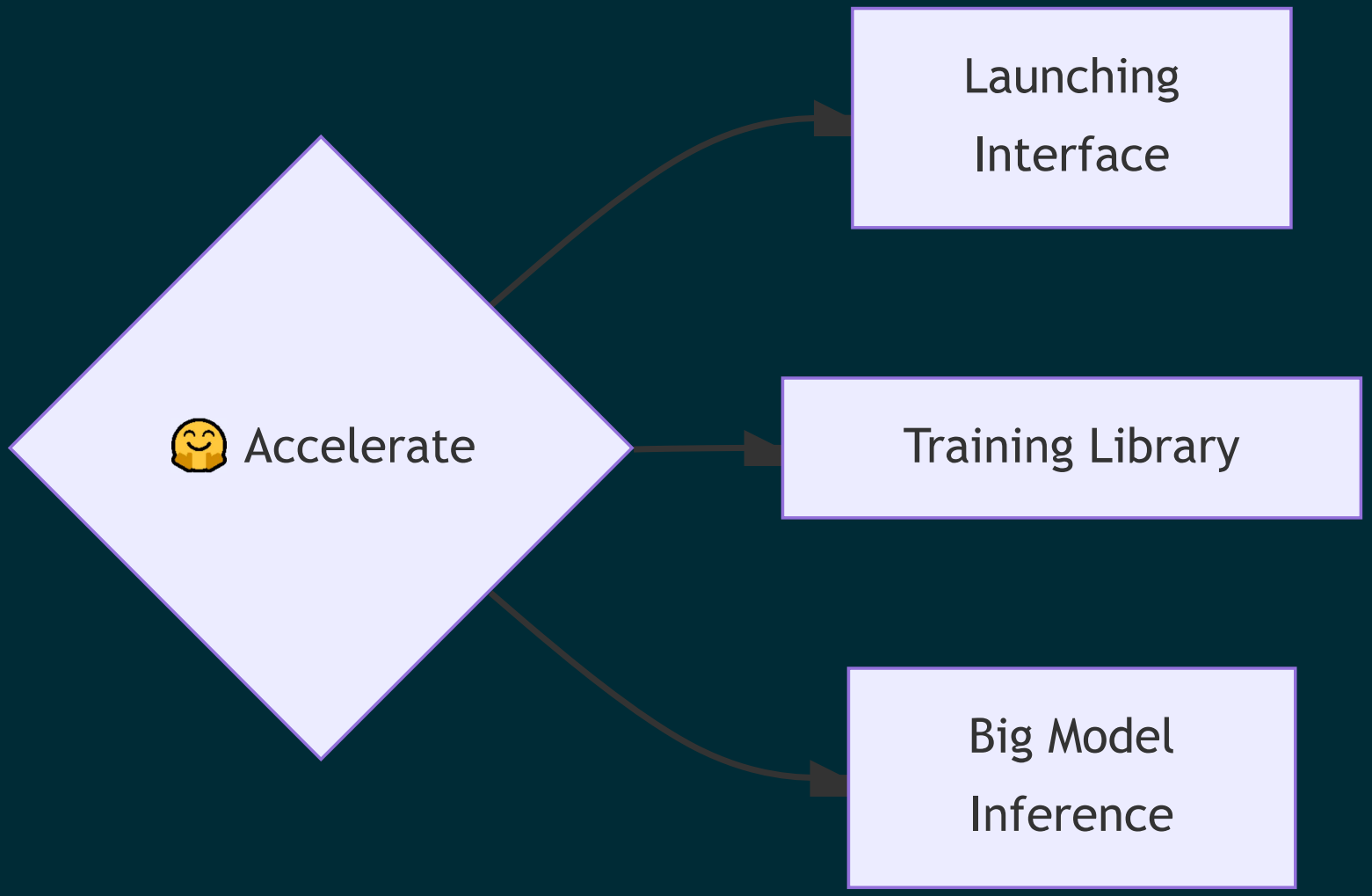
# ACCELERATE, THREE POWERFUL SUBLIBRARIES FOR PYTORCH

Zachary Mueller

# WHO AM I?

- Zachary Mueller
- Deep Learning Software Engineer at 🤗
- API design geek

# WHAT IS 🤗 ACCELERATE?



# A LAUNCHING INTERFACE

Can’t I just use `python do_the_thing.py`?

# A LAUNCHING INTERFACE

Launching scripts in different environments is complicated:

- ```
  python script.py
  ```
- ```
  torchrun --nnodes=1 --nproc_per_node=2 script.py
  ```
- ```
  deepspeed --num_gpus=2 script.py
  ```

And more!
  ```

# A LAUNCHING INTERFACE

But it doesn't have to be:

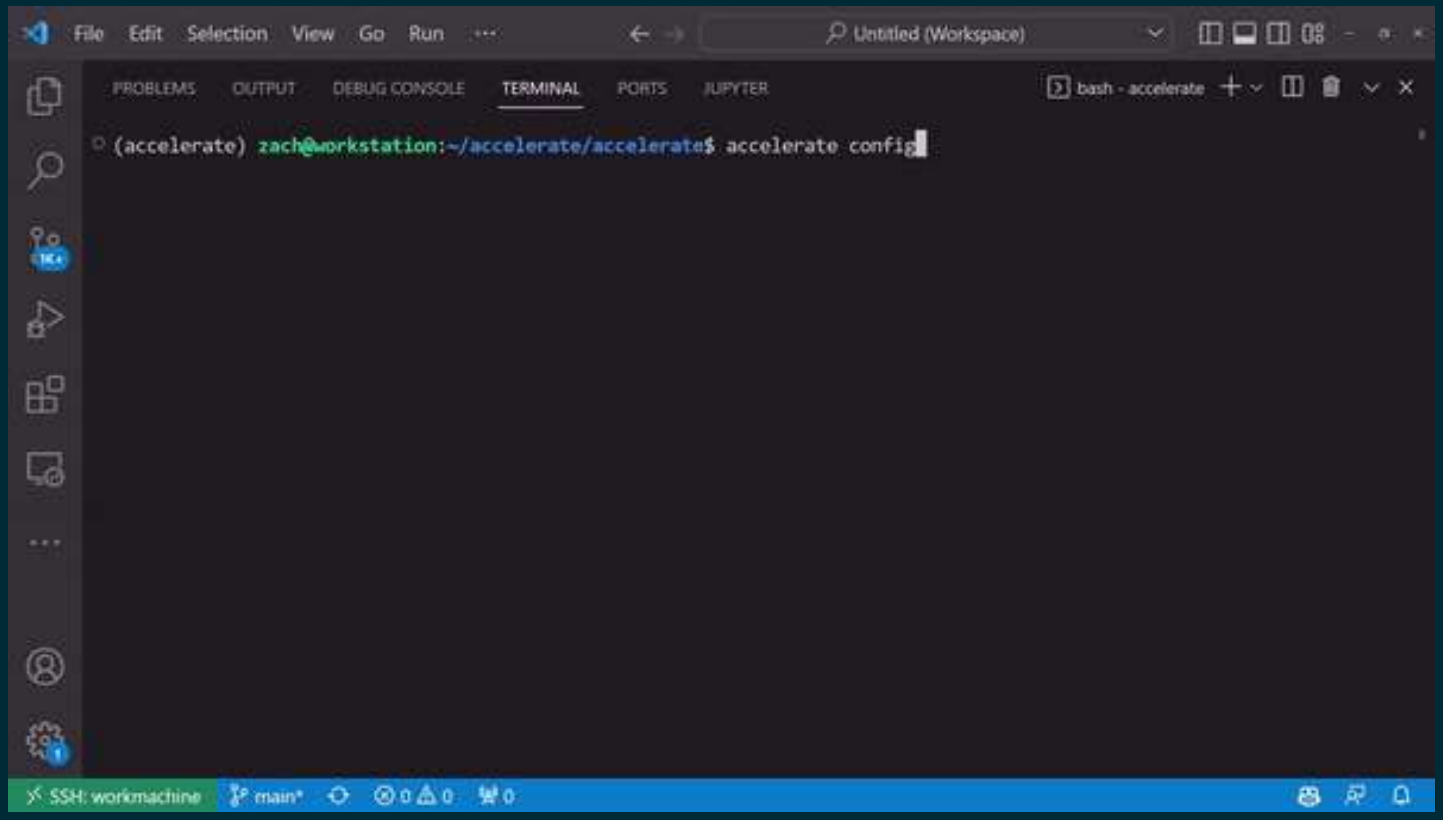
```
accelerate launch script.py
```

A single command to launch with DeepSpeed, Fully Sharded Data Parallelism, across single and multi CPUs and GPUs, and to train on TPUs[1] too!

1. Without needing to modify your code and create a `mp_fn`

# A LAUNCHING INTERFACE

Generate a device-specific configuration through `accelerate config`



# A LAUNCHING INTERFACE

Or don't. `accelerate config` doesn't *have* to be done!

```
torchrun --nnodes=1 --nproc_per_node=2 script.py
accelerate launch --multi_gpu --nproc_per_node=2 script.py
```

A quick default configuration can be made too:

```
accelerate config default
```

# A LAUNCHING INTERFACE

With the `notebook_launcher` it's also possible to launch code directly from your Jupyter environment too!

```
from accelerate import notebook_launcher
notebook_launcher(
    training_loop_function,
    args,
    num_processes=2
)
```

```
Launching training on 2 GPUs.
epoch 0: 88.12
epoch 1: 91.73
epoch 2: 92.58
epoch 3: 93.90
epoch 4: 94.71
```

# A TRAINING LIBRARY

Okay, will `accelerate launch` make `do_the_thing.py` use all my GPUs magically?

# A TRAINING LIBRARY

- Just showed that its possible using `accelerate launch` to *launch* a python script in various distributed environments
- This does *not* mean that the script will just "use" that code and still run on the new compute efficiently.
- Training on different computes often means *many* lines of code changed for each specific compute.
- 🤗 `accelerate` solves this by ensuring the same code can be ran on a CPU or GPU, multiples, and on TPUs!

# A TRAINING LIBRARY

```
for batch in dataloader:
    optimizer.zero_grad()
    inputs, targets = batch
    inputs = inputs.to(device)
    targets = targets.to(device)
    outputs = model(inputs)
    loss = loss_function(outputs, targets)
    loss.backward()
    optimizer.step()
    scheduler.step()
```

# A TRAINING LIBRARY

```python
# For alignment purposes
for batch in dataloader:
    optimizer.zero_grad()
    inputs, targets = batch
    inputs = inputs.to(device)
    targets = targets.to(device)
    outputs = model(inputs)
    loss = loss_function(outputs, targets)
    loss.backward()
    optimizer.step()
    scheduler.step()
```

```python
from accelerate import Accelerator
accelerator = Accelerator()
dataloader, model, optimizer scheduler = (
    accelerator.prepare(
        dataloader, model, optimizer, scheduler
    )
)

for batch in dataloader:
    optimizer.zero_grad()
    inputs, targets = batch
    # inputs = inputs.to(device)
    # targets = targets.to(device)
    outputs = model(inputs)
    loss = loss_function(outputs, targets)
    accelerator.backward(loss) # loss.backward()
    optimizer.step()
    scheduler.step()
```

# A TRAINING LIBRARY

What all happened in `Accelerator.prepare`?

1. `Accelerator` looked at the configuration
2. The `dataloader` was converted into one that can dispatch each batch onto a seperate GPU
3. The `model` was wrapped with the appropriate DDP wrapper from either `torch.distributed` or `torch_xla`
4. The `optimizer` and `scheduler` were both converted into an `AcceleratedOptimizer` and `AcceleratedScheduler` which knows how to handle any distributed scenario

# A TRAINING LIBRARY, MIXED PRECISION

🤗 `accelerate` also supports *automatic mixed precision*.

Through a single flag to the `Accelerator` object when calling `accelerator.backward()` the mixed precision of your choosing (such as `bf16` or `fp16`) will be applied:

```
 1  from accelerate import Accelerator
 2  accelerator = Accelerator(mixed_precision="fp16")
 3  ...
 4  for batch in dataloader:
 5      optimizer.zero_grad()
 6      inputs, targets = batch
 7      outputs = model(inputs)
 8      loss = loss_function(outputs, targets)
 9      accelerator.backward(loss)
10      optimizer.step()
11      scheduler.step()
```

# A TRAINING LIBRARY, GRADIENT ACCUMULATION

Gradient accumulation in distributed setups often need extra care to ensure gradients are aligned when they need to be and the backward pass is computationally efficient.

🤗 `accelerate` can just easily handle this for you:

```python
from accelerate import Accelerator
accelerator = Accelerator(gradient_accumulation_steps=4)
...
for batch in dataloader:
    with accelerator.accumulate(model):
        optimizer.zero_grad()
        inputs, targets = batch
        outputs = model(inputs)
        loss = loss_function(outputs, targets)
        accelerator.backward(loss)
        optimizer.step()
        scheduler.step()
```

# A TRAINING LIBRARY, GRADIENT ACCUMULATION

```python
ddp_model, dataloader = accelerator.prepare(model, dataloader)

for index, batch in enumerate(dataloader):
    inputs, targets = batch
    if index != (len(dataloader)-1) or (index % 4) != 0:
        # Gradients don't sync
        with accelerator.no_sync(model):
            outputs = ddp_model(inputs)
            loss = loss_func(outputs, targets)
            accelerator.backward(loss)
    else:
        # Gradients finally sync
        outputs = ddp_model(inputs)
        loss = loss_func(outputs)
        accelerator.backward(loss)
```

# BIG MODEL INFERENCE

Stable Diffusion taking the world by storm

# BIGGER MODELS == HIGHER COMPUTE

As more large models were being released, Hugging Face quickly realized there must be a way to continue our decentralization of Machine Learning and have the day-to-day programmer be able to leverage these big models.

Born out of this effort by Sylvain Gugger:

🤗 Accelerate: Big Model Inference.

# THE BASIC PREMISE

- In PyTorch, there exists the `meta` device.
- Super small footprint to load in huge models quickly by not loading in their weights immediatly.
- As an input gets passed through each layer, we can load and unload *parts* of the PyTorch model quickly so that only a small portion of the big model is loaded in at a single time.
- The end result? Stable Diffusion v1 can be ran on < 800mb of vRAM

# THE CODE

Generally you start with something like so:

```
1 import torch
2
3 my_model = ModelClass(...)
4 state_dict = torch.load(checkpoint_file)
5 my_model.load_state_dict(state_dict)
```

But this has issues:

1. The full version of the model is loaded at 3
2. Another version of the model is loaded into memory at 4

If a 6 *billion* parameter model is being loaded, each model class has a dictionary of 24GB so 48GB of vRAM is needed

# EMPTY MODEL WEIGHTS

We can fix step 1 by loading in an empty model skeleton at first:

```
1 from accelerate import init_empty_weights
2
3 with init_empty_weights():
4     my_model = ModelClass(...)
5 state_dict = torch.load(checkpoint_file)
6 my_model.load_state_dict(state_dict)
```

**This code will not run**

It is likely that just calling `my_model(x)` will fail as not all tensor operations are supported on the `meta` device.

# SHARDED CHECKPOINTS - THE CONCEPT

The next step is to have “Sharded Checkpoints” saved for your model.

Basically smaller chunks of your model weights stored that can be brought in at any particular time.

This reduces the amount of memory step 2 takes in since we can just load in a “chunk” of the model at a time, then swap it out for a new chunk through PyTorch hooks

# SHARDED CHECKPOINTS - THE CODE

```python
from accelerate import init_empty_weights, load_checkpoint_and_dispatch

with init_empty_weights():
    my_model = ModelClass(...)

my_model = load_checkpoint_and_dispatch(
    my_model, "sharded-weights", device_map="auto"
)
```

`device_map="auto"` will tell 🤗 Accelerate that it should determine where to put each layer of the model:

1. Maximum space on the GPU(s)
2. Maximum space on the CPU(s)
3. Utilize disk space through memory-mapped tensors

# BIG MODEL INFERENCE PUT TOGETHER

```
from accelerate import init_empty_weights, load_checkpoint_and_dispatch

with init_empty_weights():
    my_model = ModelClass(...)

my_model = load_checkpoint_and_dispatch(
    my_model, "sharded-weights", device_map="auto"
)
my_model.eval()

for batch in dataloader:
    output = my_model(batch)
```

# IS THERE AN EASIER WAY?

The `transformers` library combined with the Hub makes all this code wrapping much easier for you with the `pipeline`

```
import torch
from transformers import pipeline
pipe = pipeline(
    task="text-generation",
    model="EleutherAI/gpt-j-6B",
    device_map="auto",
    torch_dtype=torch.float16
)

text = pipe("This is some generated text, I think")
```

# WHAT ABOUT STABLE DIFFUSION?

A demo with `diffusers`

# SOME HANDY RESOURCES

- 🤗 Accelerate documentation
- Launching distributed code
- Distributed code and Jupyter Notebooks
- Migrating to 🤗 Accelerate easily
- Big Model Inference tutorial
- DeepSpeed and 🤗 Accelerate
- Fully Sharded Data Parallelism and 🤗 Accelerate