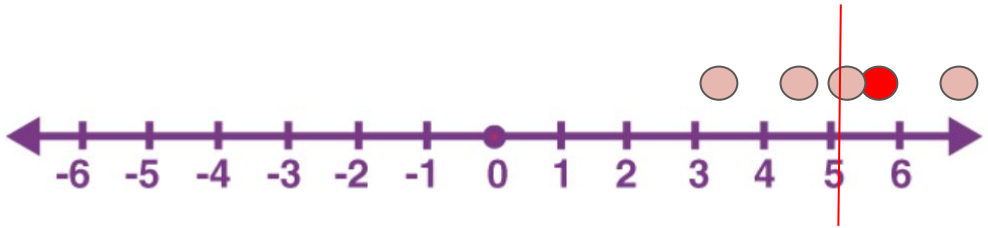
$$\times \quad r\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

RL training objective: Baseline

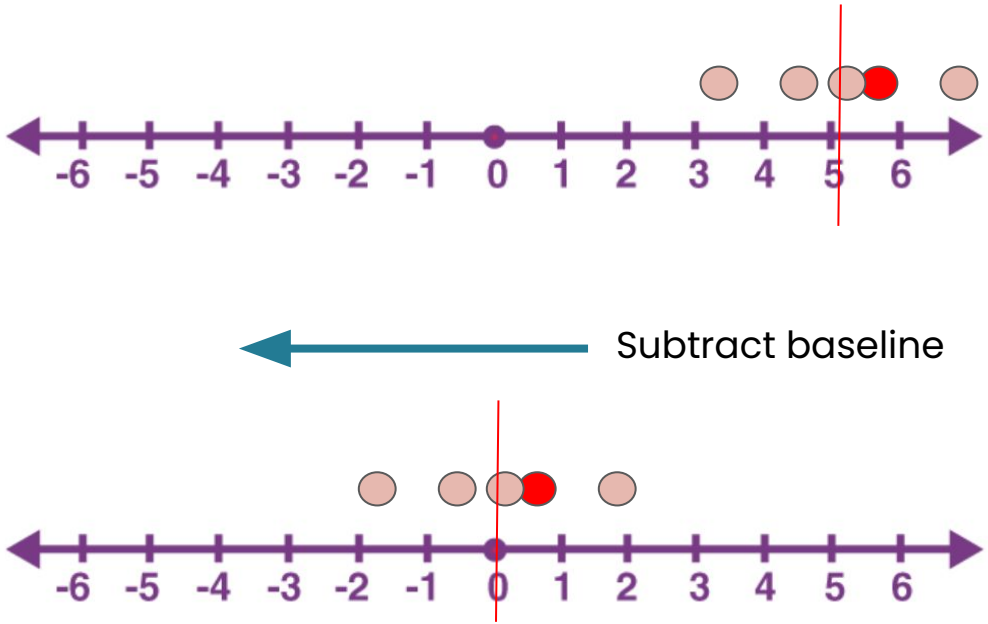
Learning is slower when the rewards for model outputs don't cluster around zero



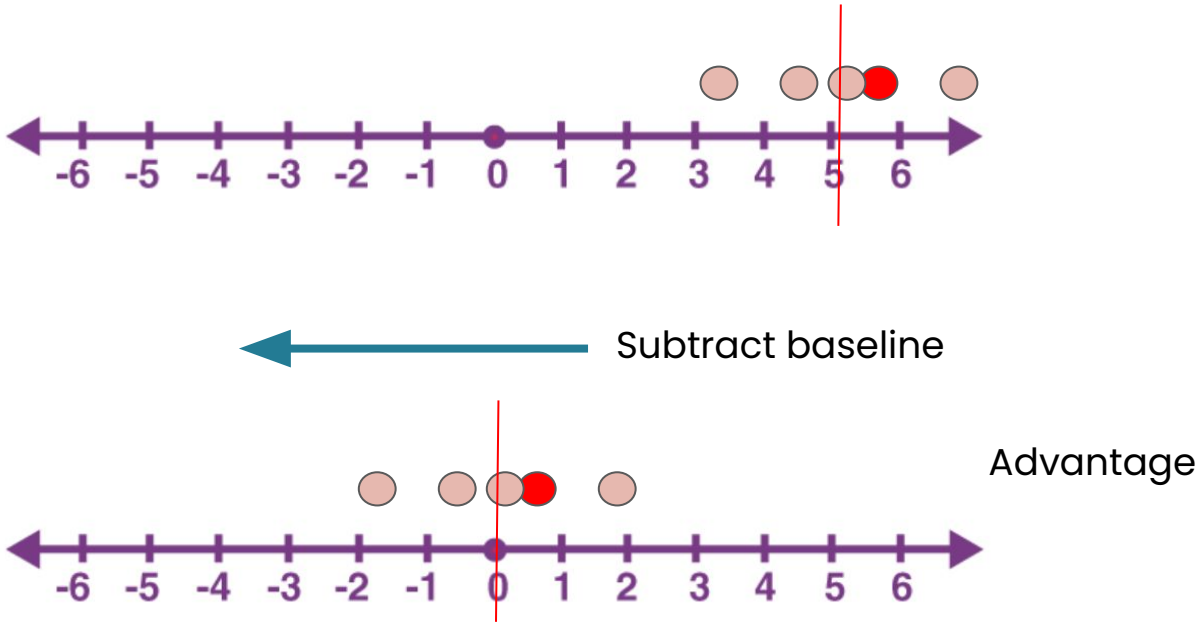
RL training objective: Baseline



RL training objective: Baseline



RL training objective: Baseline



RL training objective: Calculating advantage

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

$$r\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right) - b\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

Subtract a baseline

RL training objective: Calculating advantage

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

Advantage

$$r\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right) - b\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

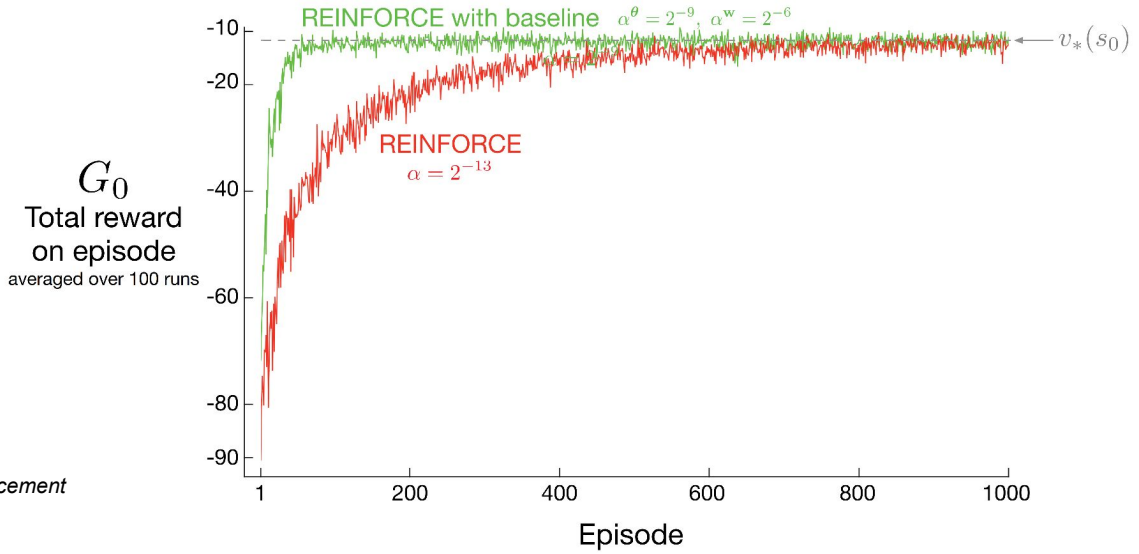
Subtract a baseline

RL training objective: Calculating advantage

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

Advantage

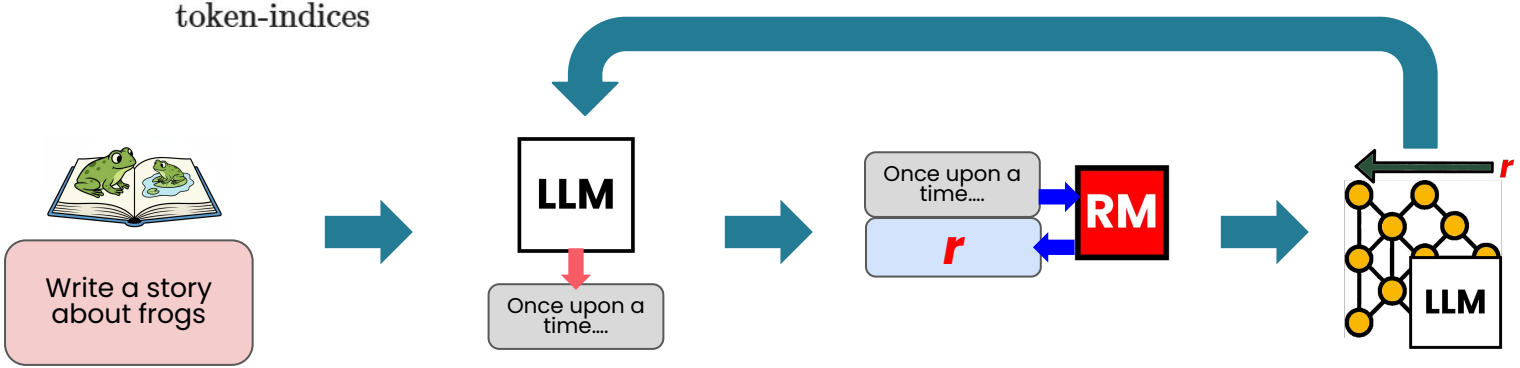
Baselining speeds up learning.



Source: Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction (2nd ed.).

RLHF: Train LLM with Reward Model + RL

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$



Sample input.

LLM generates output.

Reward model applies reward to output.

Reward is used in RL training objective to update LLM.

A lot of models to keep track of in RL

1. Your LLM - learning!

$$\mathbf{LLM}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

2. LLM_{ref} - frozen (inference only), generates rollouts

$$\mathbf{LLM}_{\text{ref}}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

A lot of models to keep track of in RL

1. Your LLM - learning! $\mathbf{LLM}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)$
2. $\mathrm{LLM}_{\mathrm{ref}}$ - frozen (inference only), generates rollouts $\mathbf{LLM}_{\mathrm{ref}}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)$
3. Reward Model $\boldsymbol{r}\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right)$
4. Baseline Estimation LLM $b\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right)$

A lot of models = RL is compute-intensive

- Memory: Multiple 3-4 models in memory.
- Need: 2–4x fine-tuning memory footprint.

Look for very high FLOPs and high VRAM.

$$\mathbf{LLM}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

$$\mathbf{LLM}_{\text{ref}}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

$$\boldsymbol{r}\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

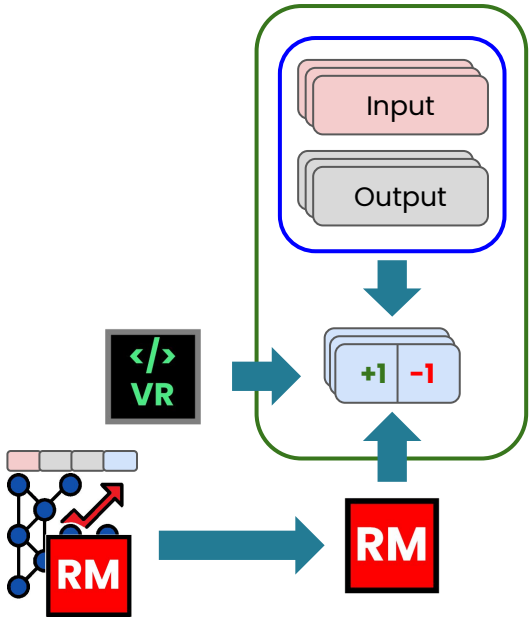
$$b\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

Running RL: Part 1

RL loop:

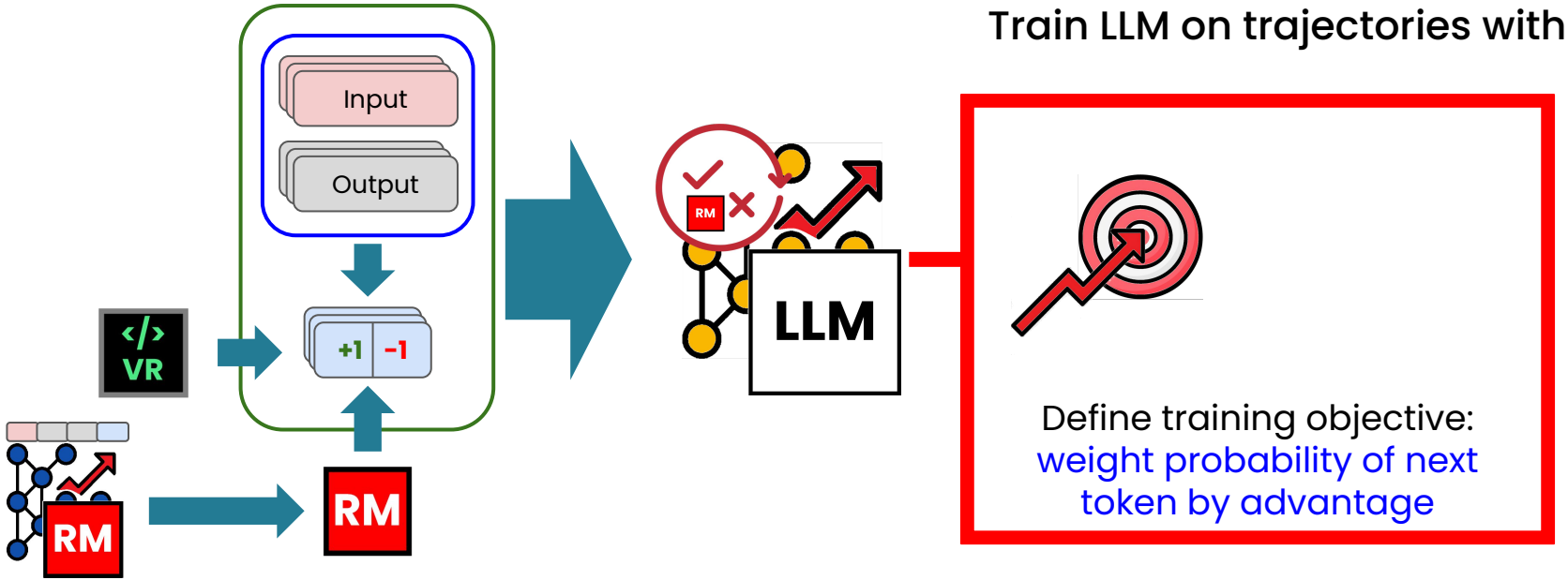
1. Get RL data
    a. Get rollouts {input, output} from reference LLM
    b. Apply reward → trajectories {input, output, reward}
        i. In RLHF, train a reward model with preference learning
2. Train your main LLM on trajectories with RL
    a. Define training objective: weight probability of next token by advantage
        i. Use reference LLM to stabilize measure for probability of next token
        ii. Measure advantage using reward, adjusted by a baseline
    b. Update LLM with training objective, to maximize reward
3. Continue steps 1+2 until completion

Running RL: Part 1

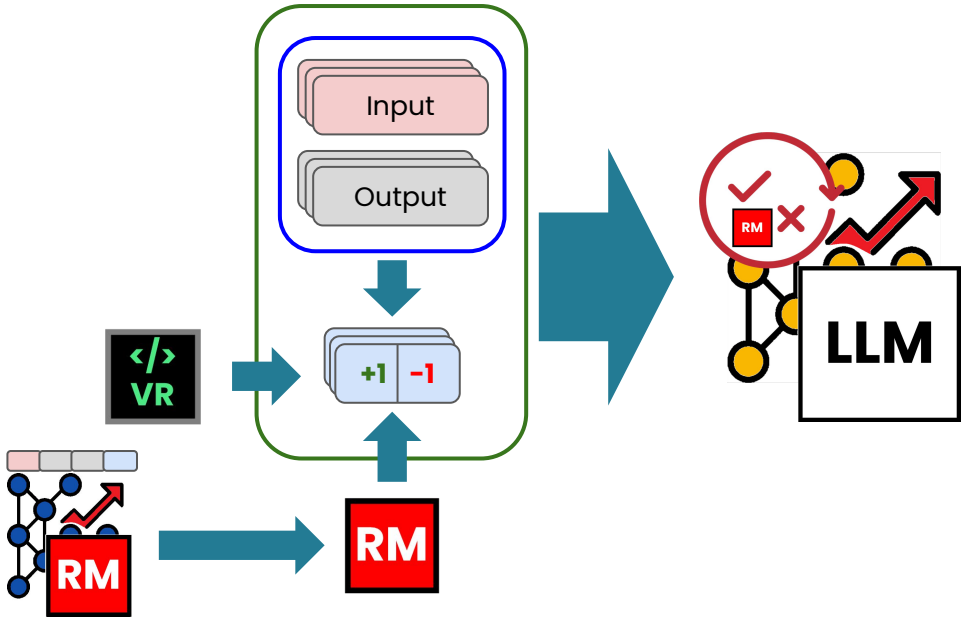


Get RL data
(as previously explained)

Running RL: Part 1



Running RL: Part 1

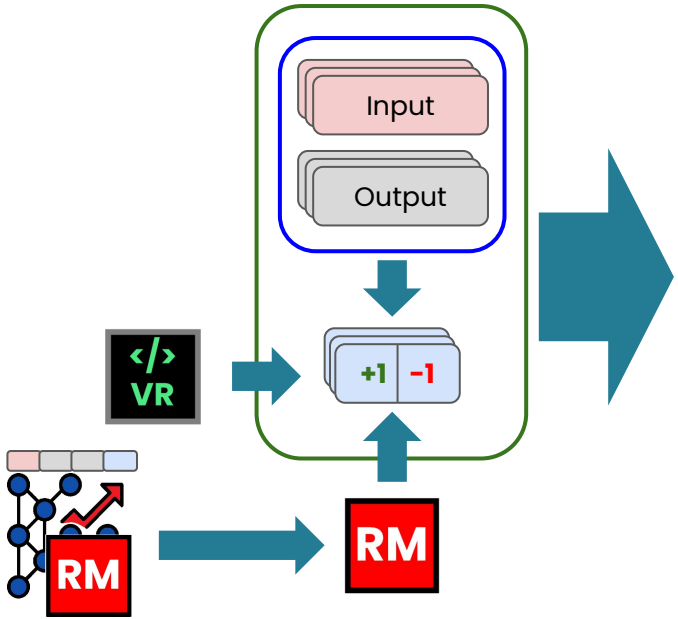


Train LLM on trajectories with RL

Use reference LLM to stabilize measure for probability of next token

$$\frac{\mathbf{LLM}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}{\mathbf{LLM}_{\text{ref}}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}$$

Running RL: Part 1

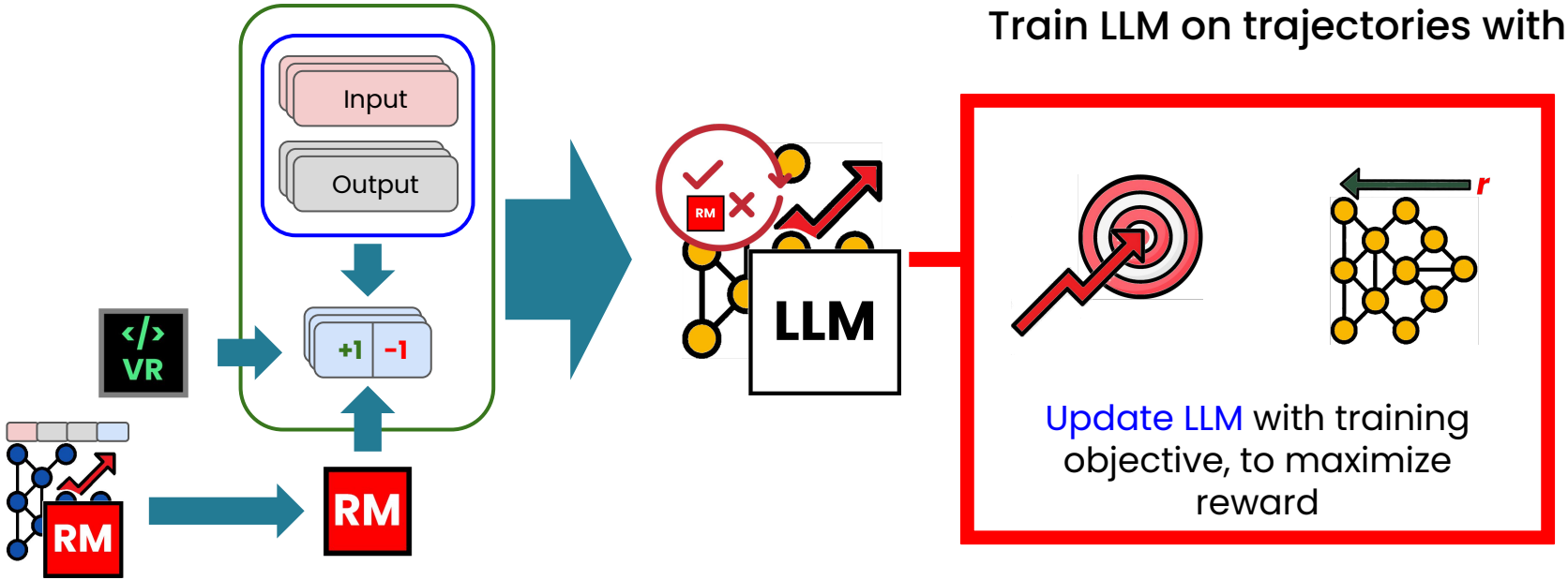


Train LLM on trajectories with RL

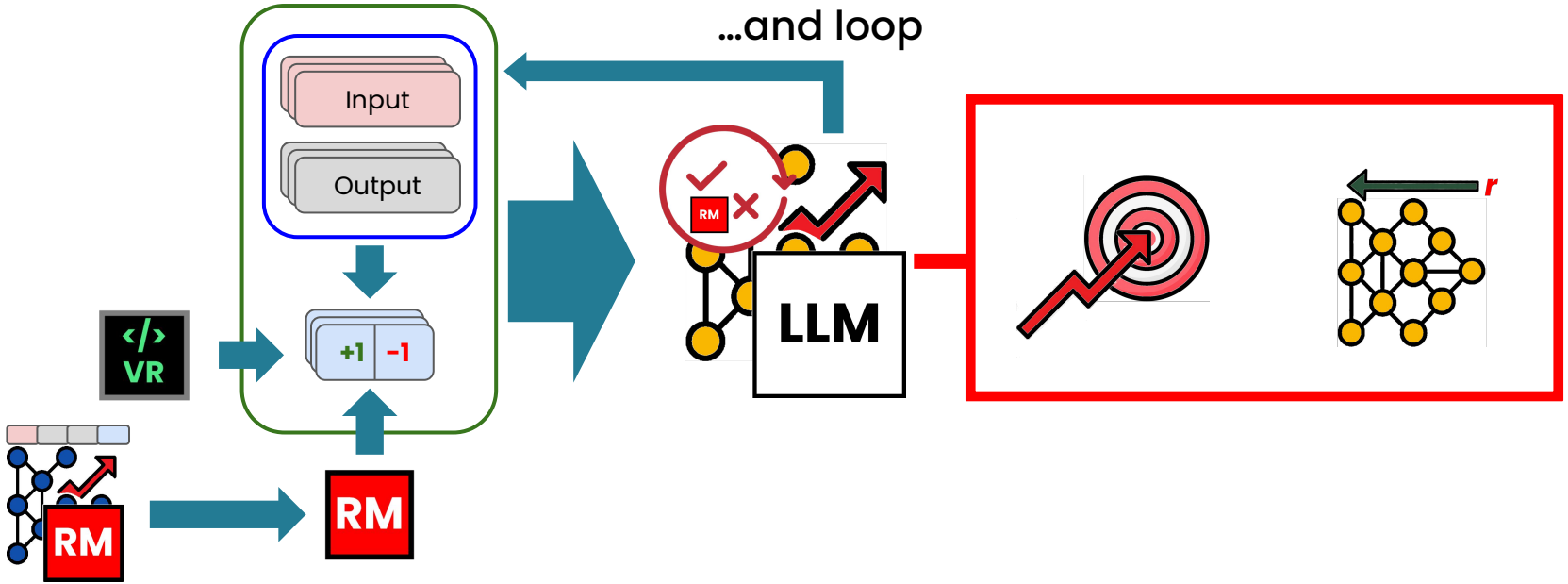
Measure advantage using reward, adjusted by a baseline

$$r\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right) - b\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

Running RL: Part 1



Running RL: Part 1



Running RL: Part 1

RL loop:

1. Get RL data
    a. Get rollouts {input, output} from reference LLM
    b. Apply reward → trajectories {input, output, reward}
        i. In RLHF, train a reward model with preference learning
2. Train your main LLM on trajectories with RL
    a. Define training objective with modern RL algorithms (next video!)
    b. Update LLM with training objective, to maximize reward
3. Continue steps 1+2 until completion

DeepLearning.AI

Core Techniques in Fine-tuning and RL

RL: PPO and GRPO algorithms

Running RL: Part 1

RL loop:

1. Get RL data
   a. Get rollouts {input, output} from reference LLM
   b. Apply reward → trajectories {input, output, reward}
2. Train your main LLM on trajectories with RL
   a. Define training objective with modern RL algorithms
   b. Update LLM with training objective, to maximize reward
3. Continue steps 1+2 until completion

RL training objective

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

$$\frac{\mathbf{LLM}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}{\mathbf{LLM}_{\text{ref}}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)} \qquad \boldsymbol{r}\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right) - b\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

RL training objective

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

Ratio

Advantage

$$\frac{\mathbf{LLM}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}{\mathbf{LLM}_{\text{ref}}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}$$

$$\boldsymbol{r}\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right) - b\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

RL training objective

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

Ratio

$$\frac{\mathbf{LLM}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}{\mathbf{LLM}_{\text{ref}}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}$$

Advantage

$$\boldsymbol{r}\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right) - b\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

How do we compute this?

RL training objective

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

$$b\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right) = \mathbf{LLM}_2\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

We can learn the baseline as a new model to predict expected reward.

Trained with regular fine-tuning by using the real reward **"r"** as the target.

RL training objective

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

$$b\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right) = \mathbf{LLM}_2\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

$$\delta_t = r_t + \gamma b(\boldsymbol{s}_{t+1}) - b(\boldsymbol{s}_t)$$

$$\hat{A}_t^{GAE} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

In practice, use GAE: Generalized Advantage Estimation.

GAE uses the baseline model to smooth and propagate final rewards back into earlier tokens, so they get a discounted share of final reward.

Proximal policy optimization (PPO)

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

- We have *almost* arrived at our first modern RL algorithm.
- Our probability ratio sometimes gets too large. We will adjust our objective to clip this ratio!

$$\sum_{\text{token-indices}} \left(\frac{\mathbf{LLM}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}{\mathbf{LLM}_{\text{ref}}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)} * \left(\boldsymbol{r}\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right) - \mathbf{LLM}_2\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right)\right) \right)$$

Proximal policy optimization (PPO)

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

- We have *almost* arrived at our first modern RL algorithm.
- Our probability ratio sometimes gets too large. We will adjust our objective to clip this ratio!

$$\sum_{\text{token-indices}} \min\left(\frac{\mathbf{LLM}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}{\mathbf{LLM}_{\text{ref}}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)} * \left(\boldsymbol{r}\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right) - \mathbf{LLM}_2\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right)\right), \text{clip}\left(\frac{\mathbf{LLM}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}{\mathbf{LLM}_{\text{ref}}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}, \epsilon, 1-\epsilon\right) * \left(\boldsymbol{r}\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right) - \mathbf{LLM}_2\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right)\right)\right)$$

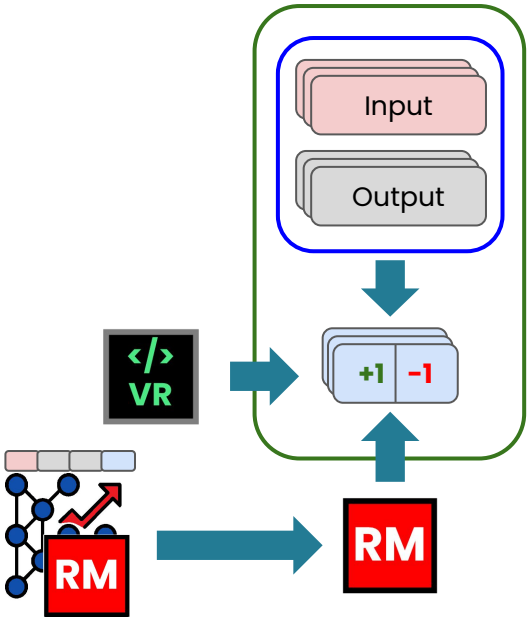
This is PPO!

Running RL: Part 2

RL loop:

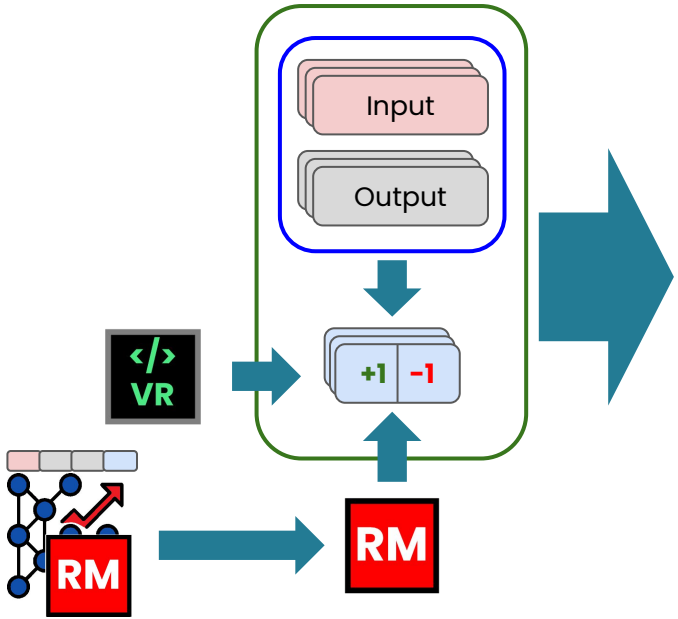
1. Get RL data
   a. Get rollouts {input, output} from reference LLM
   b. Apply reward → trajectories {input, output, reward}
2. Train your main LLM on trajectories with RL
   a. Define PPO objective: Weight probability of next token by advantage…
      i. Add clipping on your probability ratio to stabilize updates
      ii. Train another LLM to estimate the baseline for your advantage
   b. Update LLM using PPO objective, to maximize reward
3. Continue steps 1+2 until completion

Running RL: Part 2



Get RL data
(as usual)

Running RL: Part 2

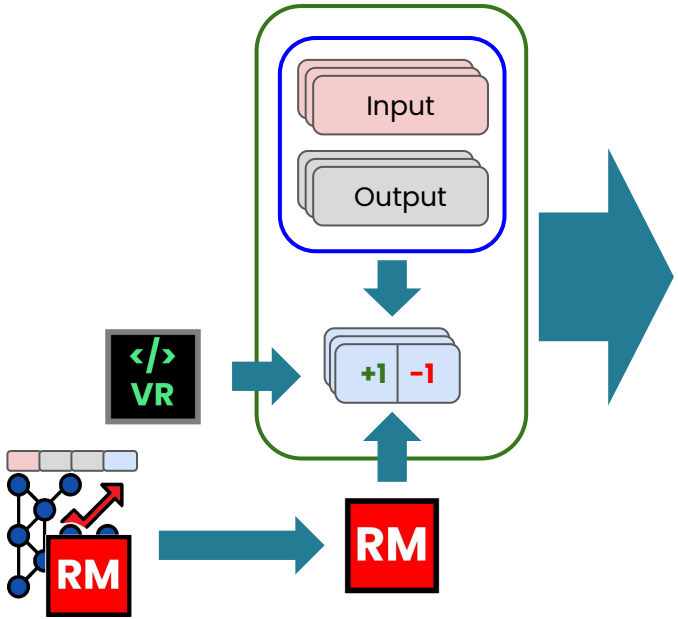


Train LLM on trajectories with RL

Add clipping on your probability ratio to stabilize updates

$$\text{clip}\left(\frac{\mathbf{LLM}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}{\mathbf{LLM}_{\text{ref}}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}, \epsilon, 1-\epsilon\right)$$

Running RL: Part 2

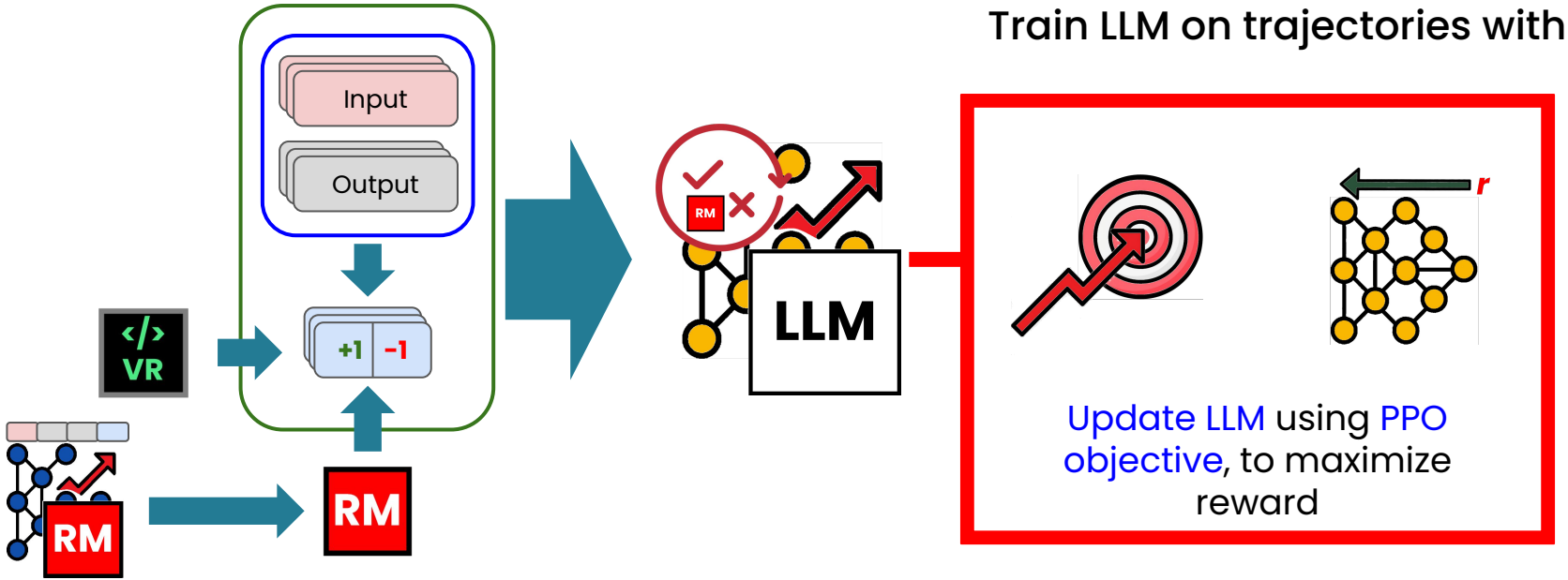


Train LLM on trajectories with RL

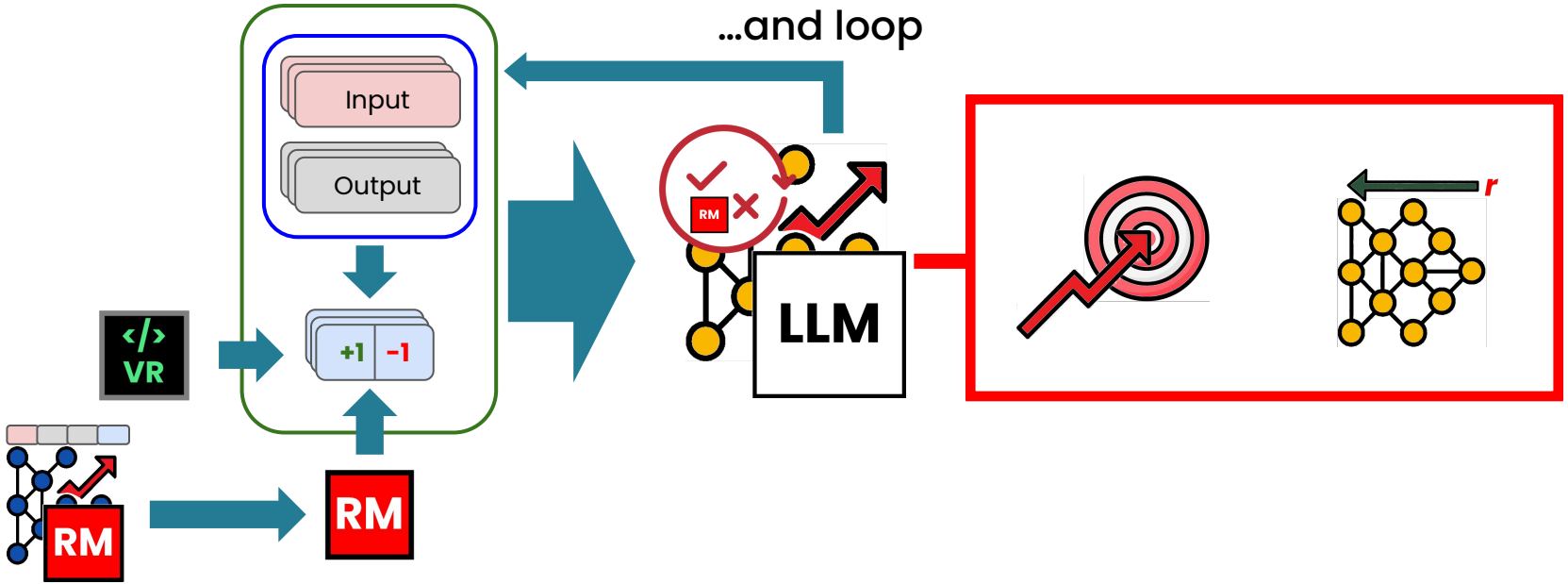
Train another LLM to estimate the baseline for your advantage

$$b\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right) = \mathbf{LLM}_2\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right)$$

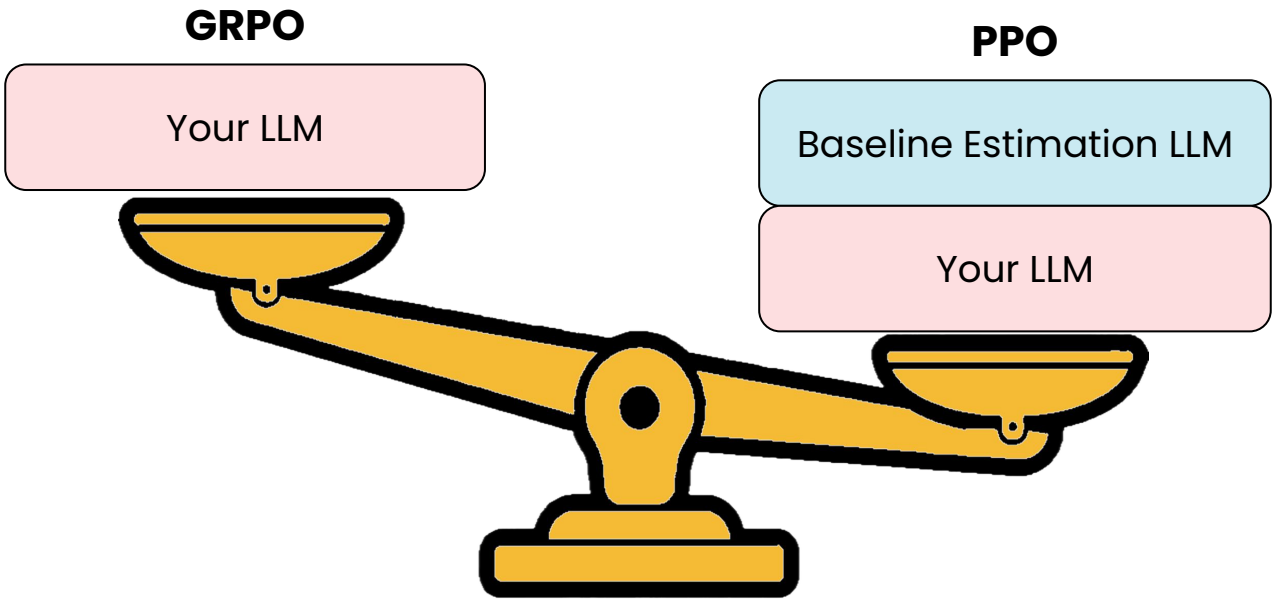
Running RL: Part 2



Running RL: Part 2



PPO limitation



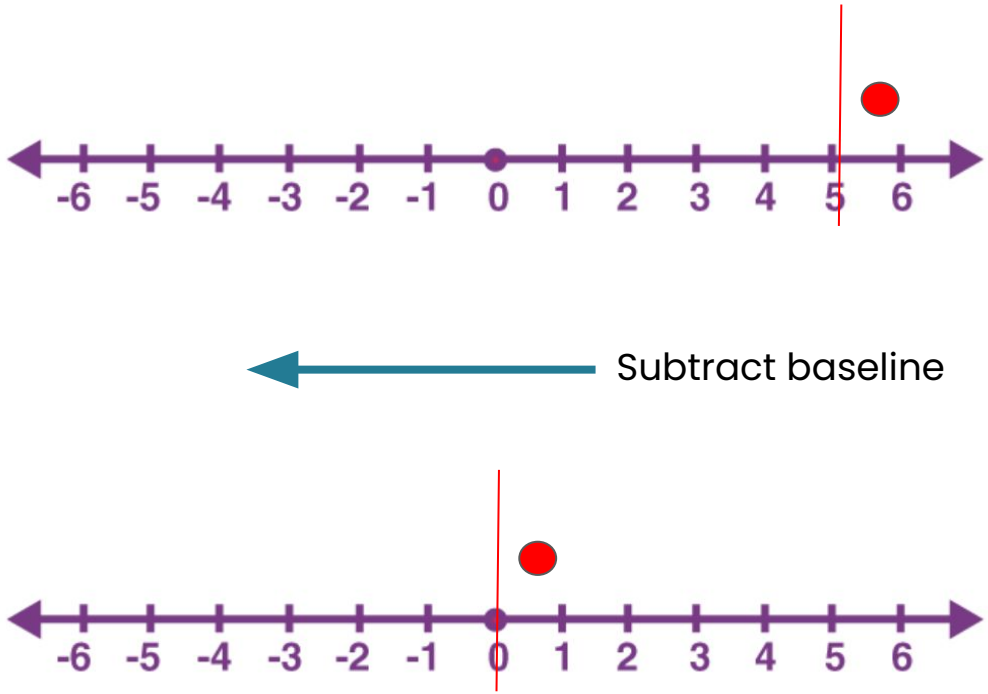
Group relative policy optimization (GRPO)

"GRPO forgoes the critic model [aka. separate baseline estimation model], instead estimating the baseline from group scores, *significantly reducing training resources."*

- DeepSeekMath paper

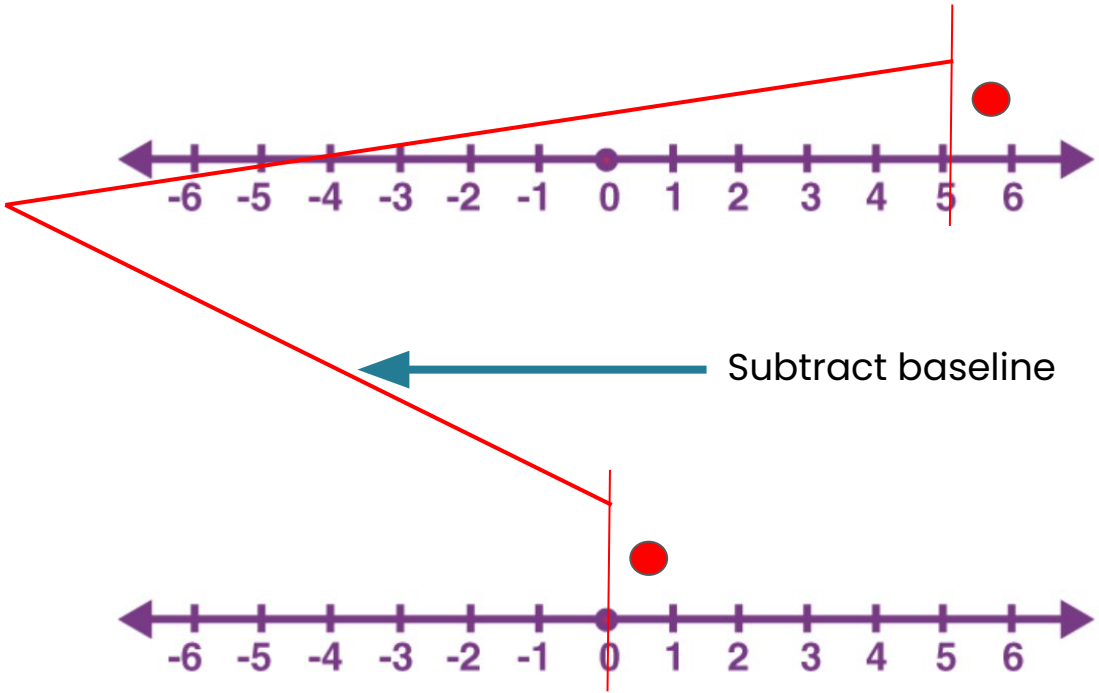
[From "DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models", Shao et al., 2024]

Recall the baseline



Recall the baseline

Instead of training a new model to compute the baseline, why not just...

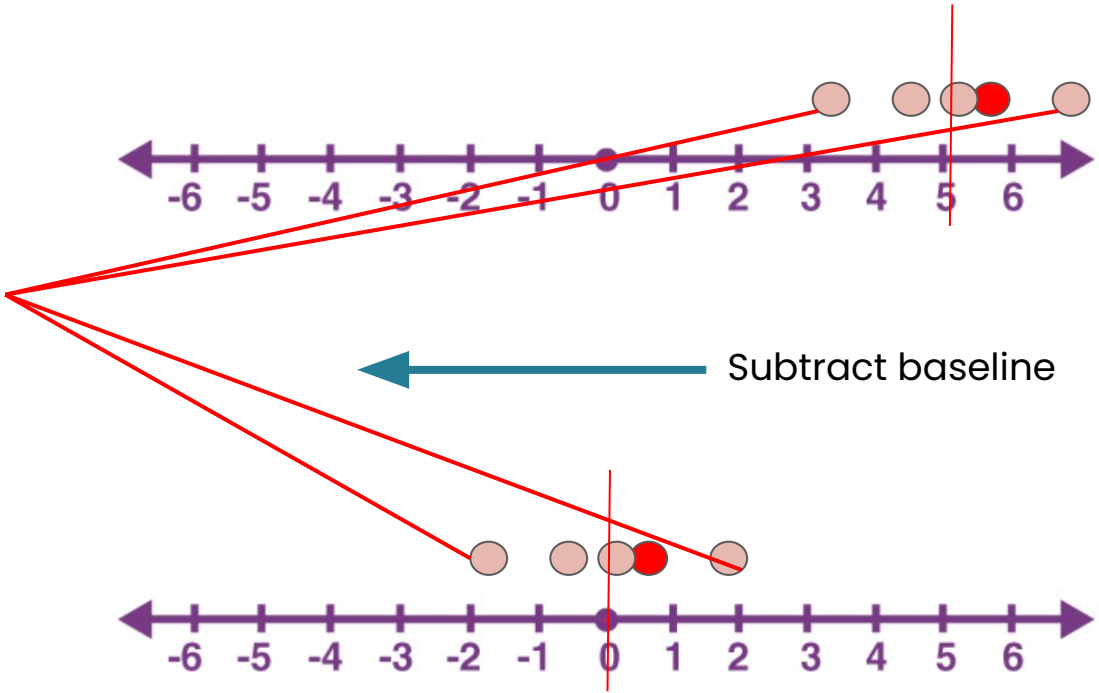


Recall the baseline

Sample many outputs per input.

These are called a "group".

Average the rewards in a group.

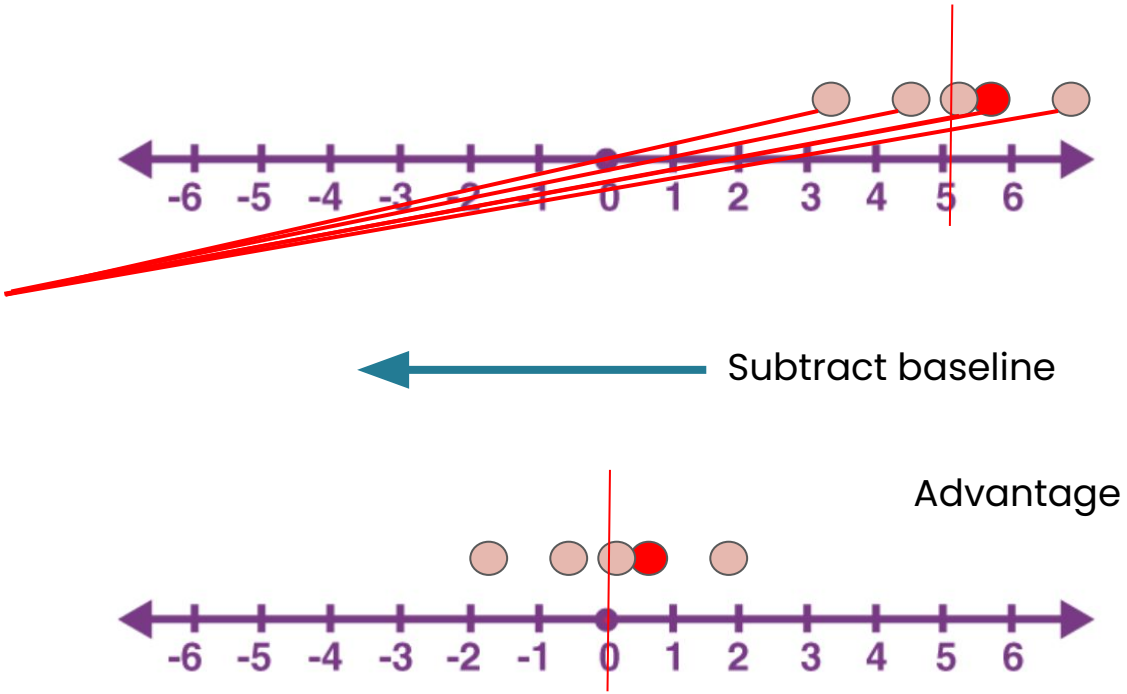


Recall the baseline

Sample many outputs per input.

These are called a "group".

Baseline: Average the rewards in a group.



In your code

Temperature helps you sample different outputs, per input.

```
GRPO_config = GRPOConfig(
    num_generations = 12
    temperature = 0.7)
```

Group Relative Policy Optimization (GRPO)

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

$$\frac{r\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right) - \text{mean}(\text{r's in same group})}{\text{std}(\text{r's in same group})}$$

Group Relative Policy Optimization (GRPO)

$$\sum_{\text{token-indices}} [\text{prob of next token}] * [\text{how good token was}]$$

$$\frac{r\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right) - \text{mean}(\text{r's in same group})}{\text{std}(\text{r's in same group})}$$

```
mean_reward = sum(group_rewards) / len(group_rewards)
normalized_rewards = [r - mean_reward for r in group_rewards]
```

GRPO - example with a group of 4 outputs

Response	Raw Reward
A	8.5
B	6.8
C	4.2
D	2.5

Sample Grader
Python grader
Unit test grader
Formatting grader

GRPO - example with a group of 4 outputs

Response	Raw Reward	Advantage
A	8.5	+1.12
B	6.8	+0.49
C	4.2	-0.49
D	2.5	-1.12

$$\frac{r\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right) - \text{mean(r's in same group)}}{\text{std(r's in same group)}}$$

GRPO - example with a group of 4 outputs

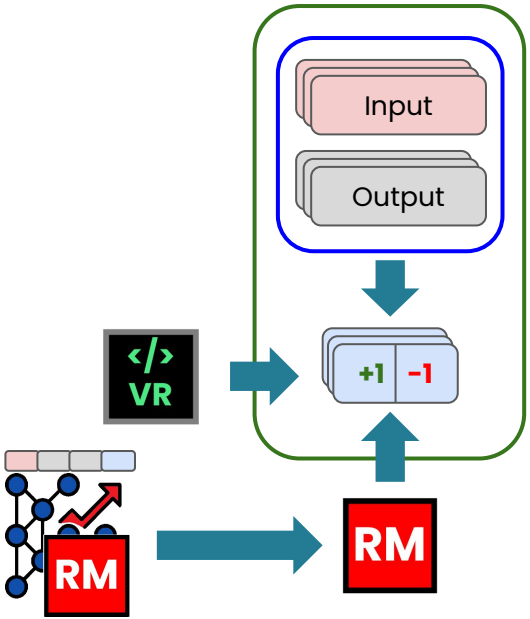
Response	Raw Reward	Advantage	Effect
A	8.5	+1.12	Increase
B	6.8	+0.49	Increase
C	4.2	-0.49	Decrease
D	2.5	-1.12	Decrease

Running RL: Part 2

RL loop:

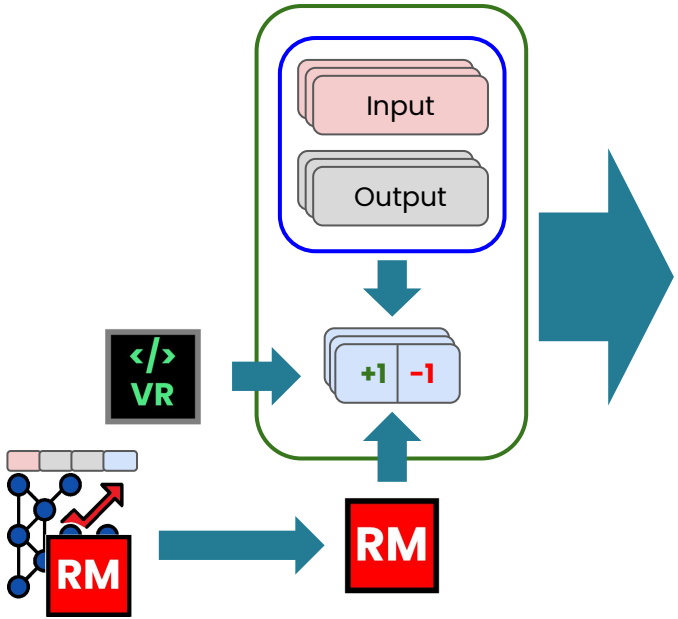
1. Get RL data
    a. Get rollouts {input, output} from reference LLM
    b. Apply reward → trajectories {input, output, reward}
2. Train your main LLM on trajectories with RL
    a. Define GRPO objective: Weight probability of next token by advantage with clipping...
        i. Calculate advantage: Normalize rewards in a group of outputs
    b. Update LLM using GRPO objective, to maximize reward
3. Continue steps 1+2 until completion

Running RL: Part 2



Get RL data

Running RL: Part 2

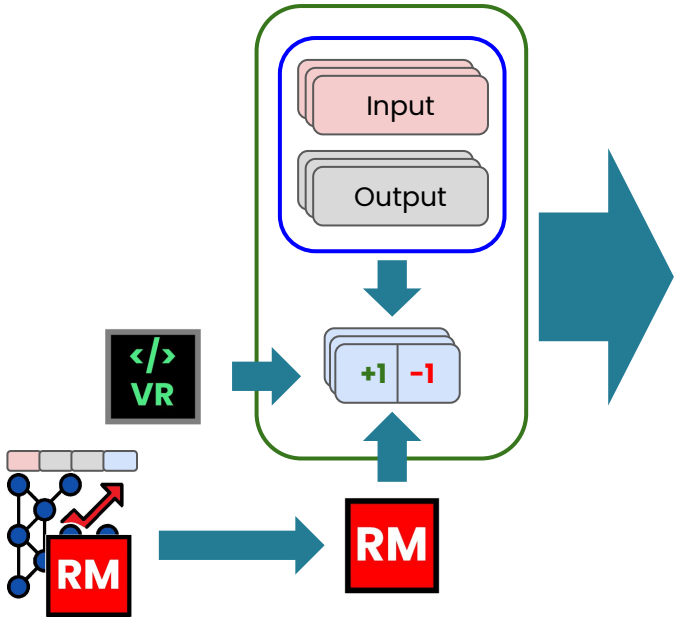


Train LLM on trajectories with RL

GRPO objective: Weight probability of next token by advantage with clipping

$$\text{clip}\left(\frac{\mathbf{LLM}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}{\mathbf{LLM}_{\text{ref}}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)}, \epsilon, 1-\epsilon\right)$$

Running RL: Part 2

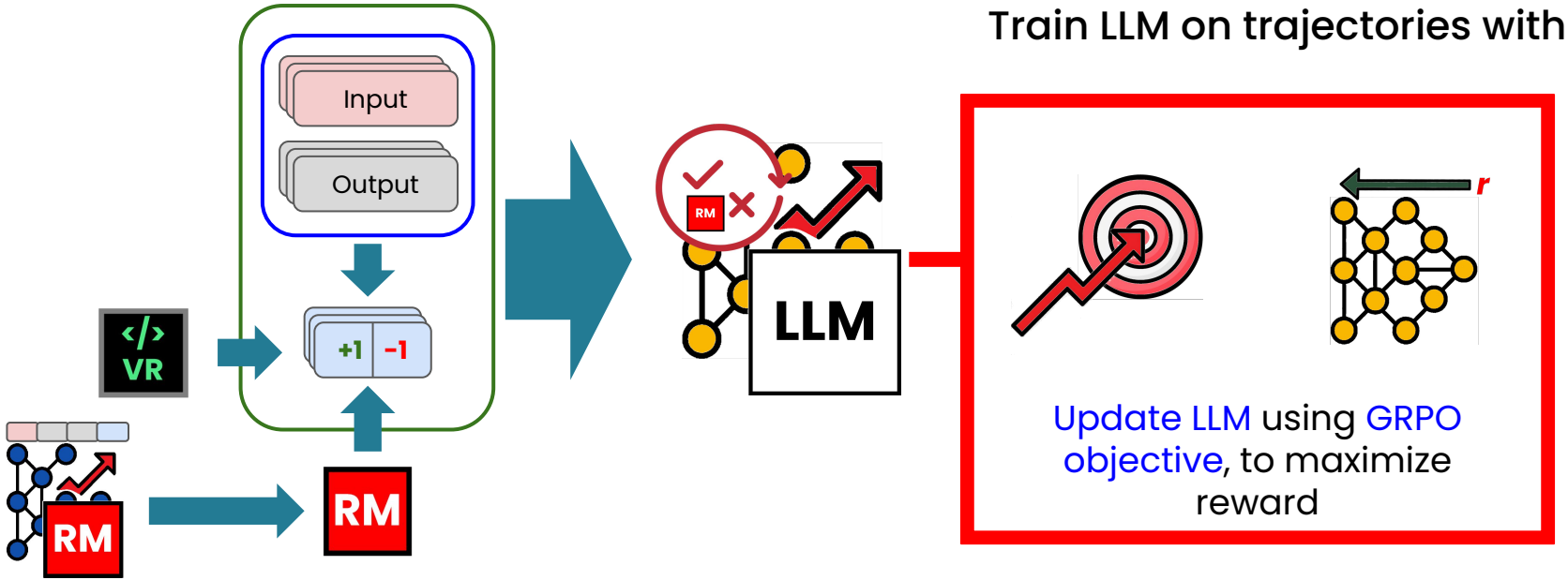


Train LLM on trajectories with RL

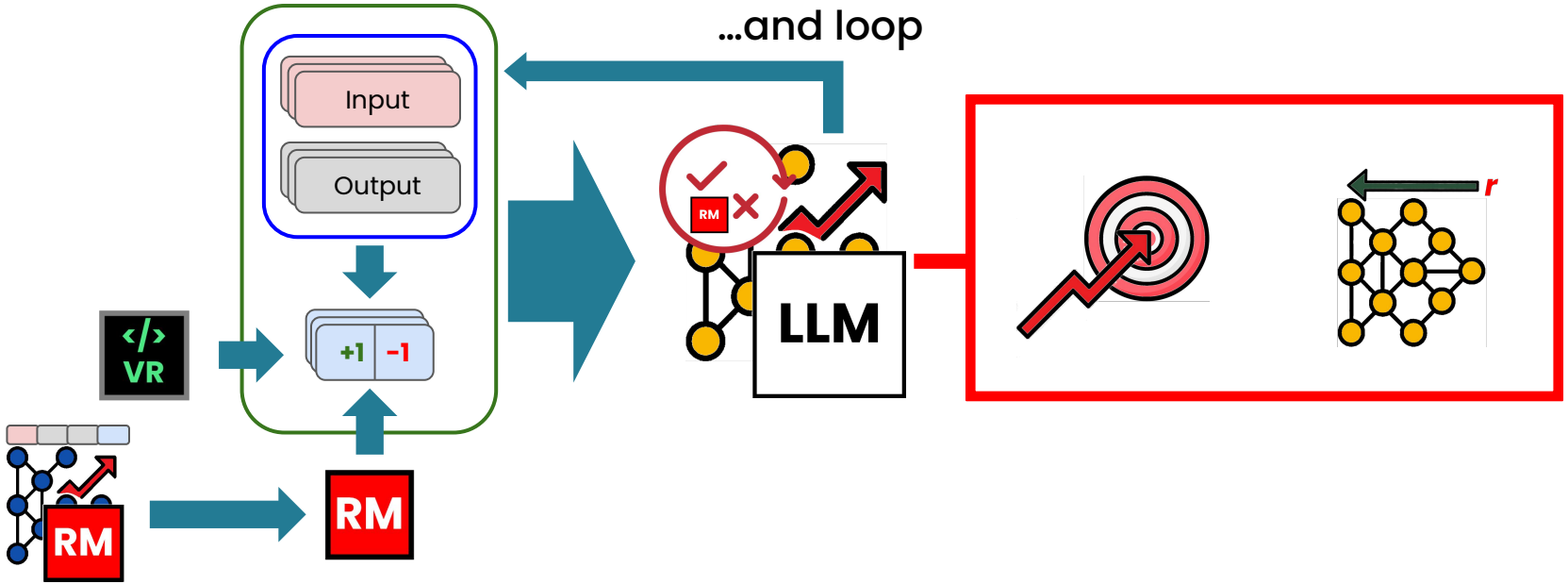
Calculate advantage:
Normalize rewards in a group of outputs

$$\frac{r\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right) - \text{mean(r's in same group)}}{\text{std(r's in same group)}}$$

Running RL: Part 2



Running RL: Part 2



In your code

```
def compute_rewards(prompts, completions):
```

In your code

```
def compute_rewards(prompts, completions):
  for i, unique_prompt in enumerate(unique_prompts):
    prompt_indices = [idx for idx, p in enumerate(prompts) if p == prompt]
    group_completions = [outputs[idx] for idx in prompt_indices]
```

In your code

```
def compute_rewards(prompts, completions):
  for i, unique_prompt in enumerate(unique_prompts):
    prompt_indices = [idx for idx, p in enumerate(prompts) if p == prompt]
    group_completions = [outputs[idx] for idx in prompt_indices]

    for j, out in enumerate(group_outputs):
      reward = reward_model.compute_reward(out, correct_out, prompt)
      group_rewards.append(reward)
```

In your code

```
def compute_rewards(prompts, completions):
  for i, unique_prompt in enumerate(unique_prompts):
    prompt_indices = [idx for idx, p in enumerate(prompts) if p == prompt]
    group_completions = [outputs[idx] for idx in prompt_indices]

    for j, out in enumerate(group_outputs):
      reward = reward_model.compute_reward(out, correct_out, prompt)
      group_rewards.append(reward)

  mean_reward = sum(group_rewards) / len(group_rewards)
  normalized_rewards = [r - mean_reward for r in group_rewards]
```

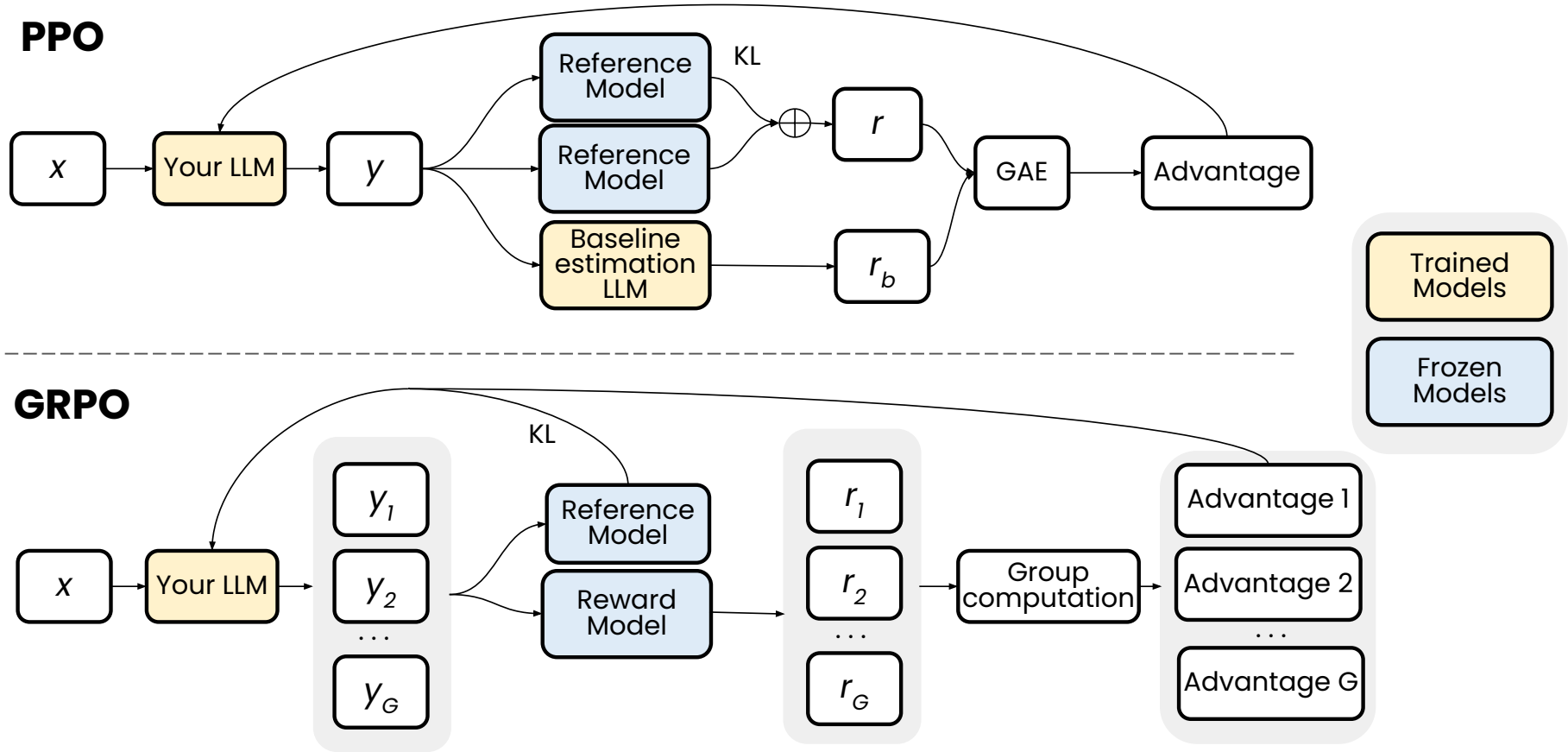
In your code

```
trainer = GRPOTrainer(
    model=model,
    reward_funcs=compute_rewards,
    args=GRPO_config,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    processing_class=tokenizer,
)

trainer.train()
```

GRPO saves you a model, reducing compute

1. Your LLM - learning! $\mathbf{LLM}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)$
2. LLM_{ref} - frozen (inference only), generates rollouts $\mathbf{LLM}_{\text{ref}}\left(y_i \mid \boldsymbol{x}, \boldsymbol{y}_{<i}\right)$
3. Reward Model $\boldsymbol{r}\left(y_i, \boldsymbol{x}, \boldsymbol{y}_{<i}\right)$
4. Baseline Estimation LLM $b\left(\boldsymbol{x}, \boldsymbol{y}_{<i}\right)$ PPO, not GRPO



DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models https://arxiv.org/abs/2402.03300

Overview of RL research

arXiv:2203.02155 (cs)

[Submitted on 4 Mar 2022]

Training language models to follow instructions with human feedback

RLHF & PPO

arXiv:2212.08073 (cs)

[Submitted on 15 Dec 2022]

Constitutional AI: Harmlessness from AI Feedback

RLAIF

arXiv:2402.03300 (cs)

[Submitted on 5 Feb 2024 (v1), last revised 27 Apr 2024 (this version, v3)]

DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models

GRPO

arXiv:2501.12948 (cs)

[Submitted on 22 Jan 2025]

DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning

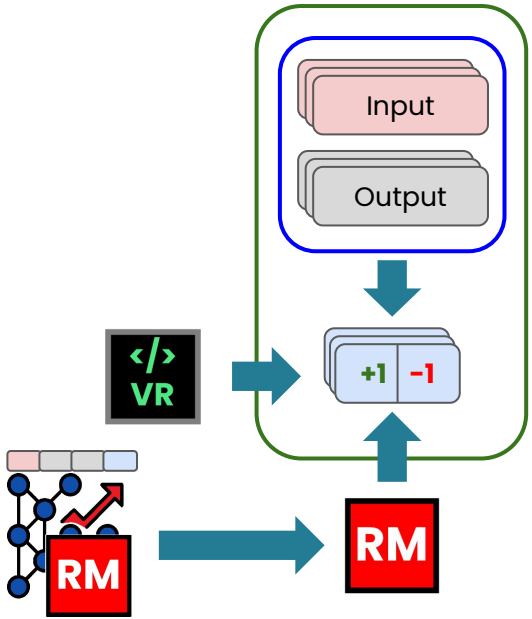
Reasoning with GRPO & verifiable rewards

Running RL

RL loop:

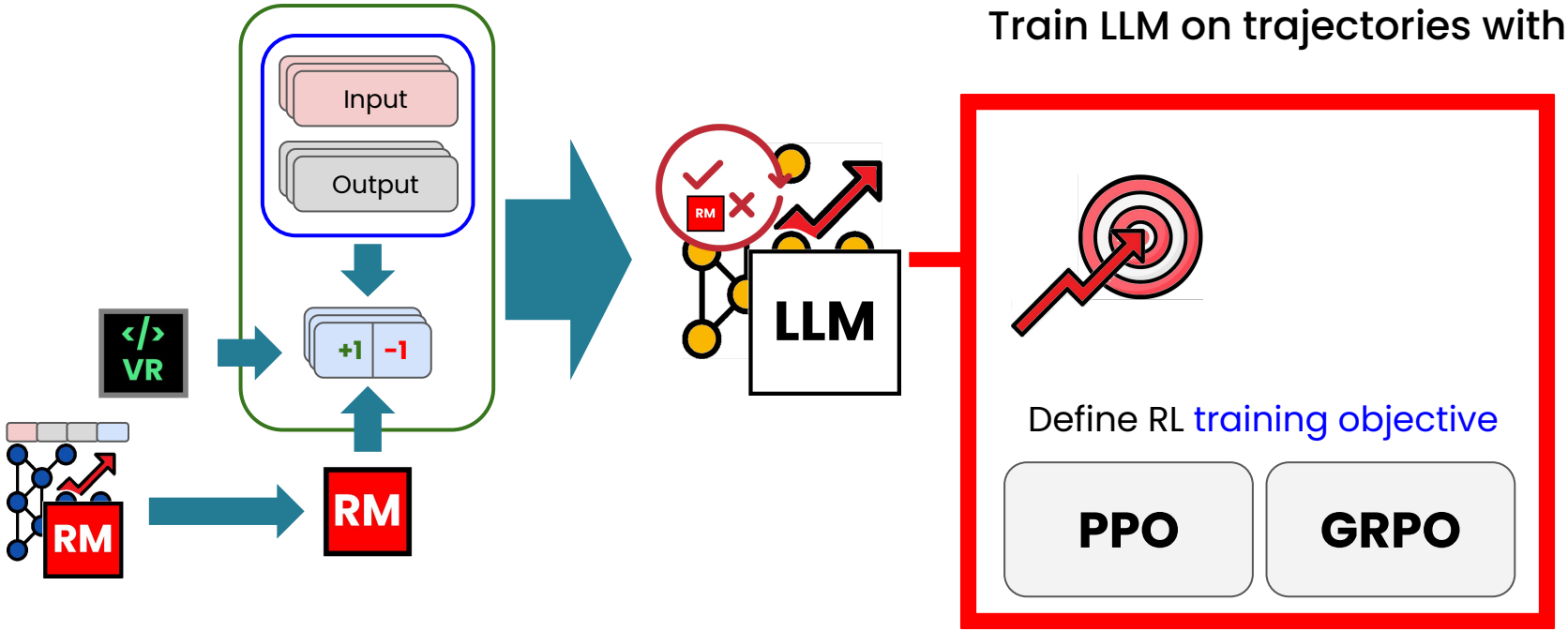
1. Get RL data
   a. Get rollouts {input, output} from reference LLM
   b. Apply reward → trajectories {input, output, reward}
2. Train LLM on trajectories with RL
   a. Define RL training objective, e.g. PPO/GRPO
   b. Update LLM with training objective, to maximize reward
3. Continue steps 1+2 until completion

Running RL

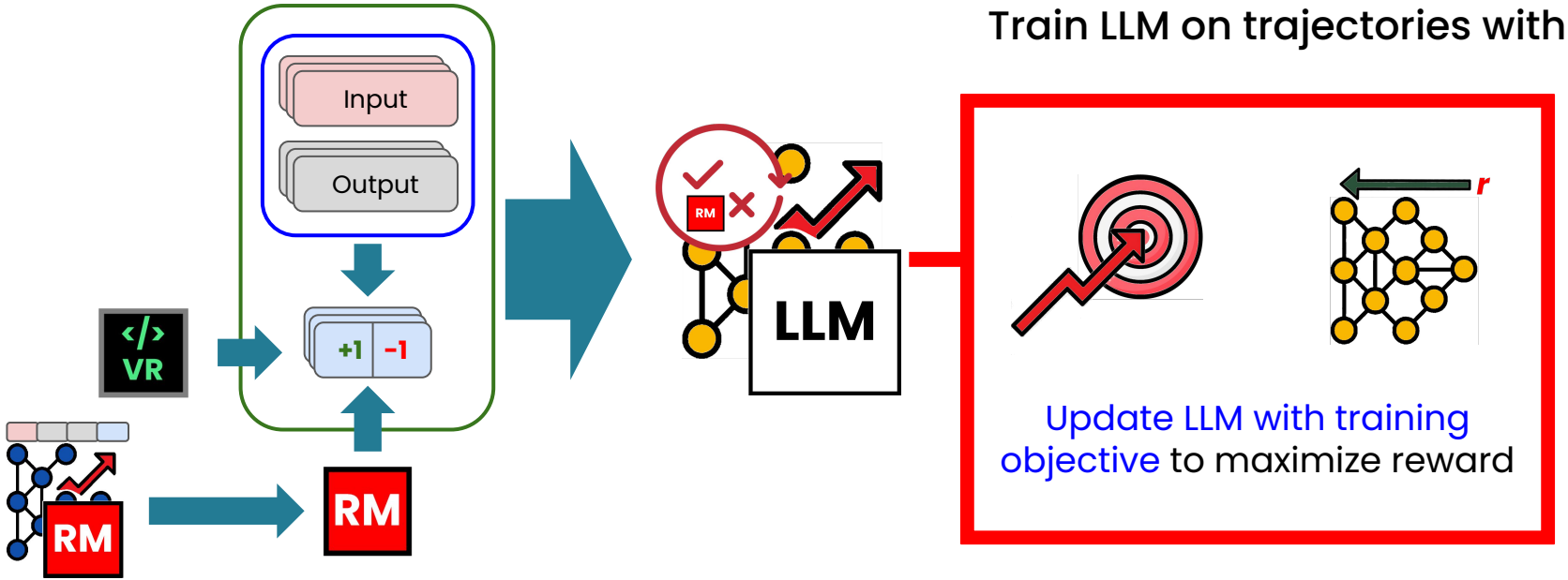


Get RL data

Running RL


Input
Output
+1
-1
</>
VR
RM
RM
RM
LLM
Train LLM on trajectories with RL
Define RL training objective
PPO
GRPO

Running RL


Input
Output
+1
-1
</>
VR
RM
RM
RM
LLM
Train LLM on trajectories with RL
r
Update LLM with training objective to maximize reward

Running RL