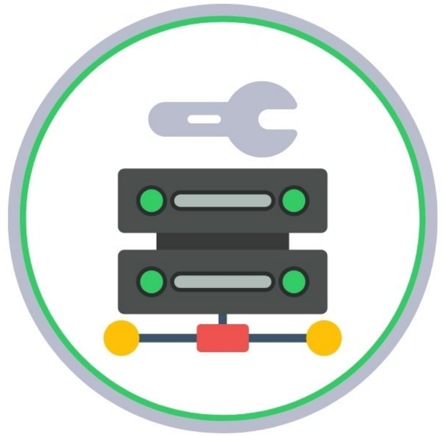
Prepare models with AutoModel and Accelerator

EFFICIENT AI MODEL TRAINING WITH PYTORCH

Dennis Lee
Data Engineer

Meet your instructor!

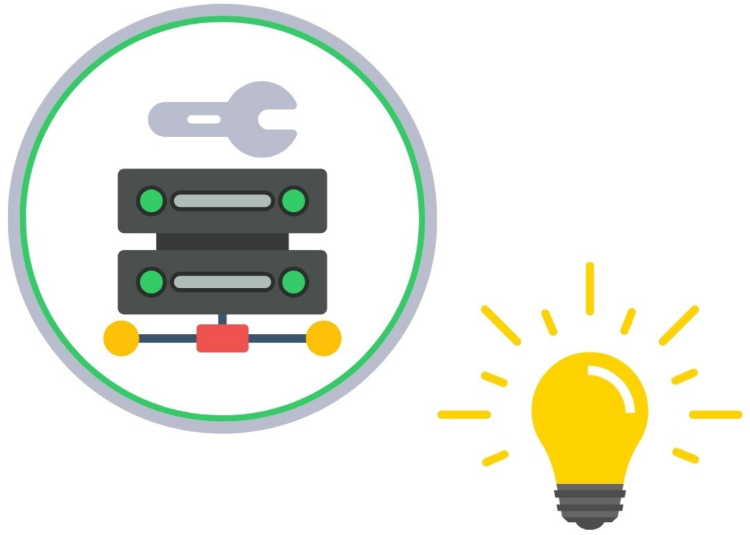
- Data engineer



datacamp

Meet your instructor!

- Data engineer
- Data scientist



datacamp

Meet your instructor!

- Data engineer
- Data scientist
- Ph.D. in Electrical Engineering

Meet your instructor!

- Data engineer
- Data scientist
- Ph.D. in Electrical Engineering

Excited to share best practices!

Our roadmap to efficient AI training

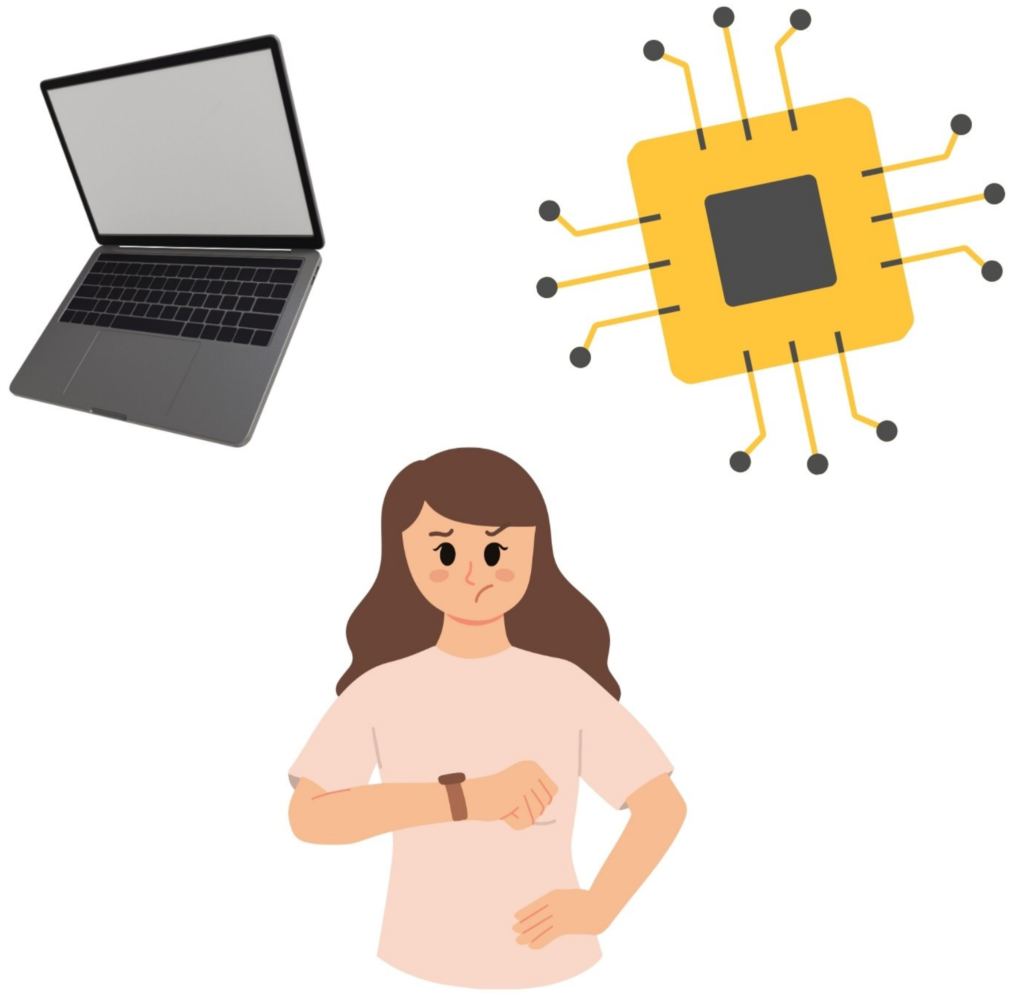
- Distributed AI model training

Our roadmap to efficient AI training

- Distributed AI model training

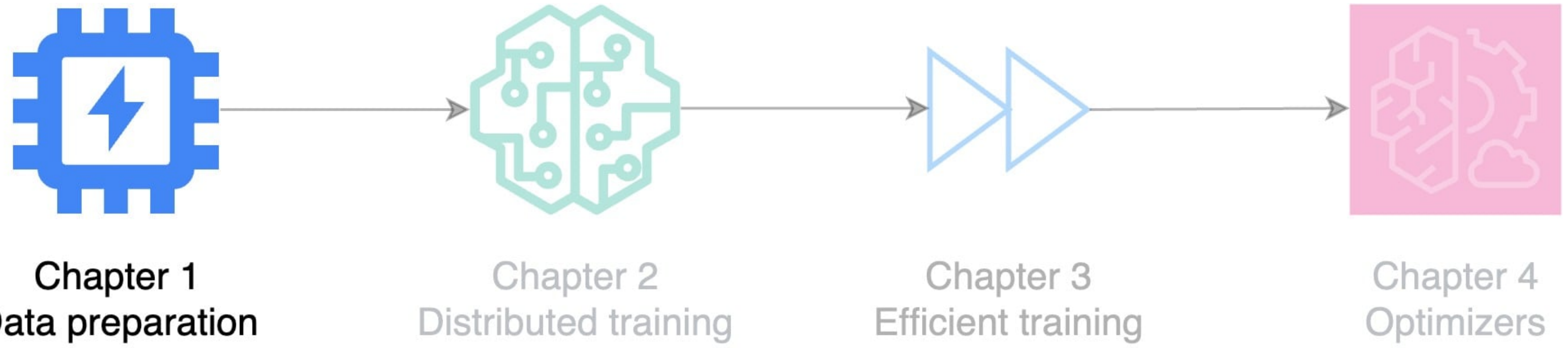
Our roadmap to efficient AI training

- Distributed AI model training



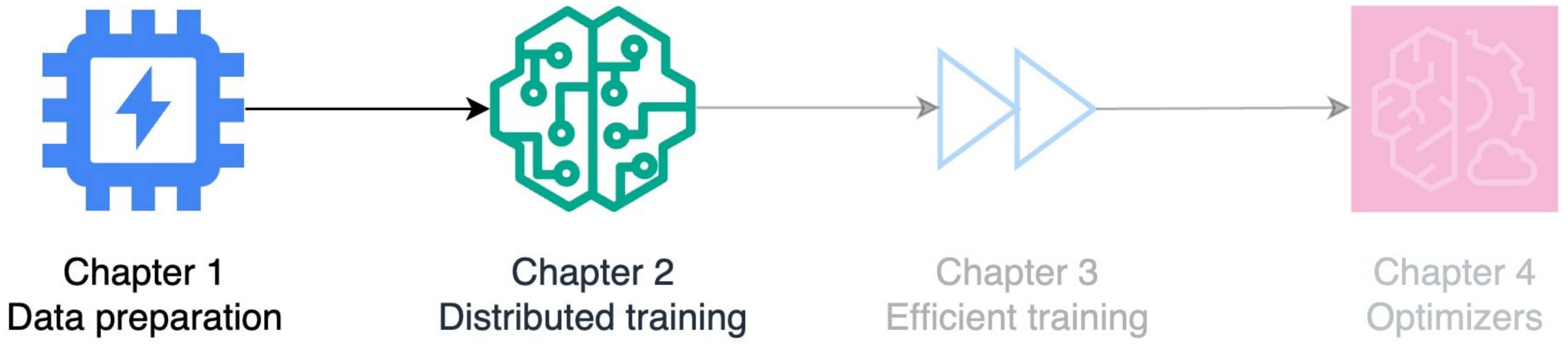
- ↓↓ Training times for large language models

Our roadmap to efficient AI training



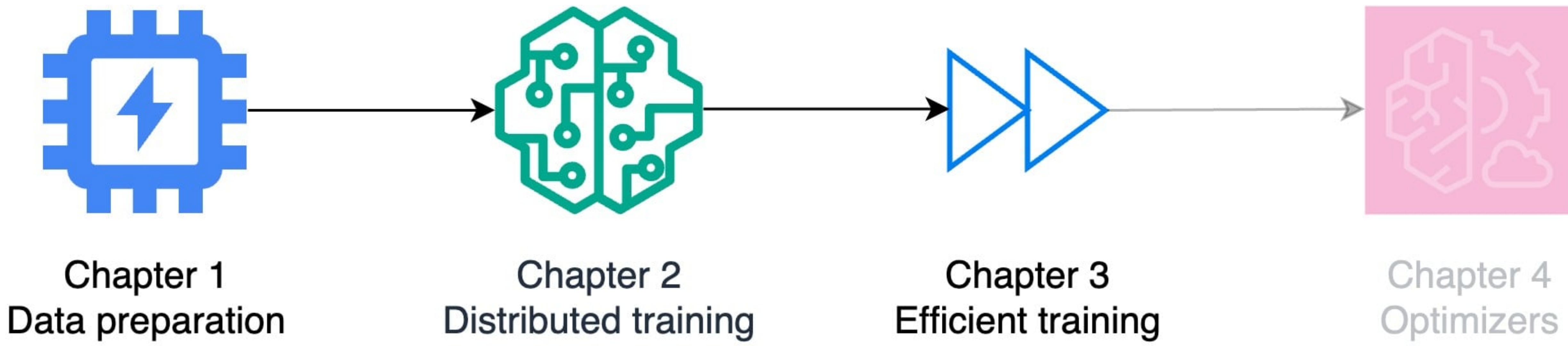
- Data preparation: placing data on multiple devices

Our roadmap to efficient AI training



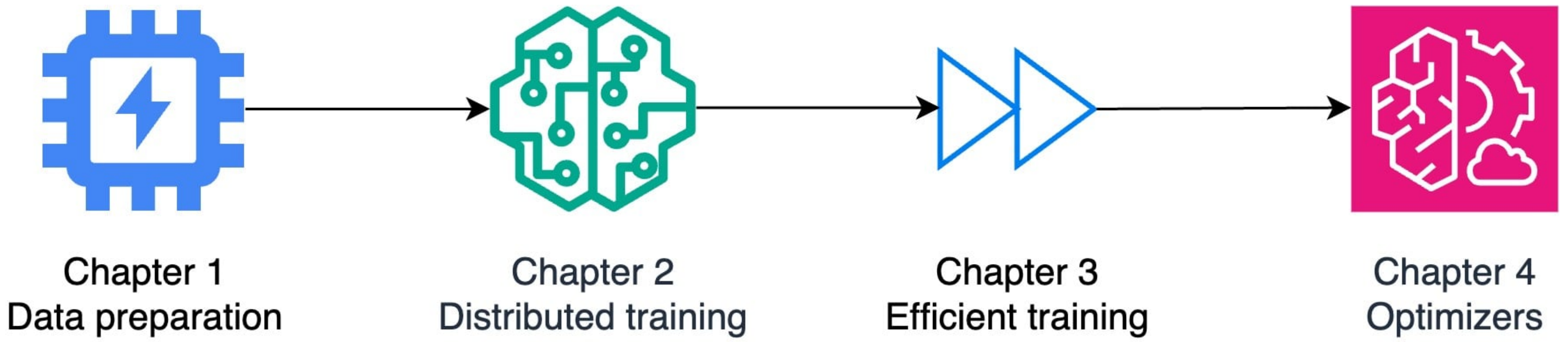
- Data preparation: placing data on multiple devices
- Distributed training: scaling training to multiple devices

Our roadmap to efficient AI training



- Data preparation: placing data on multiple devices
- Distributed training: scaling training to multiple devices
- Efficient training: optimizing available devices

Our roadmap to efficient AI training



- Data preparation: placing data on multiple devices
- Distributed training: scaling training to multiple devices
- Efficient training: optimizing available devices
- Optimizers: accelerating training

CPUs

- Most laptops have CPUs

GPUs

- GPUs can train large models

CPUs vs GPUs

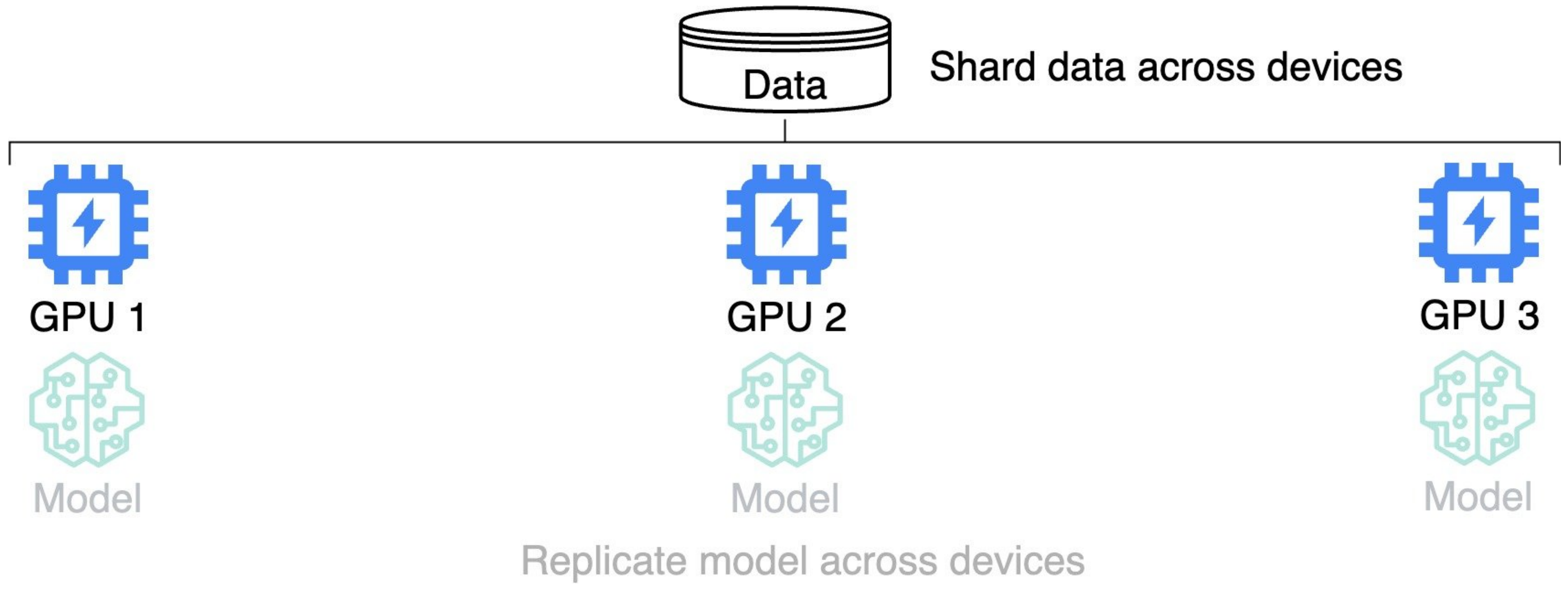
CPUs

- Most laptops have CPUs
- Designed for general purpose computing
- Better control flow

GPUs

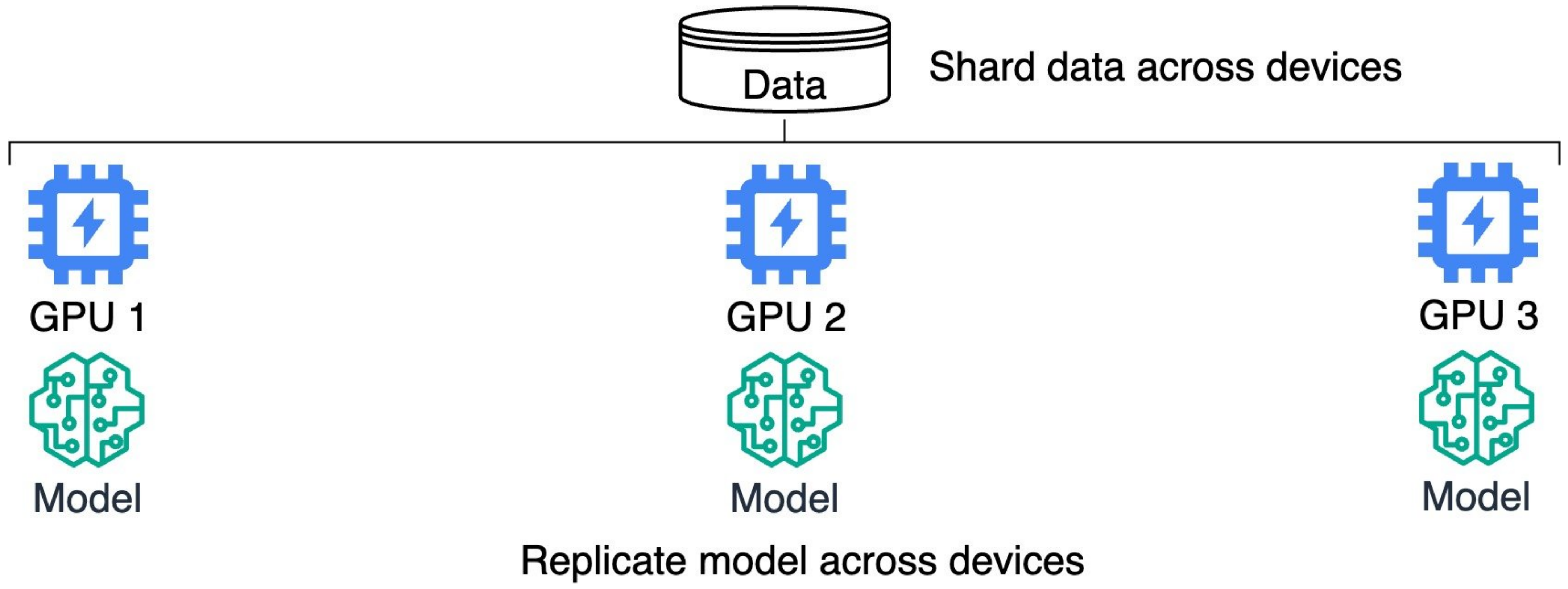
- GPUs can train large models
- Specialize in highly parallel computing
- Excel at matrix operations

Distributed training



- **Data sharding:** each device processes a subset of data in parallel

Distributed training



- **Data sharding:** each device processes a subset of data in parallel
- **Model replication:** each device performs forward/backward passes
- **Gradient aggregation:** designated device aggregates gradients
- **Parameter synchronization:** designated device shares updated parameters

Effortless efficiency: leveraging pre-trained models

- Leverage pre-trained Transformer models
- Initialize model parameters by calling `AutoModelForSequenceClassification`
- Display the configuration

```
from transformers import AutoModelForSequenceClassification
model = AutoModelForSequenceClassification.from_pretrained(model_name)
print(model.config)
```

```
DistilBertConfig {
  "architectures": ["DistilBertForMaskedLM"],
  "dim": 768,
  "dropout": 0.1,
  "hidden_dim": 3072,
  ...
```

Device placement with Accelerator

- A Hugging Face class 🤗
- `Accelerator` detects which devices are available on our computer
- Automate device placement and data parallelism: `accelerator.prepare()`
- Place the model (with type `torch.nn.Module`) on the first available GPU
- Defaults to the CPU if no GPU is found

```
from accelerate import Accelerator
accelerator = Accelerator()
model = accelerator.prepare(model)
print(accelerator.device)
```

```
cpu
```

Let's practice!

EFFICIENT AI MODEL TRAINING WITH PYTORCH

Preprocess images and audio for training

EFFICIENT AI MODEL TRAINING WITH PYTORCH

Dennis Lee
Data Engineer

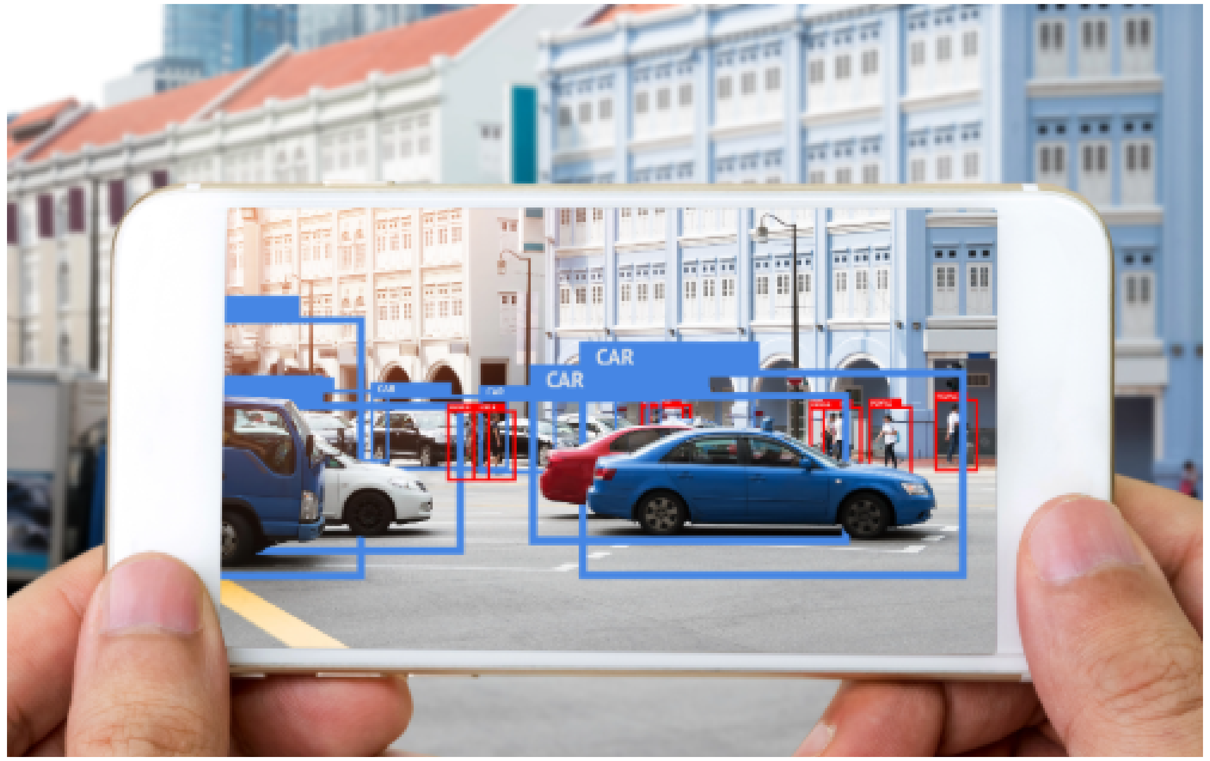
Preparing images and audio

Image application

- Image classification to identify objects
- Data sharding

Audio application

- Provide voice commands
- Example: "Turn down the volume"



Manipulating a sample image dataset

```
print(dataset)
```

```
Dataset({
    features: ['img', 'label'],
    num_rows: 1000
})
```

```
print(dataset[0]["img"])
```

```
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=720x480>
```

Standardize the image format

- Format images: width, height
- Standardize pixel values: mean, standard deviation
- `AutoImageProcessor` loads all preprocessing steps

```
from transformers import AutoImageProcessor
model = "microsoft/swin-tiny-patch4-window7-224"
image_processor = AutoImageProcessor.from_pretrained(model)
```

Standardize the image format

```
dataset = dataset.map(
    lambda examples: {
        "pixel_values": [
            image_processor(image, return_tensors="pt").pixel_values
            for image in examples["img"]
        ]}, batched=True)
print(dataset)
```

```
Dataset({
    features: ['img', 'label', 'pixel_values'],
    num_rows: 1000
})
```

Manipulating a sample audio dataset

```
print(dataset)
```

```
DatasetDict({
    train: Dataset({
        features: ['file',
                   'audio',
                   'label'],
        num_rows: 1000
    }), ...
})
```

Standardize the audio format

- Standardize number of samples
- Sampling rate: Number of samples per second
- Max duration: Number of seconds of audio

```
sampling_rate = 16000  # 16 kHz
max_duration = 1  # 1 second
max_length = sampling_rate * max_duration
print(f"max_length = {max_length:,} samples")
```

```
max_length = 16,000 samples
```

Standardize the audio format

```python
from transformers import AutoFeatureExtractor

model = "facebook/wav2vec2-base"
feature_extractor = AutoFeatureExtractor.from_pretrained(model)

def preprocess_function(split_data):
    audio_arrays = [x["array"] for x in split_data["audio"]]
    inputs = feature_extractor(audio_arrays,
                               sampling_rate=feature_extractor.sampling_rate,
                               max_length=int(feature_extractor.sampling_rate
                                              * max_duration),
                               truncation=True)
    return inputs
```

Apply the preprocesssing function

- Map the `preprocess_function` to the `dataset`
- `remove_columns` : remove `audio` and `file` columns
- `batched` : process `dataset` examples in batches

```
dataset = dataset["train"].map(preprocess_function,
                                remove_columns=["audio", "file"],
                                batched=True)
```

Apply the preprocesssing function

```python
print(dataset)
```

```
DatasetDict({
    train: Dataset({
        features: ['label', 'input_values'],
        num_rows: 1000
    })
```

Prepare data for distributed training

- `DataLoader` : prepare the data for loading and iterating during training
- `accelerator.prepare()` : place the data on CPUs or GPUs based on availability
- Data sharding: each GPU processes a subset of training data, like sharing slices of pizza
- `accelerator.prepare()` works with PyTorch DataLoaders (`torch.utils.data.DataLoader`)

```python
from accelerate import Accelerator
from torch.utils.data import DataLoader

dataloader = DataLoader(dataset, batch_size=32, shuffle=True)

accelerator = Accelerator()
dataloader = accelerator.prepare(dataloader)
```

Let's practice!

EFFICIENT AI MODEL TRAINING WITH PYTORCH

Preprocess text for training

EFFICIENT AI MODEL TRAINING WITH PYTORCH

Dennis Lee
Data Engineer

Text transformation: preparing data for model mastery

- Summarize text in documents
- Paraphrase identification
- MRPC dataset: sentence pairs with labels

Dataset structure

```python
from datasets import load_dataset
dataset = load_dataset("glue", "mrpc")
print(dataset)
```

```
DatasetDict({
    train: Dataset({
        features: ['sentence1', 'sentence2', 'label', 'idx'],
    })
    validation: Dataset({
        features: ['sentence1', 'sentence2', 'label', 'idx'],
    })
    test: Dataset({
        features: ['sentence1', 'sentence2', 'label', 'idx'],
    })
})
```

Manipulating the text dataset

- Nested dictionary of train/validation/test splits
- Example of accessing the train split:

```
dataset["train"]
```

- Access dataset-specific features within a split
- MRPC dataset features: `sentence1` , `sentence2` , `label`

```
dataset["train"]["sentence1"]
```

- Load pre-trained tokenizer

```
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-cased")
```

Define an encoding function

- Define a function to encode examples from our dataset
- Call the tokenizer; extract `sentence1` and `sentence2` from the training example
- `truncation` : Truncate inputs if longer than max length (512 tokens)
- `padding` : Pad short sequences with zeros so all inputs have the same length

```
def encode(example):
    return tokenizer(
        example["sentence1"],
        example["sentence2"],
        truncation=True,
        padding="max_length",
    )
```

Format column names

- Apply `encode` to each example in the train split using `map`

```
train_dataset = dataset["train"].map(encode, batched=True)
```

- Rename `label` to `labels`

```
train_dataset = train_dataset.map(
    lambda examples: {"labels": examples["label"]}, batched=True
)
```

- Look up model requirements for columns in the Hugging Face documentation

Saving and loading checkpoints

- Place dataset on available GPUs

```
dataloader = DataLoader(train_dataset, batch_size=32, shuffle=True)
dataloader = accelerator.prepare(dataloader)
```

- Works with any PyTorch dataset (`torch.utils.data.Dataset`) in a `DataLoader`
- Save the state of preprocessed text, called a checkpoint

```
checkpoint_dir = Path("preprocess_checkpoint")
accelerator.save_state(checkpoint_dir)
```

- Load the checkpoint when we want to resume training

```
accelerator.load_state(checkpoint_dir)
```

Let's practice!

EFFICIENT AI MODEL TRAINING WITH PYTORCH