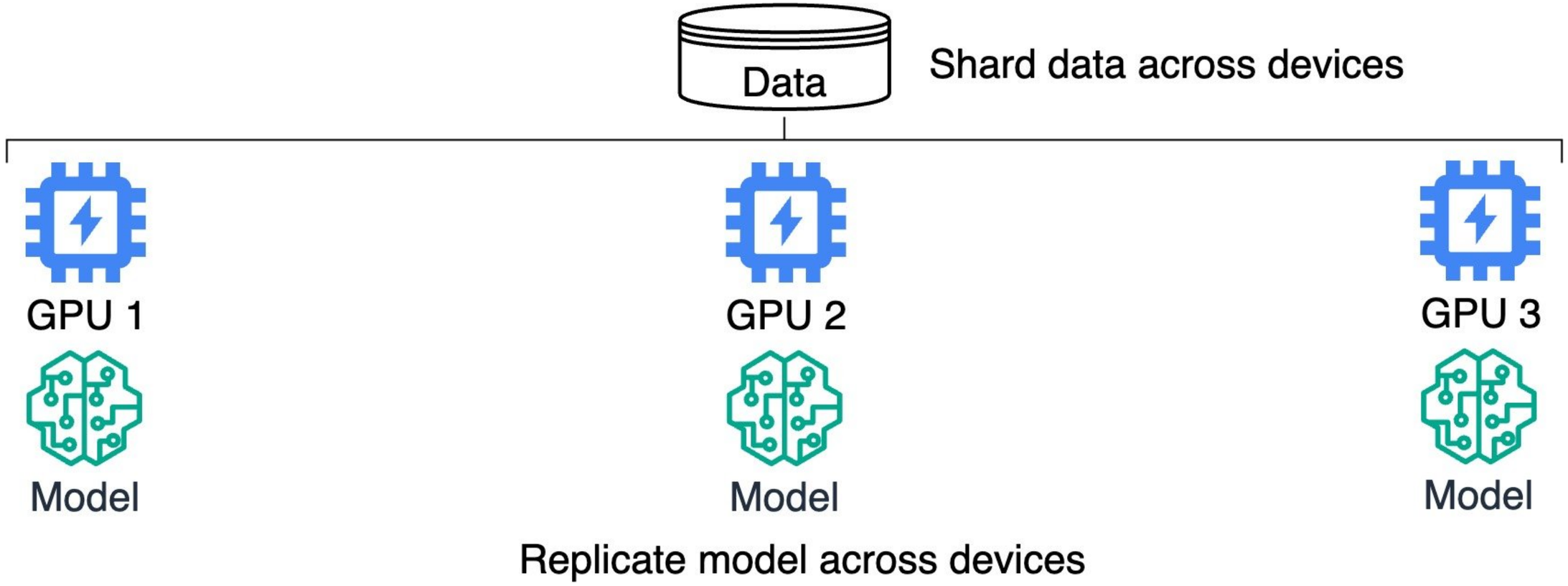
Fine-tune models with Trainer

EFFICIENT AI MODEL TRAINING WITH PYTORCH

Dennis Lee
Data Engineer

datacamp

Data preparation



Distributed training

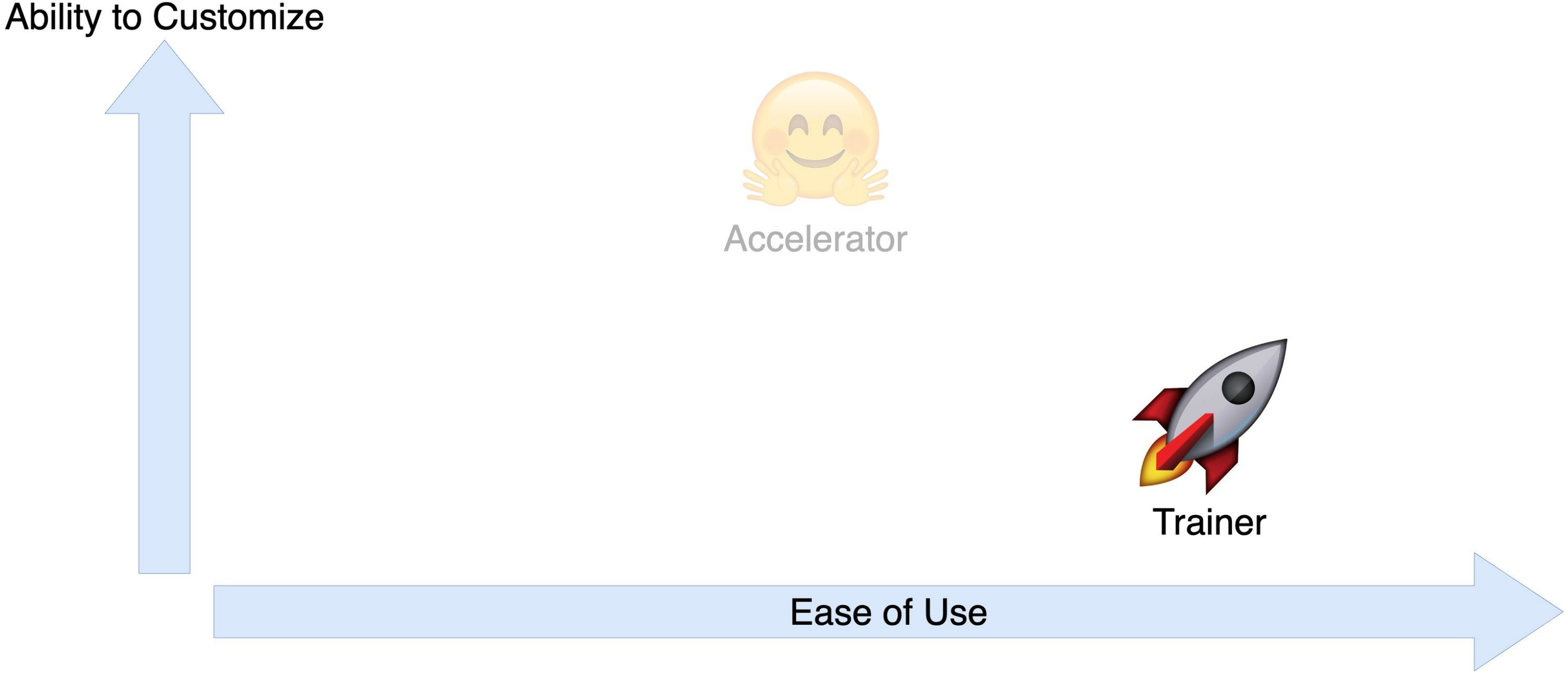
Chapter 1
Data preparation

Chapter 2
Distributed training

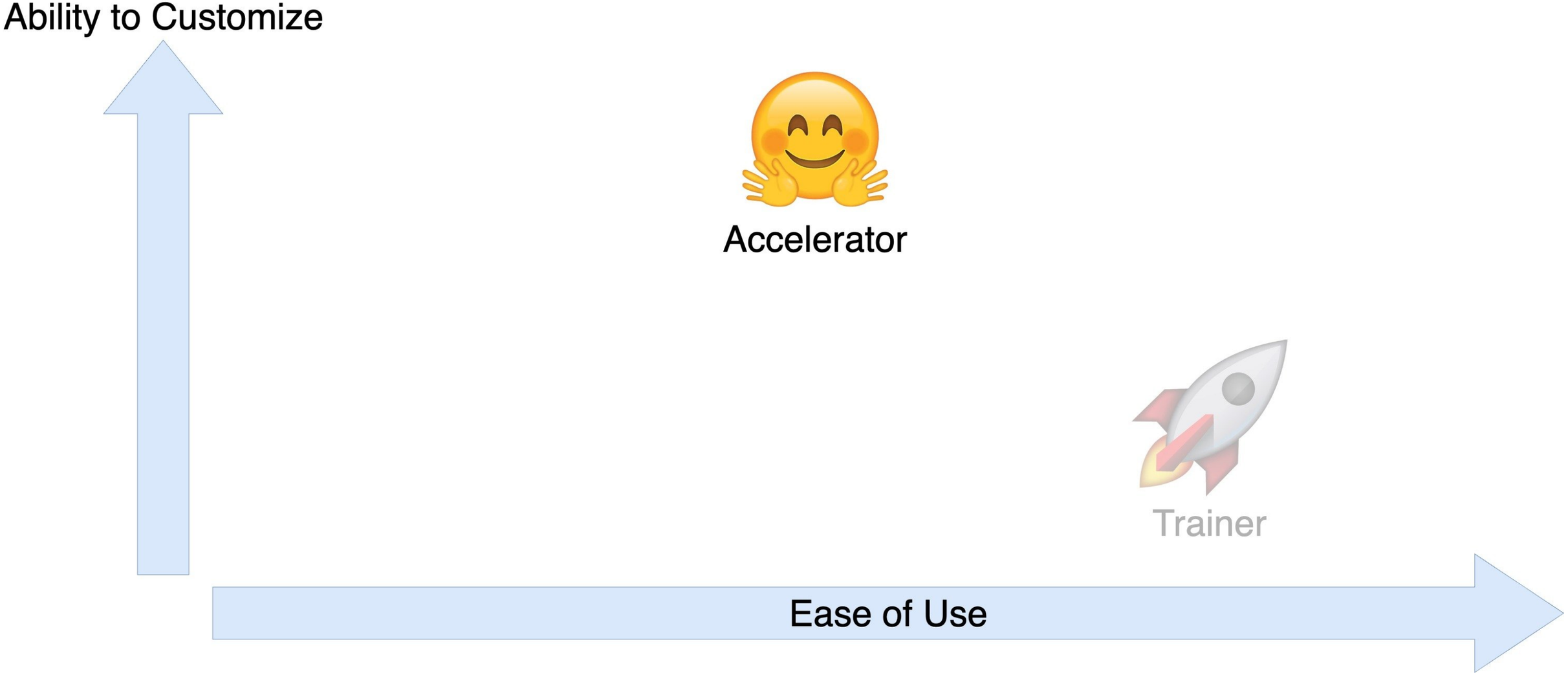
Chapter 3
Efficient training

Chapter 4
Optimizers

Trainer and Accelerator



Trainer and Accelerator



Turbocharge training with Trainer

- Trainer library

```
from transformers import Trainer
```

- Run model on each device in parallel
- Speed up training, like assembly lines
- Review inputs: dataset, model, metrics
- Develop sentiment analysis for e-commerce

Product review sentiment dataset

```
print(dataset)
```

```
DatasetDict({
    train: Dataset({
        features: ['Text', 'Label'],
        num_rows: 1000
    }), ...})
```

```
print(f'"{dataset["train"]["Text"][0]}": {dataset["train"]["Label"][0]}')
```

```
"I love this product!": positive
```

Convert labels to integers

```
def map_labels(example):
    if example["Label"] == "negative":
        return {"labels": 0}
    else:
        return {"labels": 1}

dataset = dataset.map(map_labels)
print(f'First label: {dataset["train"]["labels"][0]}')
```

```
First label: 1
```

Define the tokenizer and model

- Load pre-trained model and tokenizer:

```
model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased",
                                                           num_labels=2)
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
```

- Apply tokenizer to the `text` field:

```
def encode(examples):
    return tokenizer(examples["Text"], padding="max_length", truncation=True,
                     return_tensors="pt")
dataset = dataset.map(encode, batched=True)
print(f'The first tokenized review is {dataset["train"]["input_ids"][0]}.')
```

```
The first tokenized review is [101, 1045, 2293, 2023, 4031, 999, 102].
```

Define evaluation metrics

```
import evaluate


def compute_metrics(eval_predictions):
    load_accuracy = evaluate.load("accuracy")
    load_f1 = evaluate.load("f1")
    logits, labels = eval_predictions
    predictions = np.argmax(logits, axis=-1)
    accuracy = load_accuracy.compute(predictions=predictions, references=labels)[
        "accuracy"
    ]
    f1 = load_f1.compute(predictions=predictions, references=labels)["f1"]
    return {"accuracy": accuracy, "f1": f1}
```

Training arguments

- `output_dir` : Where to save model
- Specify hyperparameters (e.g., `learning_rate` and `weight_decay`)
- `save_strategy` : Save after each epoch
- `evaluation_strategy` : Evaluate metrics after each epoch

```
from transformers import (
    TrainingArguments)

training_args = TrainingArguments(
    output_dir="output_folder",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=2,
    weight_decay=0.01,
    save_strategy="epoch",
    evaluation_strategy="epoch",
)
```

Setting up Trainer

```
from transformers import Trainer

trainer = Trainer(model=model,
                  args=training_args,
                  train_dataset=dataset["train"],
                  eval_dataset=dataset["validation"],
                  compute_metrics=compute_metrics)
trainer.train()
```

```
{'epoch': 1.0, 'eval_loss': 0.79, 'eval_accuracy': 0.00, 'eval_f1': 0.00}
{'epoch': 2.0, 'eval_loss': 0.65, 'eval_accuracy': 0.11, 'eval_f1': 0.15}
```

```
print(trainer.args.device)
```

```
cpu
```

Running sentiment analysis for e-commerce

```
sample_review = "This product is amazing!"
input_ids = tokenizer.encode(sample_review, return_tensors='pt')
print(f"Tokenized review: {input_ids}")
```

```
Tokenized review: tensor([[ 101, 2023, 4031, 2003, 6429,  999,  102 ]])
```

Running sentiment analysis for e-commerce

```
output = model(input_ids)
print(f"Output logits: {output.logits}")
```

```
Output logits: tensor([[ -0.0538, 0.1300 ]])
```

```
predicted_label = torch.argmax(output.logits, dim=1).item()
print(f"Predicted label: {predicted_label}")
```

```
Predicted label: 1
```

```
sentiment = "Negative" if predicted_label == 0 else "Positive"
print(f'The sentiment of the product review is "{sentiment}."')
```

```
The sentiment of the product review is "Positive."
```

Checkpoints with Trainer

- Resume from the latest checkpoint, like pausing a movie

```
trainer.train(resume_from_checkpoint=True)
```

```
{'epoch': 3.0, 'eval_loss': 0.29, 'eval_accuracy': 0.37, 'eval_f1': 0.51}
{'epoch': 4.0, 'eval_loss': 0.23, 'eval_accuracy': 0.46, 'eval_f1': 0.58}
```

- Resume from specific checkpoint saved in output directory

```
trainer.train(resume_from_checkpoint="model/checkpoint-1000")
```

Let's practice!

EFFICIENT AI MODEL TRAINING WITH PYTORCH

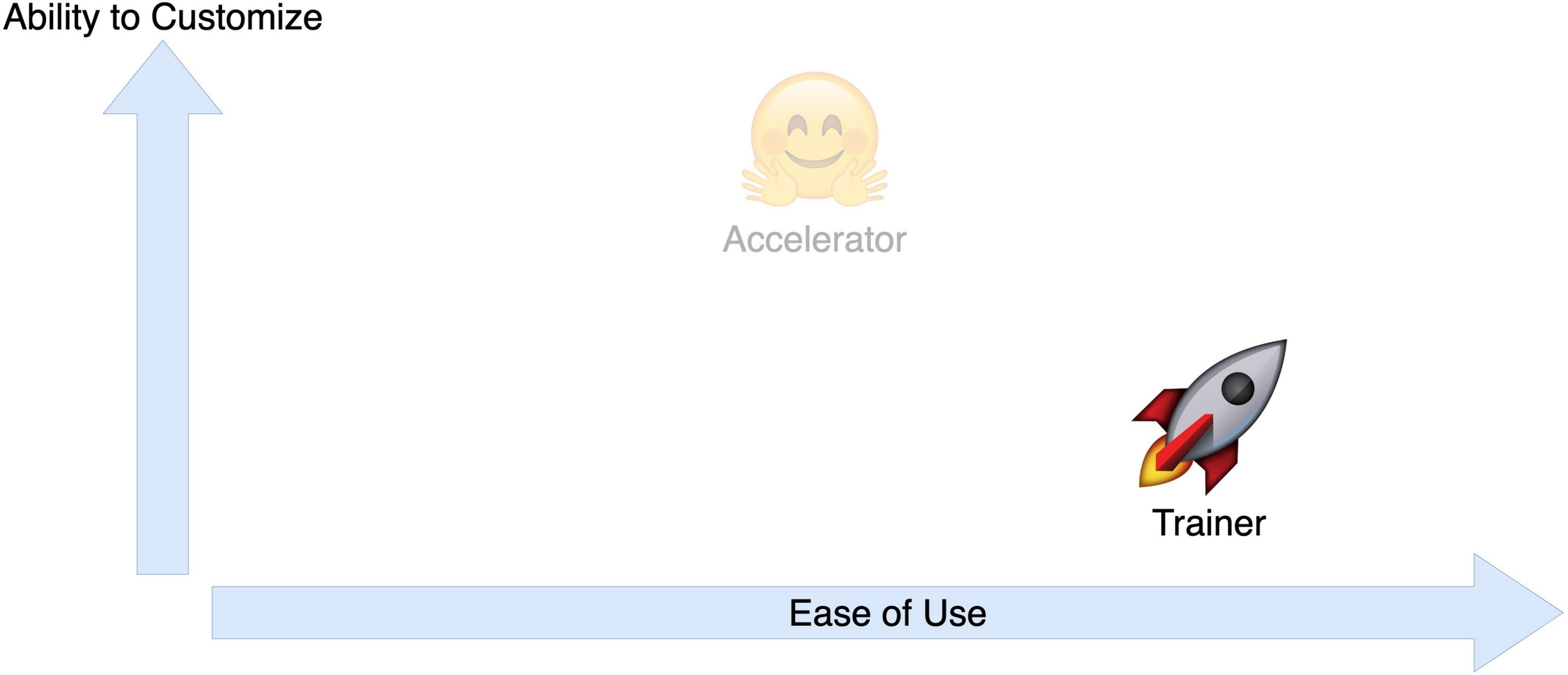
Train models with Accelerator

EFFICIENT AI MODEL TRAINING WITH PYTORCH

Dennis Lee
Data Engineer

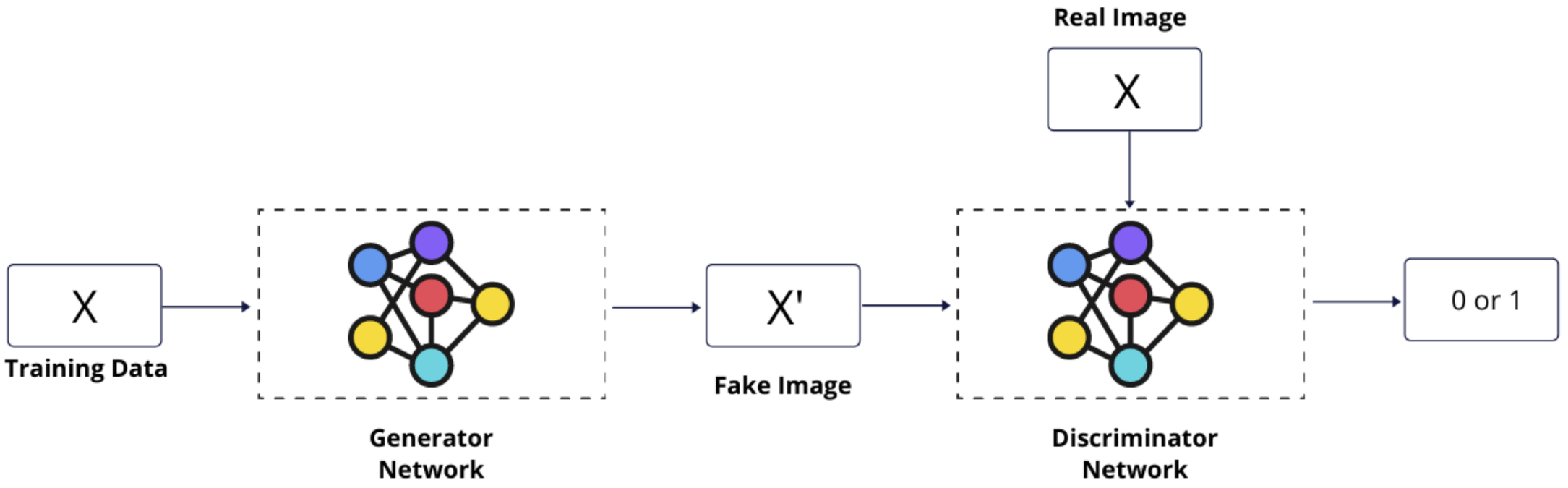
datacamp

Trainer and Accelerator



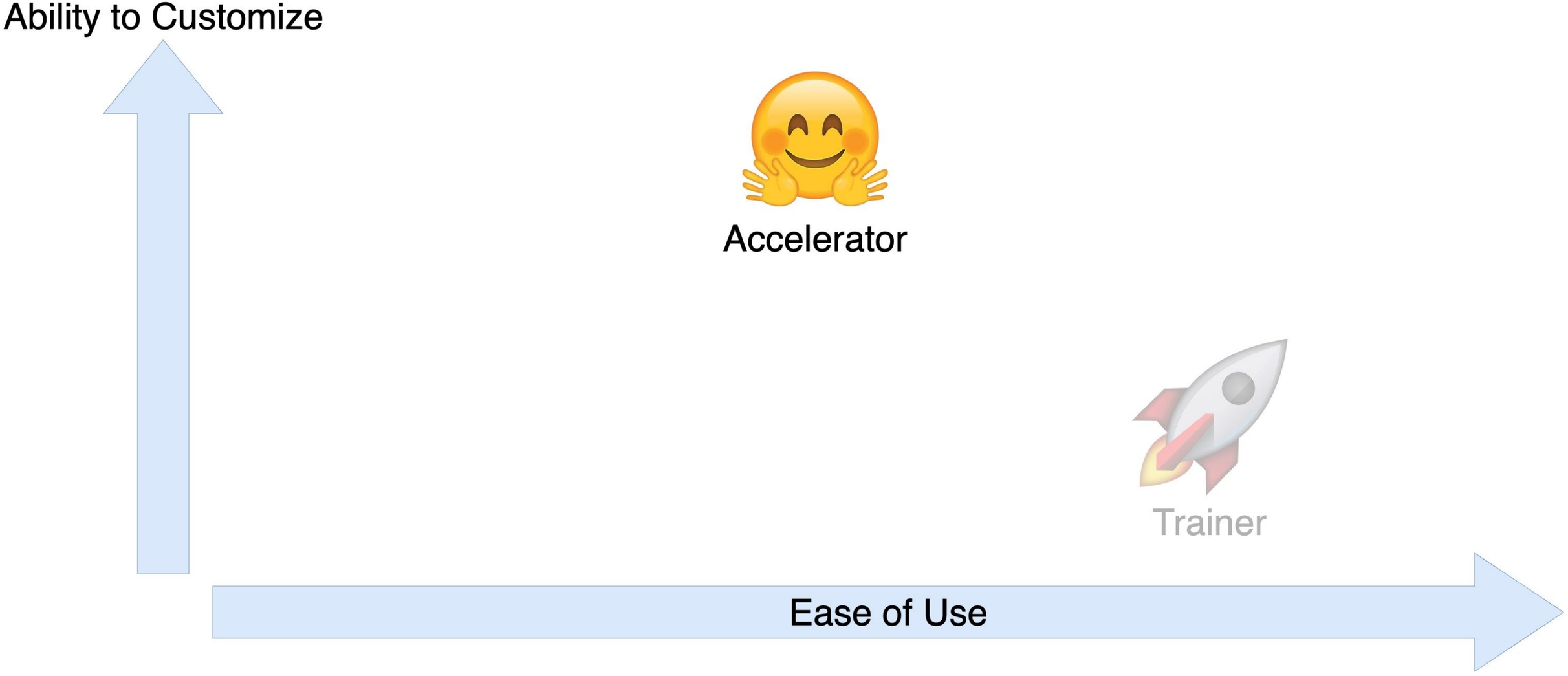
Custom training loops

- Trainer: no custom training loops
- Some advanced tasks in generative AI require two networks



[1] https://www.aitude.com/basics-of-generative-adversarial-network-model/

Trainer and Accelerator



Modifying a basic training loop

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

- Zero the gradients
- Move data to a specified device: `.to(device)`
- Perform forward pass
- Compute cross-entropy loss
- Compute gradients in a backward pass
- Update model parameters, learning rate

Create an Accelerator object

- `Accelerator` provides an interface for distributed training

```
from accelerate import Accelerator
accelerator = Accelerator(
    device_placement=True
)
```

- `device_placement` (`bool`, default `True`): Handle device placement by default

Define the model and optimizer

- Load a pre-trained model

```
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(
    "distilbert-base-cased", return_dict=True)
```

- Optimize model parameters with `Adam`

```
from torch.optim import Adam

optimizer = Adam(params=model.parameters(), lr=2e-5)
```

Define the scheduler

```
from transformers import get_linear_schedule_with_warmup

lr_scheduler = get_linear_schedule_with_warmup(
      optimizer=optimizer,
      num_warmup_steps=num_warmup_steps,
      num_training_steps=num_training_steps)
```

- `optimizer` (`obj`): PyTorch optimizer, like `Adam`
- `num_warmup_steps` (`int`): steps to linearly increase `lr`, set to `int(num_training_steps * 0.1)`
- `num_training_steps` (`int`): total training steps, set to `len(train_dataloader) * num_epochs`

Prepare the model for efficient training

- The `prepare` method handles device placement

```
model, optimizer, dataloader, lr_scheduler = \
    accelerator.prepare(model,
                        optimizer,
                        dataloader,
                        lr_scheduler)
```

Building a training loop with Accelerator

```
for batch in dataloader:
    optimizer.zero_grad()
    inputs, targets = batch
    inputs = inputs.to(device)
    targets = targets.to(device)
```

- Zero the gradients
- Previously moved data to the device

Building a training loop with Accelerator

```
for batch in dataloader:
    optimizer.zero_grad()
    inputs, targets = batch
```

- Zero the gradients
- Previously moved data to the device
- Remove lines that manually move data

Building a training loop with Accelerator

```
for batch in dataloader:
    optimizer.zero_grad()
    inputs, targets = batch
    outputs = model(inputs)
    loss = outputs.loss
    loss.backward()
```

- Zero the gradients
- Previously moved data to the device
- Remove lines that manually move data
- Perform a forward pass
- Compute cross-entropy loss and gradients

Building a training loop with Accelerator

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

- Zero the gradients
- Previously moved data to the device
- Remove lines that manually move data
- Perform a forward pass
- Compute cross-entropy loss and gradients
- Replace `loss.backward` with `accelerator`
- Update model parameters, learning rate

Summary of changes

Before Accelerator

- Need to manually move data to devices
  - `inputs.to(device)`
  - `targets.to(device)`
- Compute gradients with `loss.backward()`

After Accelerator

- Automatic device placement and data parallelism
  - `accelerator.prepare(model)`
  - `accelerator.prepare(dataloader)`
- Handle gradient synchronization with `accelerator.backward(loss)`
- Customizable loop
- User-friendly, hardware-agnostic, scalable, and maintainable

Let's practice!

EFFICIENT AI MODEL TRAINING WITH PYTORCH

Evaluate models with Accelerator

EFFICIENT AI MODEL TRAINING WITH PYTORCH

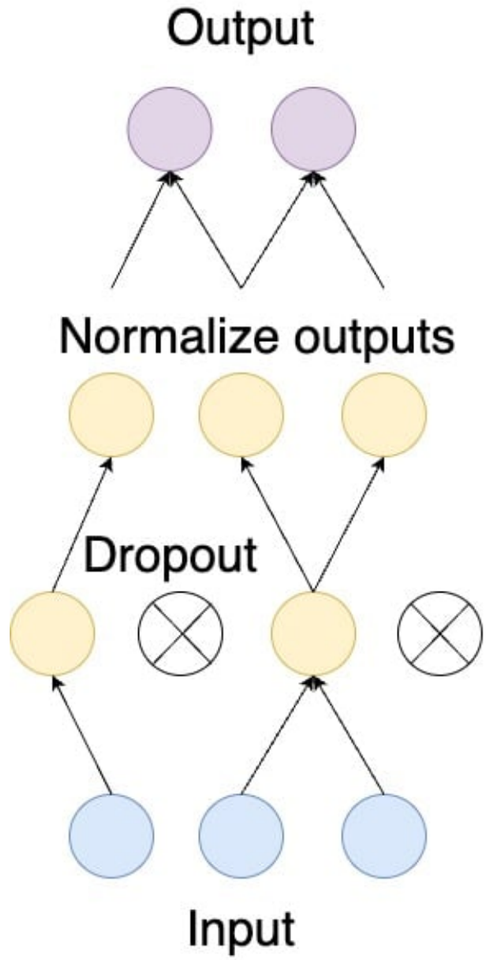
Dennis Lee
Data Engineer

datacamp

Why put a model in evaluation mode?

- Training mode
  - Dropout: Set neurons to zero
  - Batch normalization

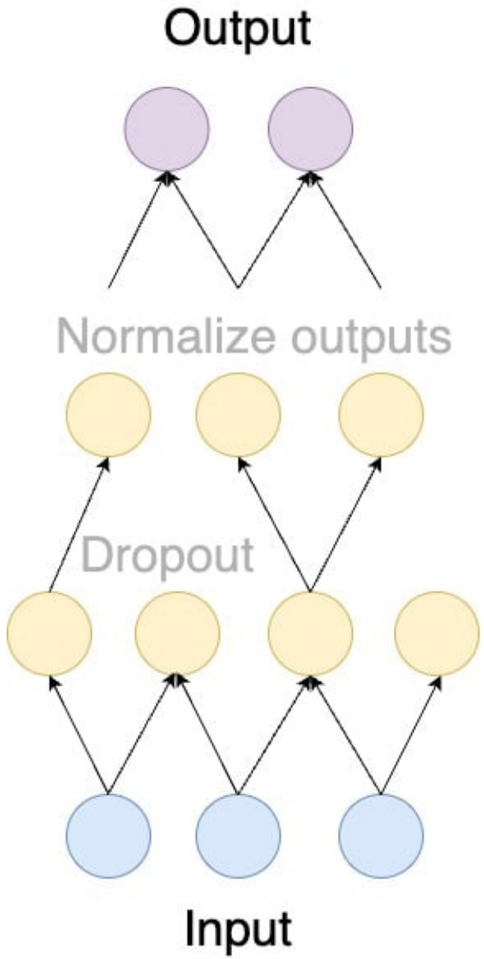
Dropout and batch normalization



Why put a model in evaluation mode?

- Training mode
  - Dropout: Set neurons to zero
  - Batch normalization
- Evaluation mode disables these layers
- `model.eval()` activates evaluation mode

Dropout and batch normalization

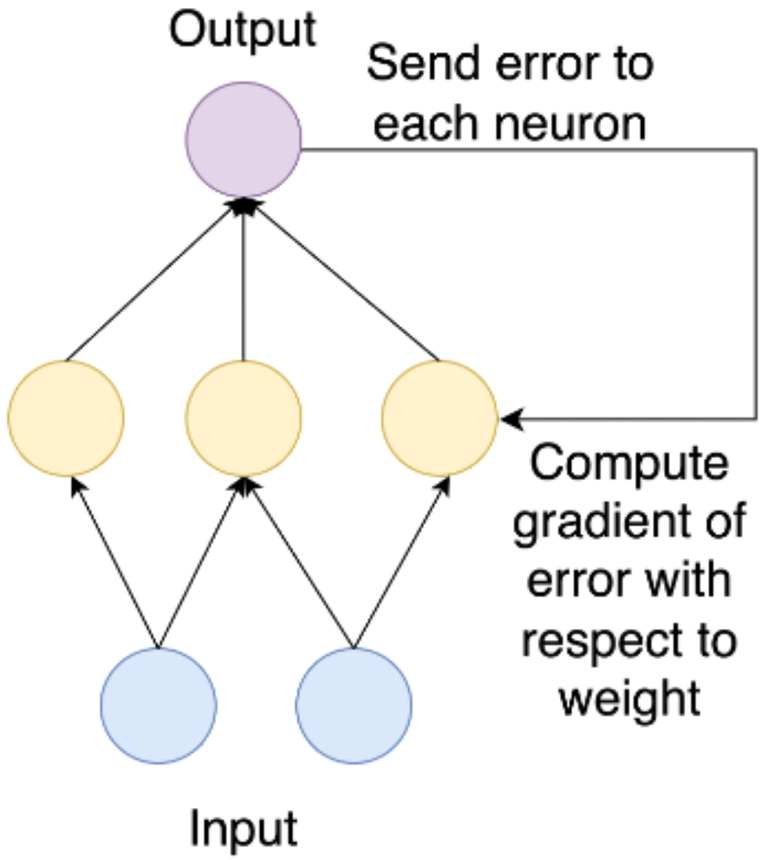


Disable gradients with torch.no_grad()

- Training requires gradient computation
- `torch.no_grad()` disables gradients
- Call both `model.eval` and `torch.no_grad`:

```
model.eval()
with torch.no_grad():
    outputs = model(**inputs)
```

Computing gradients in backpropagation



Prepare a validation dataset

- Load validation split of the MRPC dataset

```
validation_dataset = load_dataset("glue", "mrpc", split="validation")
```

- Tokenize the validation dataset

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

validation_dataset = validation_dataset.map(encode, batched=True)
```

Life of an epoch: training and evaluation loops

- For each epoch, iterate over the train and validation datasets
- First run the model in training mode
- Then run the model in evaluation mode and log metrics after evaluation

```
for epoch in range(num_epochs):
    model.train()
    for step, batch in enumerate(train_dataloader):
        # Perform training step
    model.eval()
    for step, batch in enumerate(eval_dataloader):
        # Perform evaluation step
    # Log evaluation metrics
```

Inside the evaluation loop

```
metric = evaluate.load("glue", "mrpc")
model.eval()
for step, batch in enumerate(eval_dataloader):
    with torch.no_grad():
        outputs = model(**batch)
    predictions = outputs.logits.argmax(dim=-1)
    predictions, references = accelerator.gather_for_metrics((predictions, batch["labels"]))
    metric.add_batch(predictions=predictions, references=references)
eval_metric = metric.compute()

print(f"Eval metrics: \n{eval_metric}")
```

```
Eval metrics:
{'accuracy': 0.81, 'f1': 0.77}
```

Log metrics after evaluation

- Tracking tools: notebooks that log metrics; examples are TensorBoard and MLflow
- `log_with` : use all experiment tracking tools
- `.init_trackers()` : initialize tracking tools
- `.log()` : track `accuracy` , `f1` , `epoch`
- `.end_training()` : finish tracking

```
accelerator = Accelerator(project_dir=".",
                          log_with="all")
accelerator.init_trackers("my_project")

for epoch in range(num_epochs):
    # Training loop is here
    # Evaluation loop is here
    accelerator.log({
        "accuracy": eval_metric["accuracy"],
        "f1": eval_metric["f1"],
    }, step=epoch)

accelerator.end_training()
```

Let's practice!

EFFICIENT AI MODEL TRAINING WITH PYTORCH