Refiners

Introducing 'Refiners' – A Micro-Framework for Seamless Integration of Adapters in Neural Networks

Benjamin Trom ML @ Finegrain

FOSDEM - 4th February 2024

Evolution of Deep Learning

0 - Statistical Modeling

Problems were solved with mathematical models and statistics based on insights and patterns observed in the data.

1 - Native Deep learning

For every unique task, a new dataset was curated and a model was trained from scratch.

2 - Transfer Learning

Even with smaller datasets, effective models could be developed by transferring knowledge.

3 - Foundational Models

With the invention of Transformers, it was possible to train massive models on massive datasets, e.g. Large Language Models

∞ - AGI

Every single task can be solved in zero-shot, i.e. without training.

AGI

Every single task can be solved in zero-shot, i.e. without training.

AGI

Every single task can be solved in zero-shot, i.e. without training.

=

We all become Unemployed Engineers

In the meantime...

You can either rely on

Prompt Engineering

- ✅ Do not require GPUs or vast amount of data
- ✅ Very practical for fast, iterative problem solving
- ❌ Limited capabilities, highly dependent on foundation model capabilities

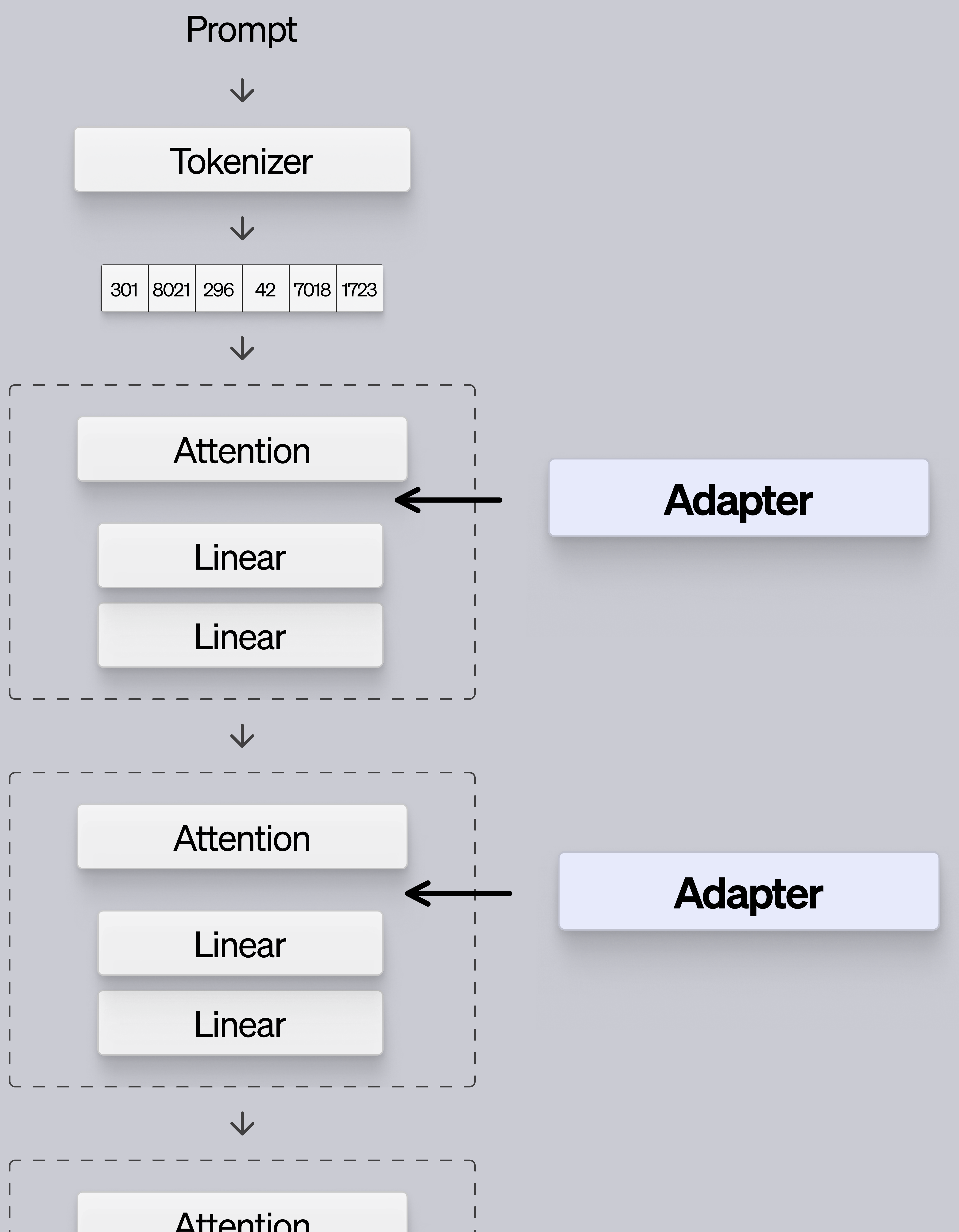
Train Foundation Models

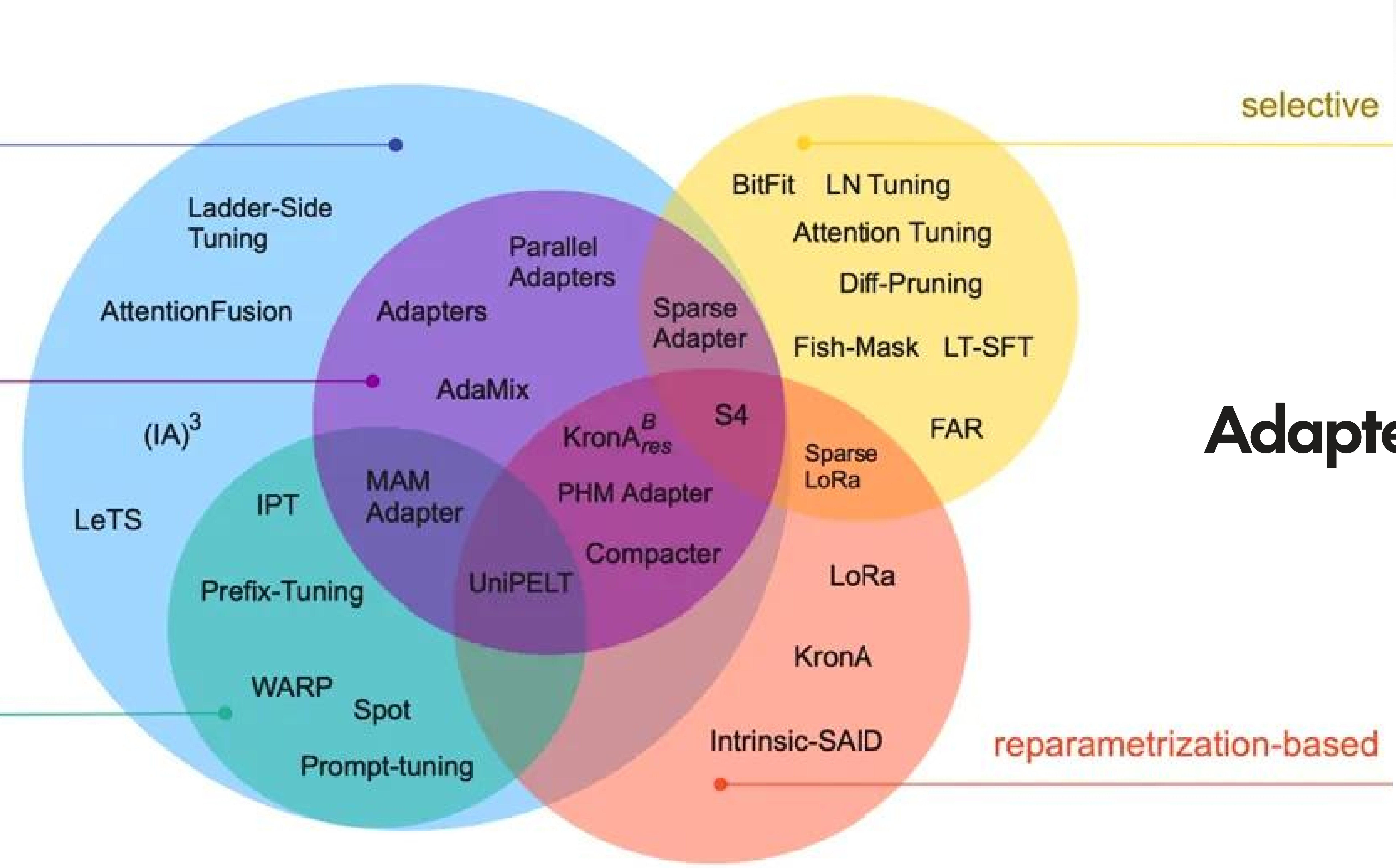
- ✅ Very good bragging material
- ❌ Requires amounts of data and GPUs inaccessible to most individuals, small companies or research labs
- ❌ Very risky: no guarantee that it will solve the actual problem you may want it for

The third way: Adapters

Adaptation is the idea of patching existing powerful models to implement new capabilities

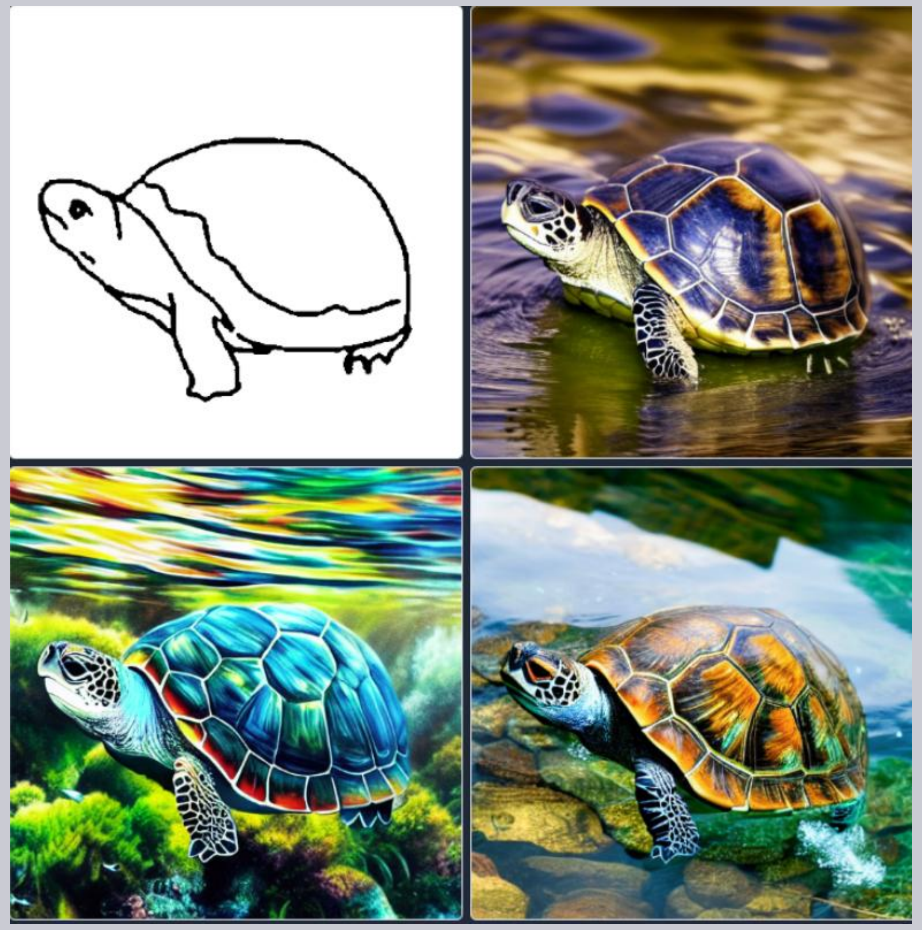
- ✅ Parameter efficient: train with smaller GPUs, less data, and more rapidly.
- ✅ Flexible and composable: you can train multiple adapters and use them together
- ✅ Can extend a foundation model capabilities outside of its training data and even add new modalities.
- ✅ Still a good bragging material 😏





Adapters for LLMs

source: https://medium.com/@shivansh.kaushik/efficient-model-fine-tuning-for-llms-understanding-peft-by-implementation-fc4d5e985389




Adapters for Image Generation

→ ControlNet

→ T2I-Adapter

→ IP-Adapter

→ StyleAligned

→ InstantID

→ ... and many more, with a 2+/week rate for new papers coming out

Imperative code is hard to patch cleanly

There are several ways to patch a foundation model implemented in PyTorch:

- ❌ Just duplicate the original codedase and edit it in place
- ❌ Refactor the entire codebase to optionally support the adapter.
- ❌ Monkey patch 🙈

```python
class BasicModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(1, channels, kernel_size)
        self.linear_1 = nn.Linear(hidden_layer_in, hidden_layer_out)
        self.maxpool = nn.MaxPool2d(2)
        self.linear_2 = nn.Linear(hidden_layer_out, output_size)

    def forward(self, x):
        x = self.conv(x)
        x = nn.functional.relu(x)
        x = self.maxpool(x)
        x = x.flatten(start_dim=1)
        x = self.linear_1(x)
        x = nn.functional.relu(x)
        x = self.linear_2(x)
        return nn.functional.softmax(x, dim=0)
```

So, we wrote (yet another) machine learning framework?

We wrote a machine learning micro-framework.

Introducing Refiners

a declarative machine learning library built on top of PyTorch

Chain

Python class to implement models as trees of layers.

Context

Simplify to flow of data by providing a stateful store to Chains.

Adapter

Tool to simplify “model surgery” required to patch models.

Chain

- ✅ Python class to implement models as trees of layers in a declarative manner.
- ✅ WYSIWYG: if look at the representation of the model in the REPL, you know exactly what it does.
- ✅ Contains a lot of helpers to manipulate dynamically the model.

Chain

PyTorch (Before)

```python
class BasicModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(1, channels, kernel_size)
        self.linear_1 = nn.Linear(hidden_layer_in, hidden_layer_out)
        self.maxpool = nn.MaxPool2d(2)
        self.linear_2 = nn.Linear(hidden_layer_out, output_size)

    def forward(self, x):
        x = self.conv(x)
        x = nn.functional.relu(x)
        x = self.maxpool(x)
        x = x.flatten(start_dim=1)
        x = self.linear_1(x)
        x = nn.functional.relu(x)
        x = self.linear_2(x)
        return nn.functional.softmax(x, dim=0)
```

Refiners (After)

```python
class BasicModel(fl.Chain):
    def __init__(self):
        super().__init__(
            fl.Conv2d(1, channels, kernel_size),
            fl.ReLU(),
            fl.MaxPool2d(2),
            fl.Flatten(start_dim=1),
            fl.Linear(hidden_layer_in, hidden_layer_out),
            fl.ReLU(),
            fl.Linear(hidden_layer_out, output_size),
            fl.Lambda(lambda x: torch.nn.functional.softmax(x, dim=0)),
        )
```

Chain

Let us instantiate the BasicModel we just defined and inspect its representation in a Python REPL:

```
>>> m = BasicModel()
>>> m
(CHAIN) BasicModel()
    ├── Conv2d(in_channels=1, out_channels=128, kernel_size=(3, 3), device=cpu, dt
    ├── ReLU() #1
    ├── MaxPool2d(kernel_size=2, stride=2)
    ├── Flatten(start_dim=1)
    ├── Linear(in_features=21632, out_features=200, device=cpu, dtype=float32) #1
    ├── ReLU() #2
    ├── Linear(in_features=200, out_features=10, device=cpu, dtype=float32) #2
    └── Softmax()
```

Chain

Chain includes several helpers to manipulate the tree. Let's organise the model by wrapping each layer in a subchain.

```
class ConvLayer(fl.Chain):
    pass

class HiddenLayer(fl.Chain):
    pass

class OutputLayer(fl.Chain):
    pass

m.insert(0, ConvLayer(m.pop(0), m.pop(0), m.pop(0)))
m.insert_after_type(ConvLayer, HiddenLayer(m.pop(1), m.pop(1), m.pop(1)))
m.append(OutputLayer(m.pop(2), m.pop(2)))
```

Chain

Did it work? Let's see:

```
>>> m
(CHAIN) BasicModel()
    ├── (CHAIN) ConvLayer()
    │   ├── Conv2d(in_channels=1, out_channels=128, kernel_size=(3, 3), device=cpu
    │   ├── ReLU()
    │   └── MaxPool2d(kernel_size=2, stride=2)
    ├── (CHAIN) HiddenLayer()
    │   ├── Flatten(start_dim=1)
    │   ├── Linear(in_features=21632, out_features=200, device=cpu, dtype=float32)
    │   └── ReLU()
    └── (CHAIN) OutputLayer()
        ├── Linear(in_features=200, out_features=10, device=cpu, dtype=float32)
        └── Softmax()
```

Context

- ✔ Simplify the flow of data by providing a stateful store to nested Chains.
- ✔ Avoiding "props drilling", exactly like in UI frameworks.
- ✔ Allow flexibility of using new inputs/modality without modifying existing code.

Context

```python
from refiners.fluxion.context import Contexts

class MyProvider(fl.Chain):
    def init_context(self) -> Contexts:
        return {"my context": {"my key": None}}

m = MyProvider(
    fl.Chain(
        fl.Sum(
            fl.UseContext("my context", "my key"),
            fl.Lambda(lambda: 2),
        ),
        fl.SetContext("my context", "my key"),
    ),
    fl.Chain(
        fl.UseContext("my context", "my key"),
        fl.Lambda(print),
    ),
)

m.set_context("my context", {"my key": 4})
m()  # prints 6
```

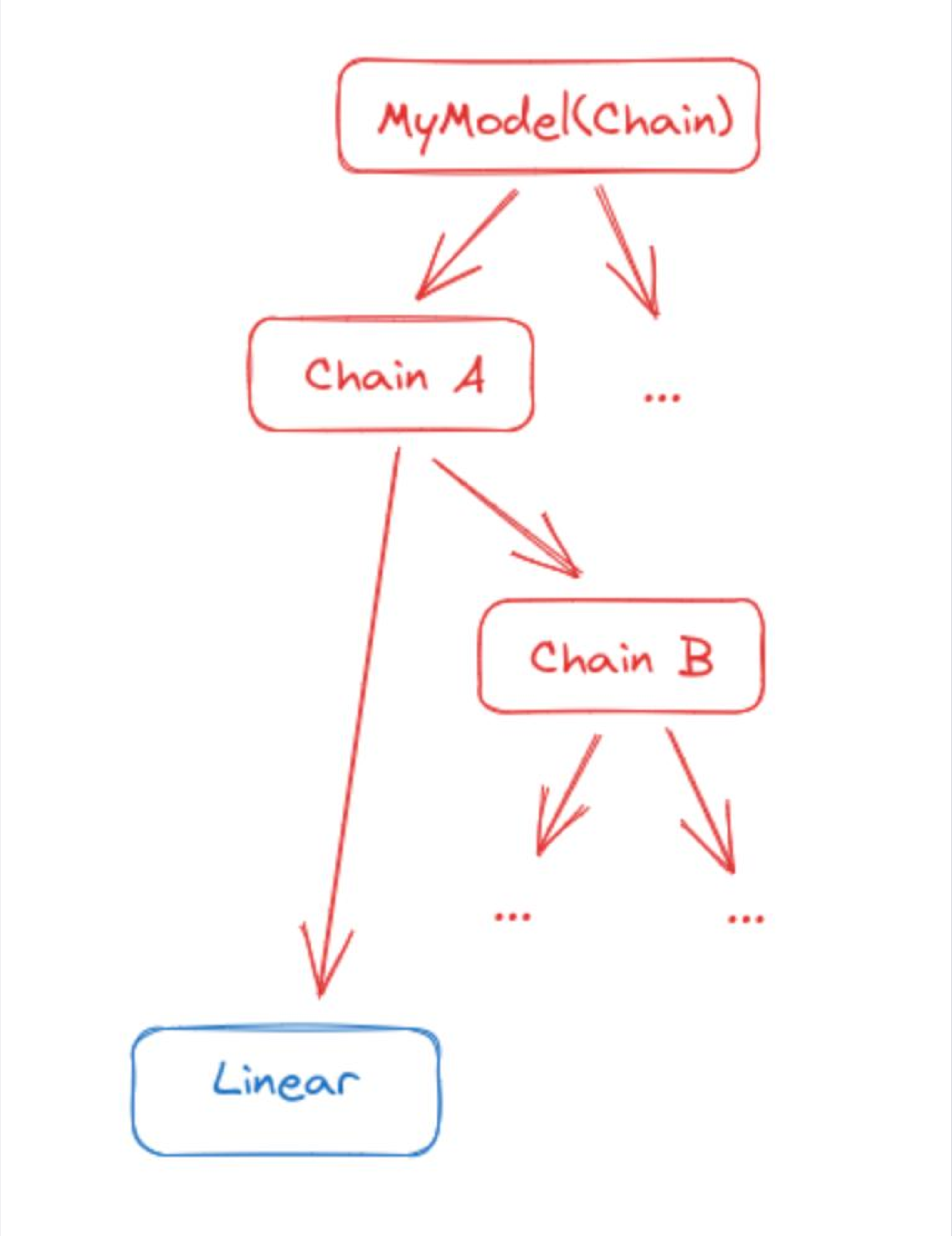
Adapter

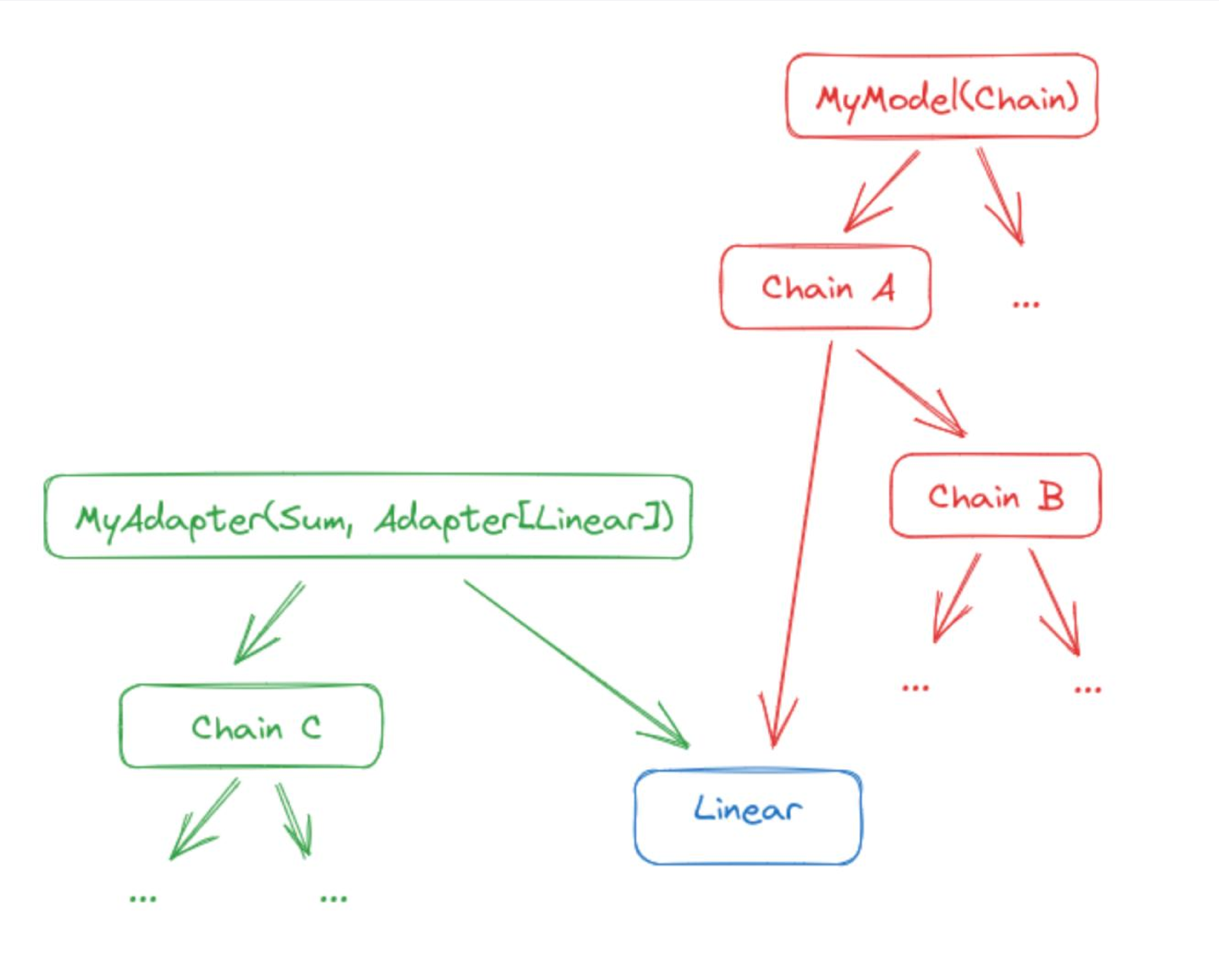
- ✅ Turn the concept of adaptation into code.
- ✅ Provide high-level abstractions to “inject” and “eject” adapters (i.e. restore state)
- ✅ Support model surgery by building upon Chain manipulation methods.

Adapter

Let us take a simple example to see how this works.

We want to adapt the Linear layer.





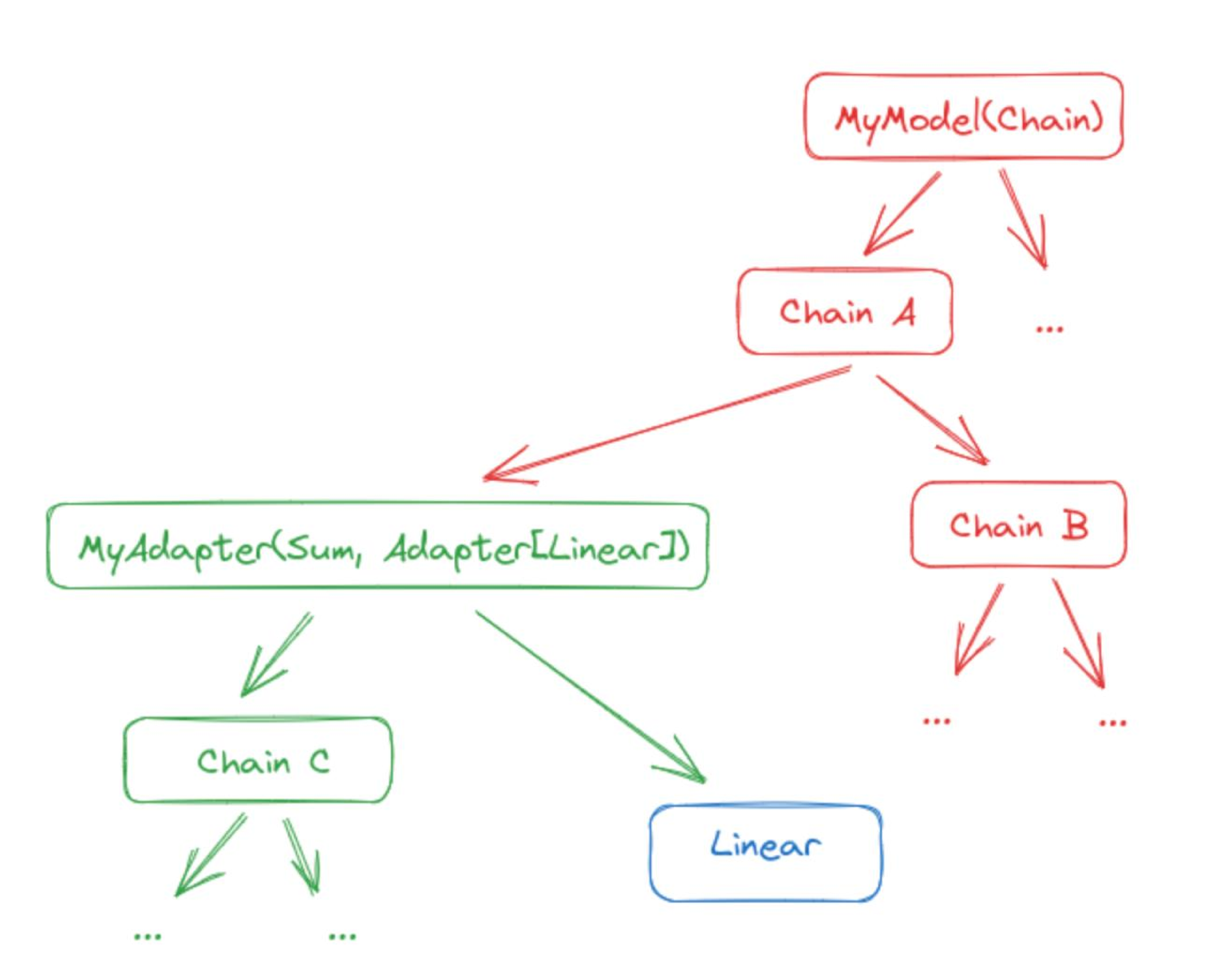
We want to wrap the Linear into a new Chain that is our Adapter

Note that the original chain is unmodified. You can run inference as if the adapter did not exist.

Adapter

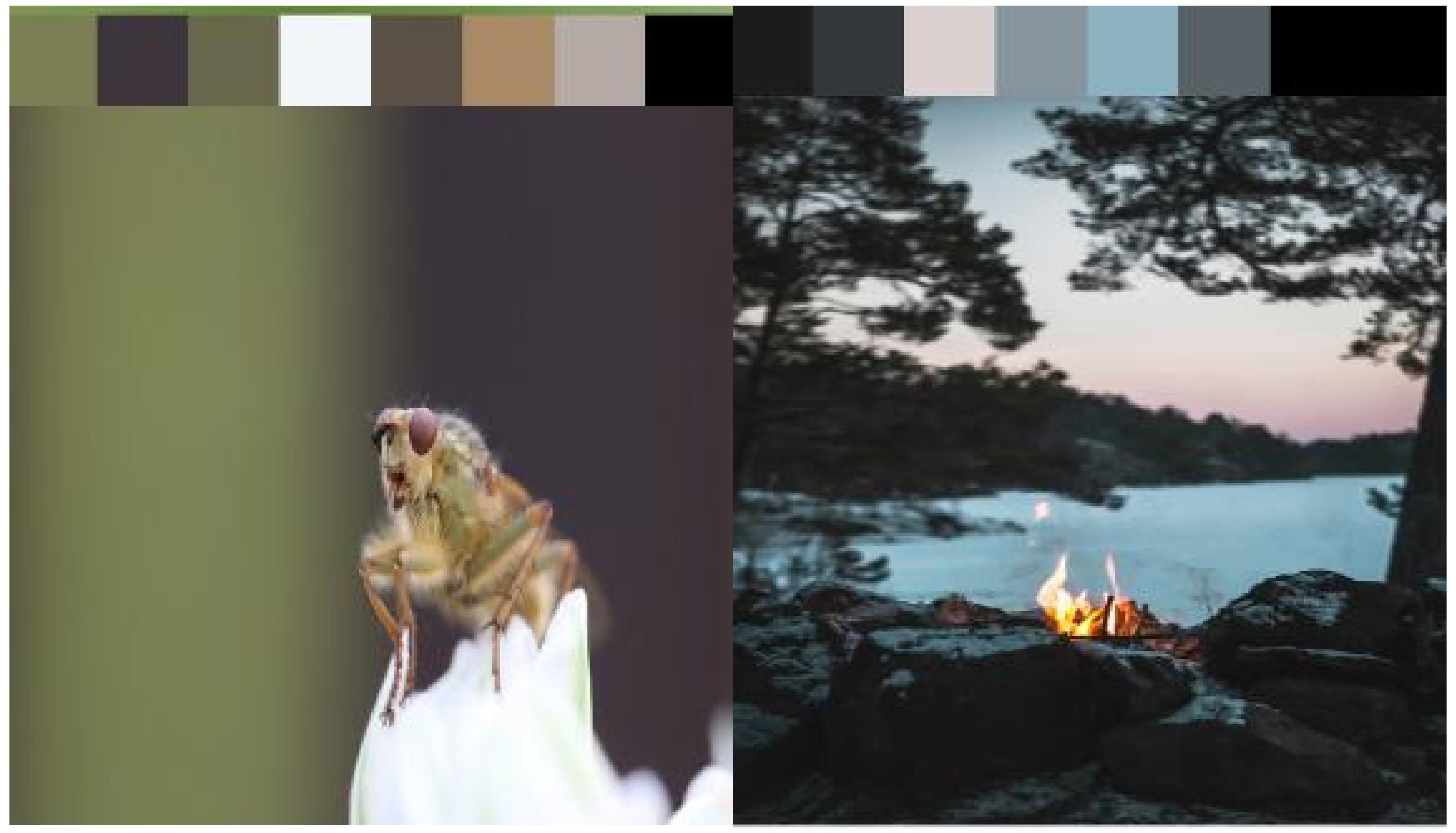
```python
class MyAdapter(fl.Sum, fl.Adapter[fl.Linear]):
    def __init__(self, target: fl.Linear) -> None:
        with self.setup_adapter(target):
            super().__init__(fl.Chain(...), target)

# Find the target and its parent in the chain.
# For simplicity let us assume it is the only Linear.
for target, parent in my_model.walk(fl.Linear):
    break

adapter = MyAdapter(target)
```

```
adapter.inject(parent)
```



We're currently training adapters in the open

Color Palette adapter



IP-Adapter with Dinov2 embeddings

if you want to train/implement adapters have a look at finegrain.ai/bounties 💰

Thank you for listening!

Please help us by leaving a ⭐ Starred on Github to support the project!