Low Rank Adapters (LoRA)

Advanced LLM Night

Johannes Kohlmann – Findable

January 15th, 2025

Who is this guy?

Who is this guy?

Johannes Kohlmann (Me)

Who is this guy?

- Originally from a tiny village in southern Germany

Johannes Kohlmann (Me)

Who is this guy?

- Originally from a tiny village in southern Germany
- Masters degree in Computer Science (ML & theoretical CS focus)

Johannes Kohlmann (Me)

Who is this guy?

- Originally from a tiny village in southern Germany
- Masters degree in Computer Science (ML & theoretical CS focus)
- Moved to Oslo in the beginning of 2024

Johannes Kohlmann (Me)

Who is this guy?

- Originally from a tiny village in southern Germany
- Masters degree in Computer Science (ML & theoretical CS focus)
- Moved to Oslo in the beginning of 2024
- Software Engineer at Findable since June 2024

Johannes Kohlmann (Me)

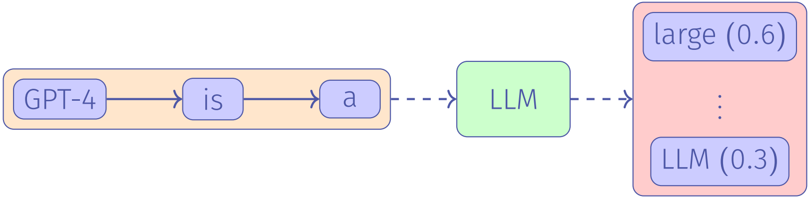
How do modern LLMs work?

How do modern LLMs work?

Modern LLMs predict the most probable word given the input:

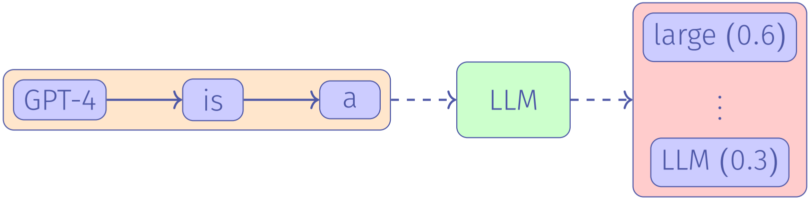
How do modern LLMs work?

Modern LLMs predict the most probable word given the input:



How do modern LLMs work?

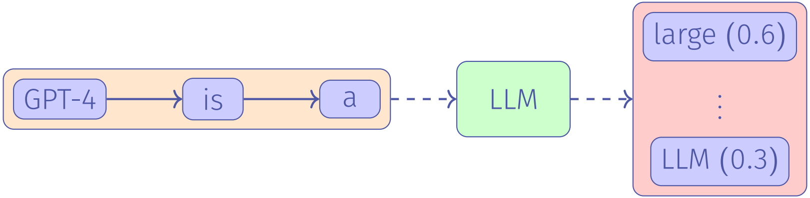
Modern LLMs predict the most probable word given the input:



Sooo.... how do they learn to do that?

How do modern LLMs work?

Modern LLMs predict the most probable word given the input:



Sooo.... how do they learn to do that?

Next Token Prediction

Given a dataset $\mathcal{D}$ of sentences s and target words s_t, minimize the loss $\mathcal{L}$

$$\mathcal{L}(\mathcal{D}) = \sum_{s \in \mathcal{D}} L(\mathrm{LLM}(s), s_t)$$

Pre-training done. Now what?

Pre-training done. Now what?

After pre-training, LLMs are generalists (a.k.a. Foundational Models).

Pre-training done. Now what?

After pre-training, LLMs are generalists (a.k.a. Foundational Models).

However, to utilize them for specific tasks, we need to finetune!

Pre-training done. Now what?

After pre-training, LLMs are generalists (a.k.a. Foundational Models).

However, to utilize them for specific tasks, we need to finetune!

Finetuning

Finetuning specializes a pre-trained model for a specific task. This is often done by changing/adding weights.

Pre-training done. Now what?

After pre-training, LLMs are generalists (a.k.a. Foundational Models).

However, to utilize them for specific tasks, we need to finetune!

Finetuning

Finetuning specializes a pre-trained model for a specific task. This is often done by changing/adding weights.

There is a variety of approaches to finetuning:

Pre-training done. Now what?

After pre-training, LLMs are generalists (a.k.a. Foundational Models).

However, to utilize them for specific tasks, we need to finetune!

Finetuning

Finetuning specializes a pre-trained model for a specific task. This is often done by changing/adding weights.

There is a variety of approaches to finetuning:

- Full Model Finetuning (Expensive, Hard to maintain)

Pre-training done. Now what?

After pre-training, LLMs are generalists (a.k.a. Foundational Models).

However, to utilize them for specific tasks, we need to finetune!

Finetuning

Finetuning specializes a pre-trained model for a specific task. This is often done by changing/adding weights.

There is a variety of approaches to finetuning:

- Full Model Finetuning (Expensive, Hard to maintain)
- Additional Adapter Layers (Limited capacity, Inference latency)

Pre-training done. Now what?

After pre-training, LLMs are generalists (a.k.a. Foundational Models).

However, to utilize them for specific tasks, we need to finetune!

Finetuning

Finetuning specializes a pre-trained model for a specific task. This is often done by changing/adding weights.

There is a variety of approaches to finetuning:

- Full Model Finetuning (Expensive, Hard to maintain)
- Additional Adapter Layers (Limited capacity, Inference latency)
- Prompt Optimization (Limited capacity & generalization)

Pre-training done. Now what?

After pre-training, LLMs are generalists (a.k.a. Foundational Models).

However, to utilize them for specific tasks, we need to finetune!

Finetuning

Finetuning specializes a pre-trained model for a specific task. This is often done by changing/adding weights.

There is a variety of approaches to finetuning:

- Full Model Finetuning (Expensive, Hard to maintain)
- Additional Adapter Layers (Limited capacity, Inference latency)
- Prompt Optimization (Limited capacity & generalization)

Sooo... what now?

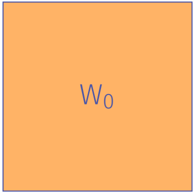
Full Model Finetuning

Full Model Finetuning

During full model finetuning, we do small updates to all weights of the model:

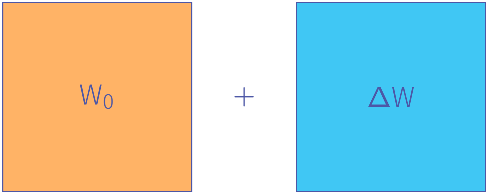
Full Model Finetuning

During full model finetuning, we do small updates to all weights of the model:



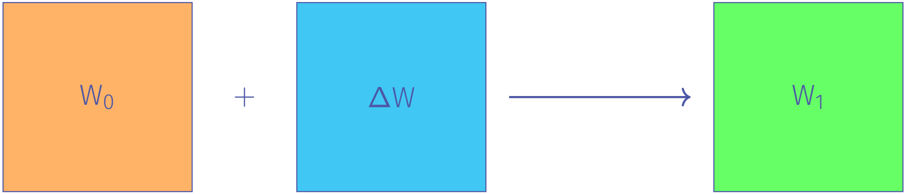
Full Model Finetuning

During full model finetuning, we do small updates to all weights of the model:



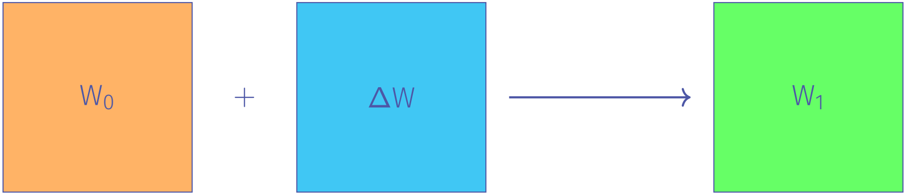
Full Model Finetuning

During full model finetuning, we do small updates to all weights of the model:



Full Model Finetuning

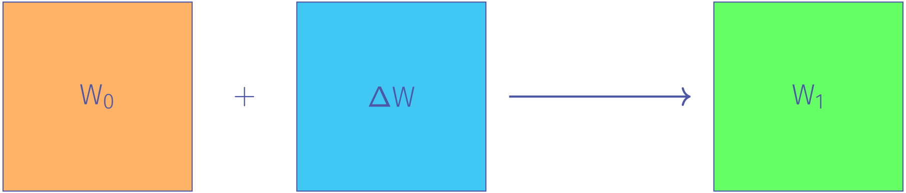
During full model finetuning, we do small updates to all weights of the model:



Therefore, finetuning a single fully-connected layer has the same complexity as learning it from scratch!

Full Model Finetuning

During full model finetuning, we do small updates to all weights of the model:



Therefore, finetuning a single fully-connected layer has the same complexity as learning it from scratch!

So, doing this for all layers makes finetuning as compute-intensive as the original pre-training!

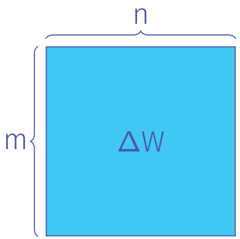
A little bit of math

A little bit of math

How can we reduce the number of trainable parameters during finetuning?

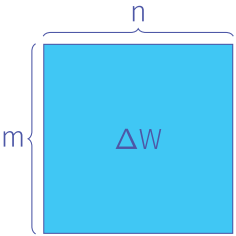
A little bit of math

How can we reduce the number of trainable parameters during finetuning?



A little bit of math

How can we reduce the number of trainable parameters during finetuning?

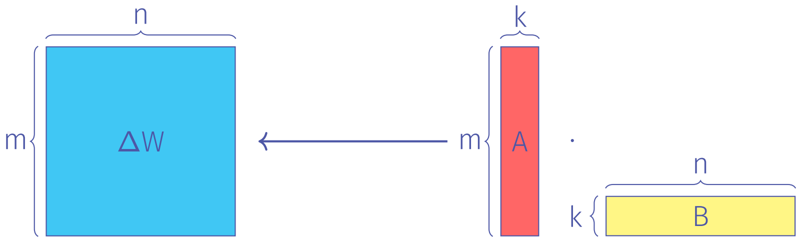


Matrix multiplication ”eliminates” the ”inner” dimension:

$$(m \times n) \longleftarrow (m \times k) \cdot (k \times n)$$

A little bit of math

How can we reduce the number of trainable parameters during finetuning?

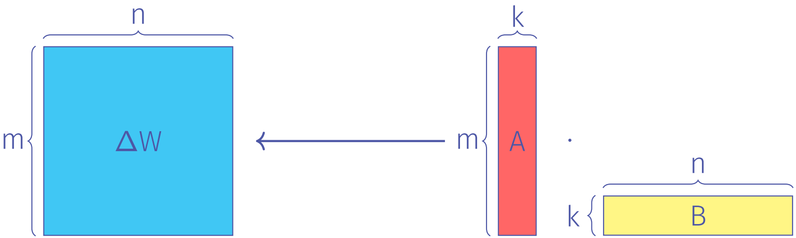


Matrix multiplication ”eliminates” the ”inner” dimension:

$$(m \times n) \longleftarrow (m \times k) \cdot (k \times n)$$

A little bit of math

How can we reduce the number of trainable parameters during finetuning?



Matrix multiplication ”eliminates” the ”inner” dimension:

$$(m \times n) \longleftarrow (m \times k) \cdot (k \times n)$$

Therefore, we can represent ΔW by two much smaller matrices!

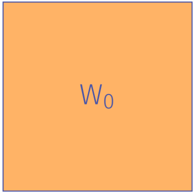
LoRA for efficient Finetuning

LoRA for efficient Finetuning

Low-rank Adapters (LoRA) make use of this trick:

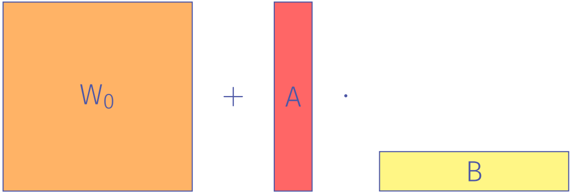
LoRA for efficient Finetuning

Low-rank Adapters (LoRA) make use of this trick:



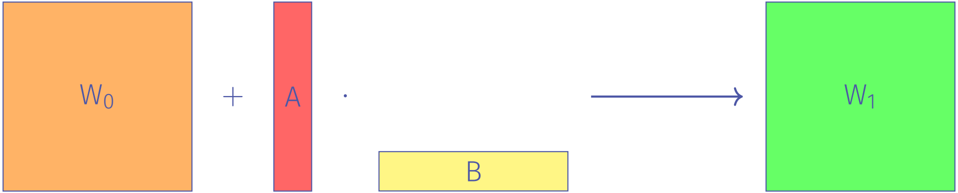
LoRA for efficient Finetuning

Low-rank Adapters (LoRA) make use of this trick:



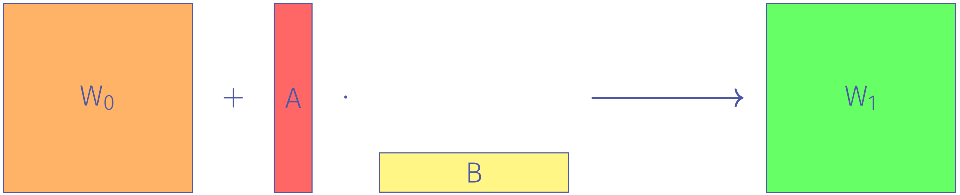
LoRA for efficient Finetuning

Low-rank Adapters (LoRA) make use of this trick:



LoRA for efficient Finetuning

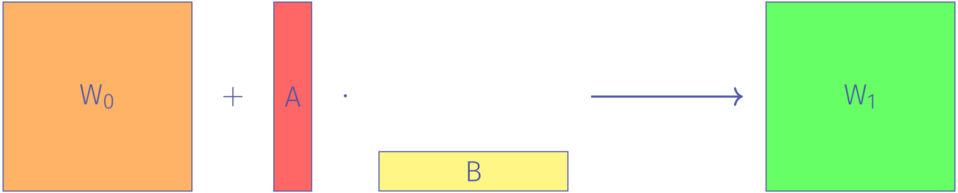
Low-rank Adapters (LoRA) make use of this trick:



Or a little more mathy:

LoRA for efficient Finetuning

Low-rank Adapters (LoRA) make use of this trick:

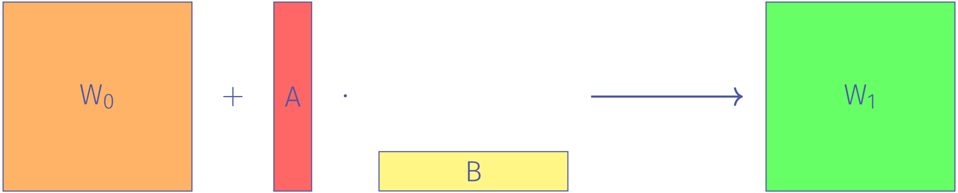


Or a little more mathy:

$$W_0$$

LoRA for efficient Finetuning

Low-rank Adapters (LoRA) make use of this trick:

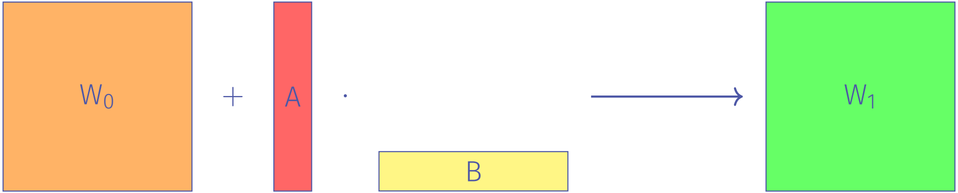


Or a little more mathy:

$$W_0 + A \cdot B$$

LoRA for efficient Finetuning

Low-rank Adapters (LoRA) make use of this trick:

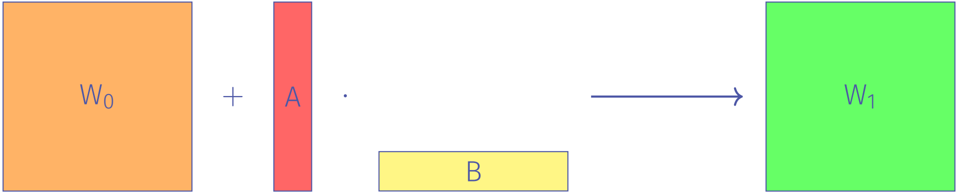


Or a little more mathy:

$$W_0 + A \cdot B \approx W_0 + \Delta W$$

LoRA for efficient Finetuning

Low-rank Adapters (LoRA) make use of this trick:

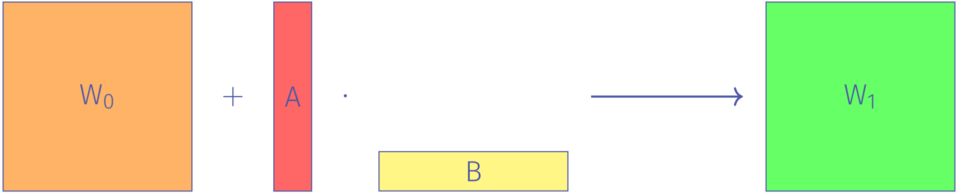


Or a little more mathy:

$$W_0 + A \cdot B \approx W_0 + \Delta W \longrightarrow W_1$$

LoRA for efficient Finetuning

Low-rank Adapters (LoRA) make use of this trick:



Or a little more mathy:

$$W_0 + A \cdot B \approx W_0 + \Delta W \longrightarrow W_1$$

Parameter reduction from $m \cdot n$ to $k \cdot (m + n)$ which is very significant for $m, n \gg k$!

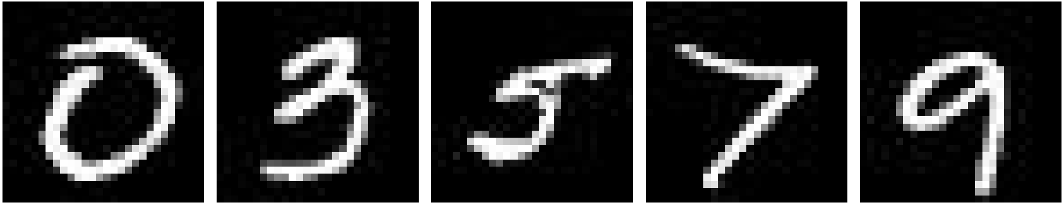
Classifying MNIST

Classifying MNIST

Imagine, we are tasked to recognize hand-written digits:

Classifying MNIST

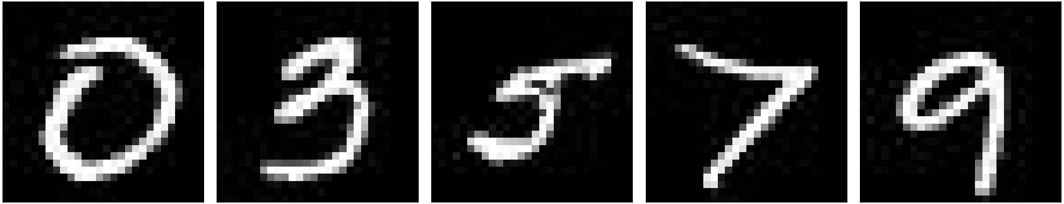
Imagine, we are tasked to recognize hand-written digits:



Example images from the MNIST dataset

Classifying MNIST

Imagine, we are tasked to recognize hand-written digits:



Example images from the MNIST dataset

Task: Given an input image, determine the digit this image depicts.

Model Architecture

Model Architecture

We'll us a simple, fully-connected model:

Model Architecture

We'll us a simple, fully-connected model:

```
class SimpleMnistClassifier(nn.Module):
    def __init__(self):
        super(SimpleMnistClassifier, self).__init__()
        self._fc1 = nn.Linear(784, 256) # 28 * 28 = 784
        self._fc2 = nn.Linear(256, 128)
        self._fc3 = nn.Linear(128, 10)

    def forward(x: torch.Tensor) -> torch.Tensor:
        # Uses ReLU and returns logits
```

Model Architecture

We'll us a simple, fully-connected model:

```python
class SimpleMnistClassifier(nn.Module):
    def __init__(self):
        super(SimpleMnistClassifier, self).__init__()
        self._fc1 = nn.Linear(784, 256) # 28 * 28 = 784
        self._fc2 = nn.Linear(256, 128)
        self._fc3 = nn.Linear(128, 10)

    def forward(x: torch.Tensor) -> torch.Tensor:
        # Uses ReLU and returns logits
```

This model has 235.146 trainable parameters and reaches a test accuracy of 97.78%.

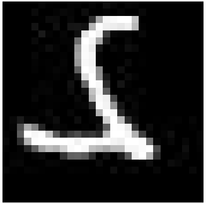
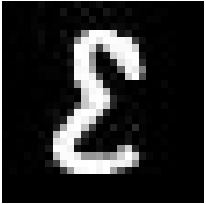
Flipped MNIST

Flipped MNIST

It turns out, that we did not train on exactly the right data:

Flipped MNIST

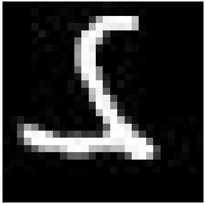
It turns out, that we did not train on exactly the right data:

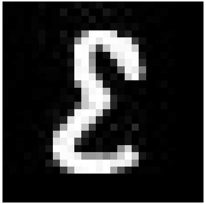



Flipped-MNIST (F-MNIST) has 30% of its two's and three's flipped

Flipped MNIST

It turns out, that we did not train on exactly the right data:





Flipped-MNIST (F-MNIST) has 30% of its two's and three's flipped

How does our model perform on this data? Is it going to be able to generalize by itself?

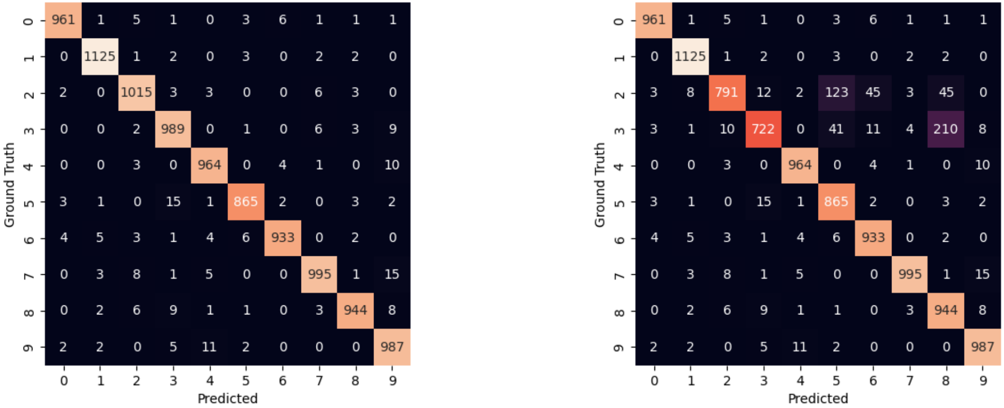
Evaluation on F-MNIST

Evaluation on F-MNIST

Evaluating our model on F-MNIST yields a test accuracy of 92.87%.

Evaluation on F-MNIST

Evaluating our model on F-MNIST yields a test accuracy of 92.87%.



Confusion matrices for MNIST **(left)** and F-MNIST **(right)**

SALVAGING OUR EFFORT WITH LORA

Salvaging our effort with LoRA

We can define a LoRA layer as follows:

Salvaging our effort with LoRA

We can define a LoRA layer as follows:

```
class LoRALayer(nn.Module):
    def __init__(self, inner: nn.Linear, k: int):
        self._inner = inner
        self._inner.requires_grad_(False)
```

Salvaging our effort with LoRA

We can define a LoRA layer as follows:

```
class LoRALayer(nn.Module):
    def __init__(self, inner: nn.Linear, k: int):
        self._inner = inner
        self._inner.requires_grad_(False)
        n, m = inner.in_features, inner.out_features
        self._A = nn.Parameter(torch.randn(n, k))
        self._B = nn.Parameter(torch.zeros(k, m))
```

Salvaging our effort with LoRA

We can define a LoRA layer as follows:

```
class LoRALayer(nn.Module):
    def __init__(self, inner: nn.Linear, k: int):
        self._inner = inner
        self._inner.requires_grad_(False)
        n, m = inner.in_features, inner.out_features
        self._A = nn.Parameter(torch.randn(n, k))
        self._B = nn.Parameter(torch.zeros(k, m))

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self._inner(x) + x @ self._A @ self._B
```

Results with LoRA

Results with LoRA

Augmenting the linear layers with LoRA ($k = 4$) yields a model with only

$$(784 \cdot 4 + 4 \cdot 256) + (256 \cdot 4 + 4 \cdot 128) + (128 \cdot 4 + 4 \cdot 10) = 6248$$

trainable parameters.

Results with LoRA

Augmenting the linear layers with LoRA ($k = 4$) yields a model with only

$$(784 \cdot 4 + 4 \cdot 256) + (256 \cdot 4 + 4 \cdot 128) + (128 \cdot 4 + 4 \cdot 10) = 6248$$

trainable parameters. That is only 3% of the original parameters!

Results with LoRA

Augmenting the linear layers with LoRA ($k = 4$) yields a model with only

$$(784 \cdot 4 + 4 \cdot 256) + (256 \cdot 4 + 4 \cdot 128) + (128 \cdot 4 + 4 \cdot 10) = 6248$$

trainable parameters. That is only 3% of the original parameters!

The finetuned model reaches a test accuracy of 97.74%, i.e. it is as good as the original model was on MNIST!

Results with LoRA

Augmenting the linear layers with LoRA ($k = 4$) yields a model with only

$$(784 \cdot 4 + 4 \cdot 256) + (256 \cdot 4 + 4 \cdot 128) + (128 \cdot 4 + 4 \cdot 10) = 6248$$

trainable parameters. That is only 3% of the original parameters!

The finetuned model reaches a test accuracy of 97.74%, i.e. it is as good as the original model was on MNIST!

Imagine doing this with GPT3-175B! Checkpoint sizes are reduced by 99.99%, finetuning is sped up by 25% and you only need a third of the GPUs to train!

Conclusion

- Based on the LoRA paper by Hu et al.
- Code and slides will be available online.
- Thank you for listening! Questions?