Balanced training with AdamW

EFFICIENT AI MODEL TRAINING WITH PYTORCH

Dennis Lee
Data Engineer

datacamp

Efficient training

Chapter 1
Data preparation

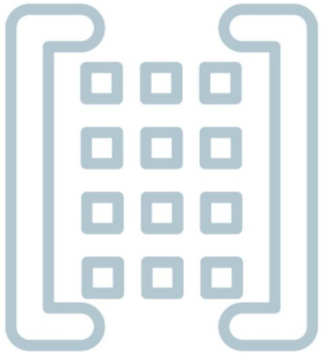
Chapter 2
Distributed training

Chapter 3
Efficient training

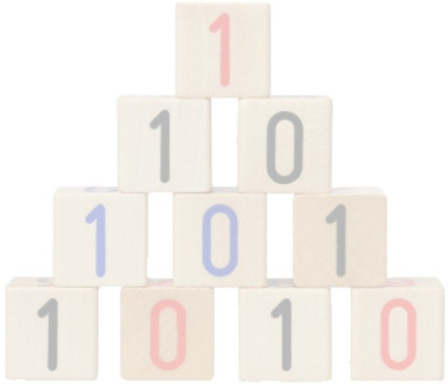
Chapter 4
Optimizers

Optimizers for training efficiency

AdamW



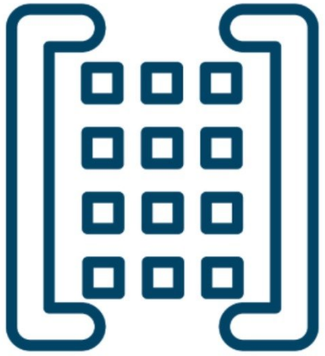
Adafactor



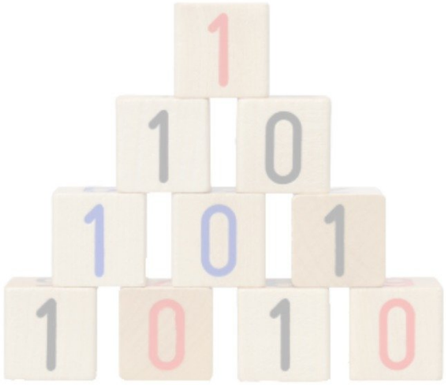
8-bit Adam

Optimizers for training efficiency

AdamW



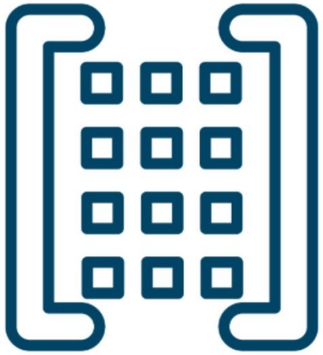
Adafactor



8-bit Adam

Optimizers for training efficiency

AdamW



Adafactor

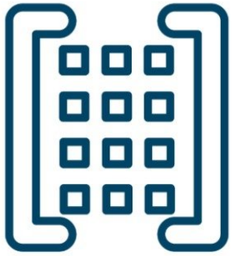
8-bit Adam

Optimizer tradeoffs

More parameters

8-bit Adam

AdamW

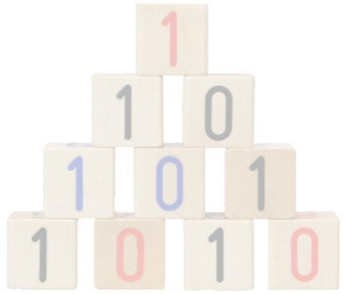


Adafactor

Higher precision

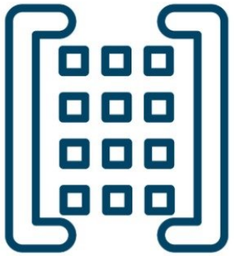
Optimizer tradeoffs

More parameters



8-bit Adam

AdamW

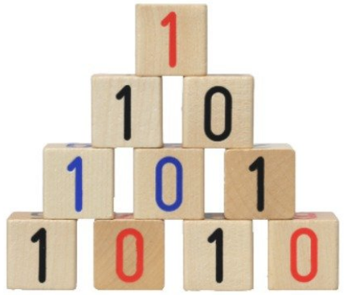


Adafactor

Higher precision

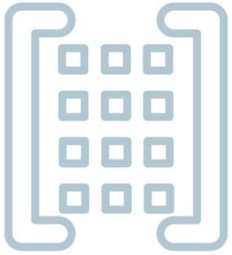
Optimizer tradeoffs

More parameters



8-bit Adam

AdamW

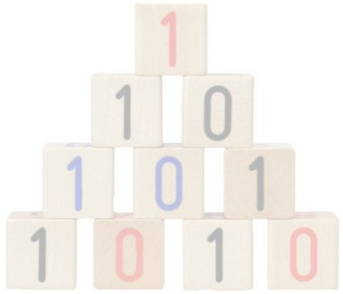


Adafactor

Higher precision

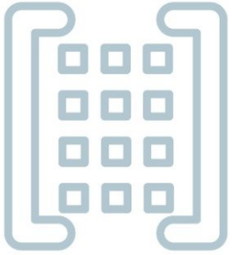
Optimizer tradeoffs

More parameters



8-bit Adam

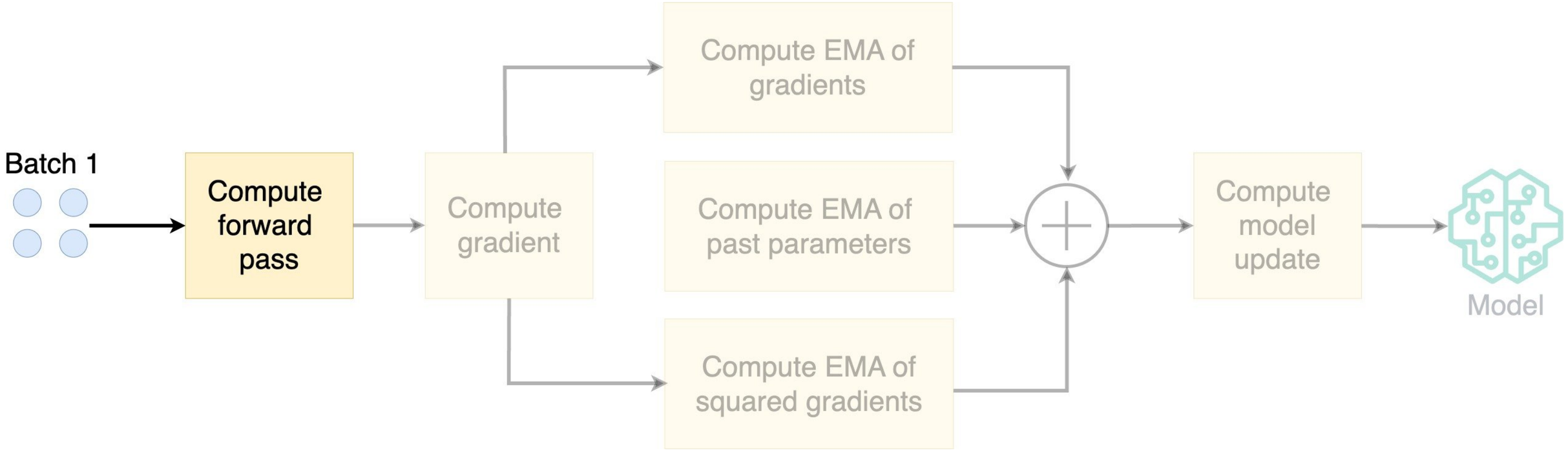
AdamW



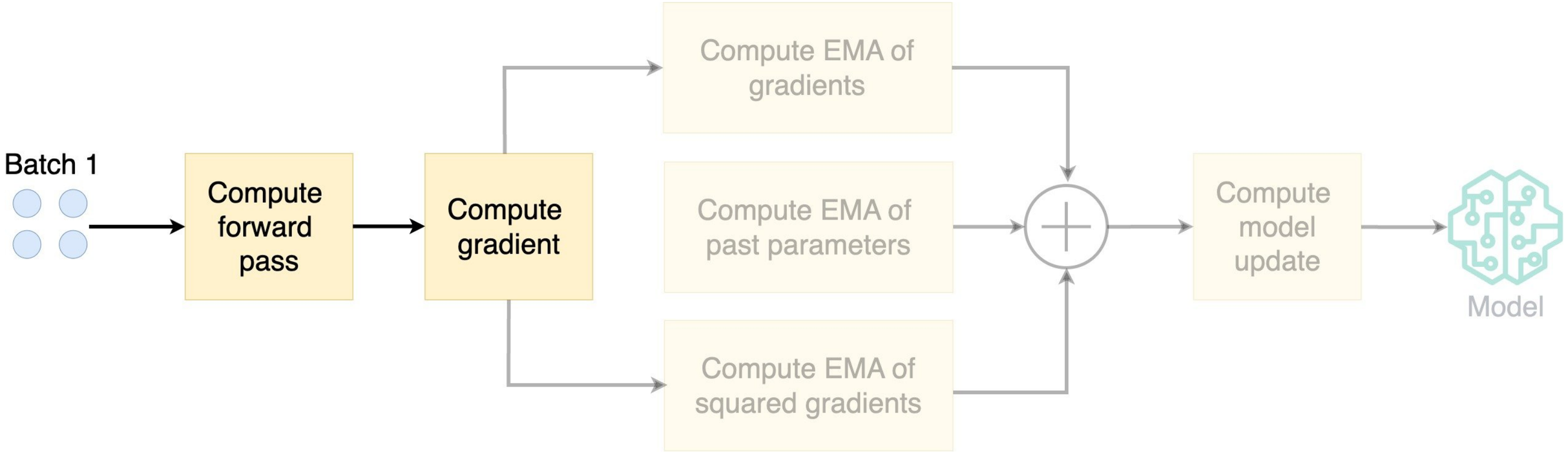
Adafactor

Higher precision

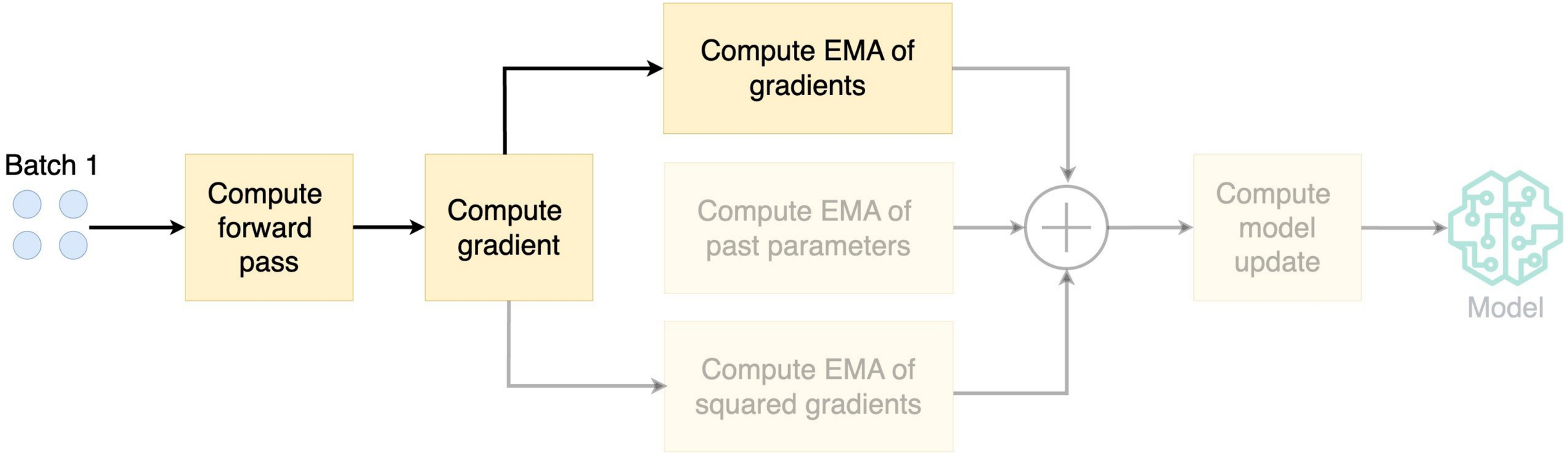
How does AdamW work?



How does AdamW work?

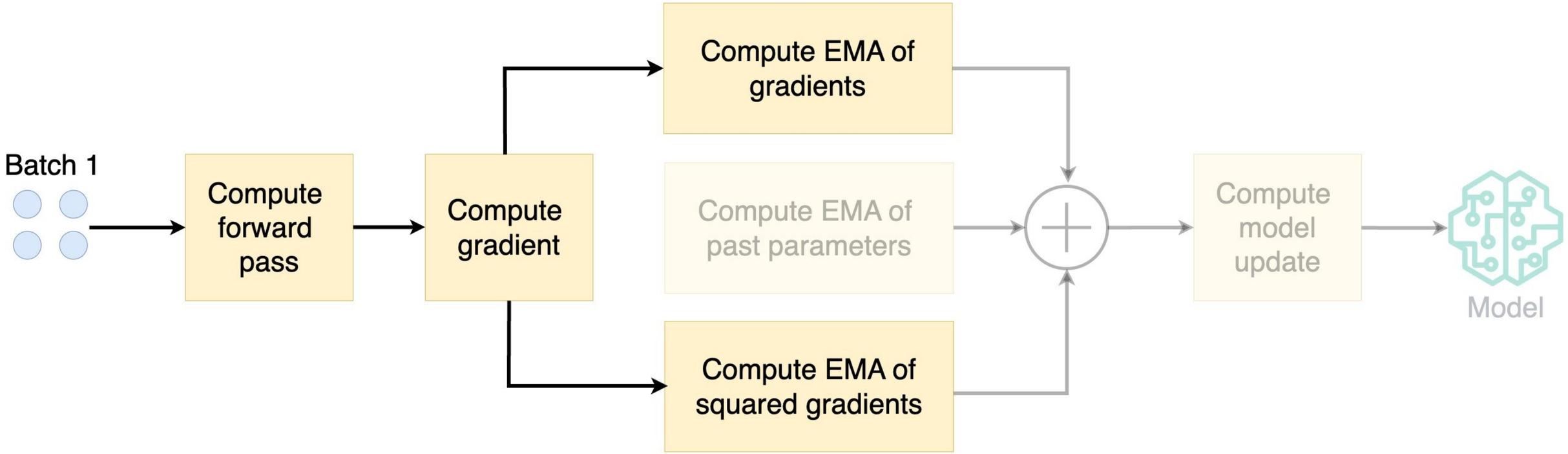


How does AdamW work?



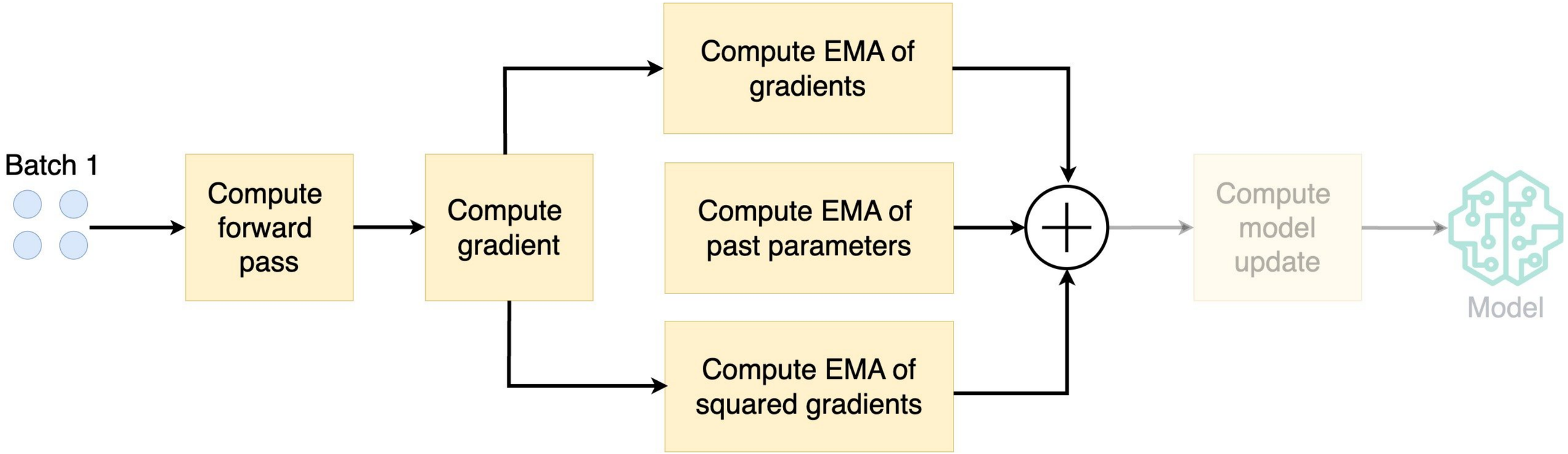
- Compute the exponential moving average (EMA) of the gradients

How does AdamW work?



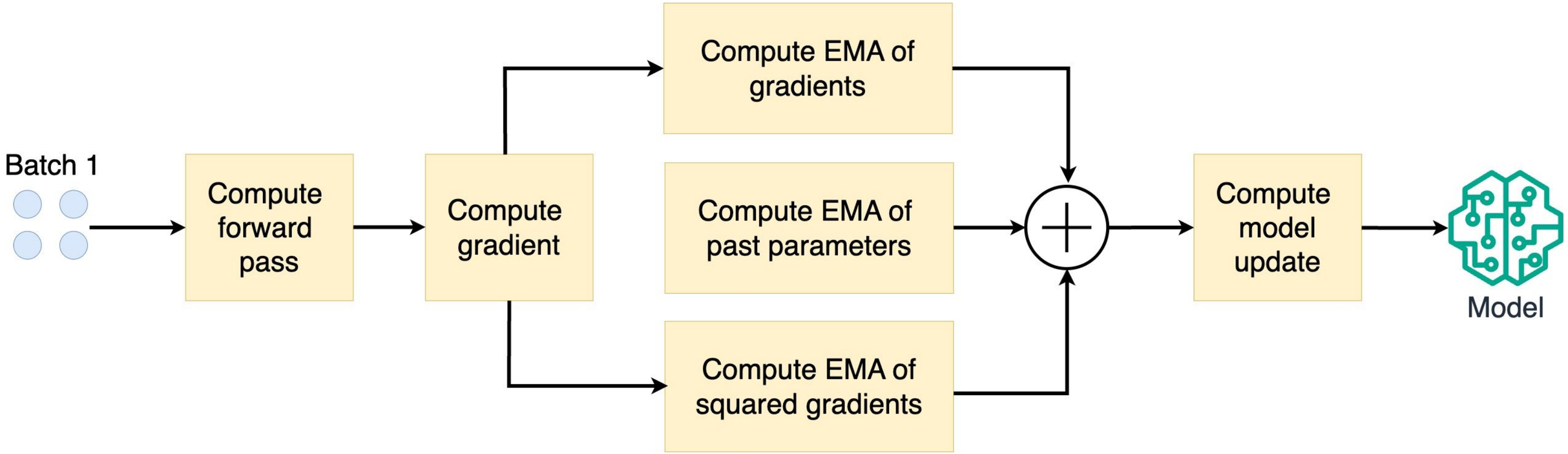
- Compute the exponential moving average (EMA) of the gradients
- Compute EMA of squared gradients

How does AdamW work?



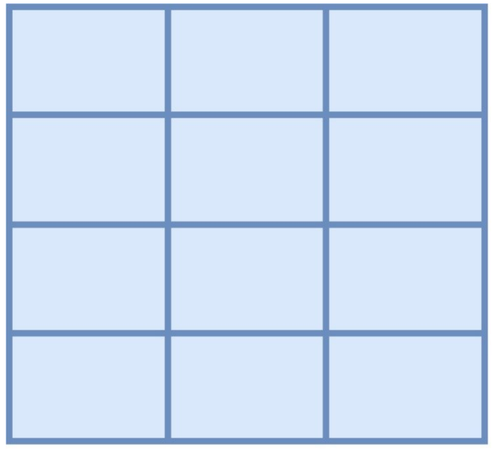
- Compute the exponential moving average (EMA) of the gradients
- Compute EMA of squared gradients

How does AdamW work?

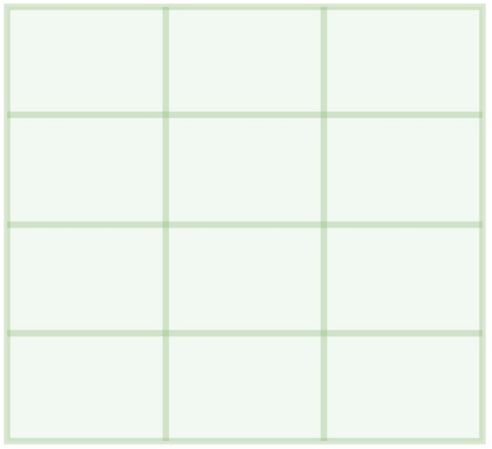


- Compute the exponential moving average (EMA) of the gradients
- Compute EMA of squared gradients

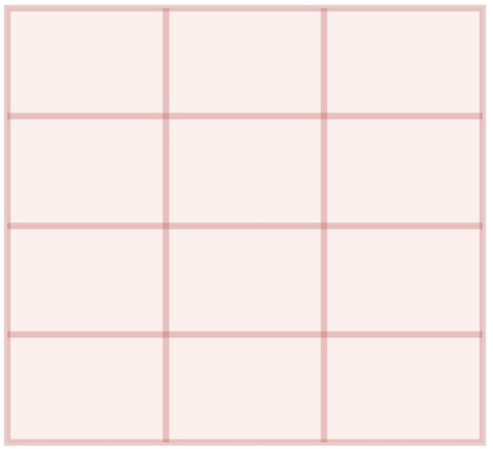
Memory usage of AdamW



Parameter gradients



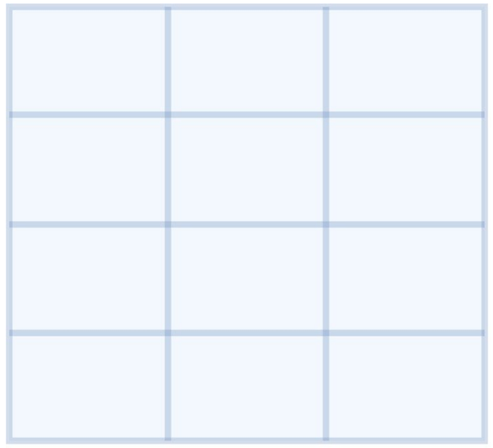
EMA of gradients



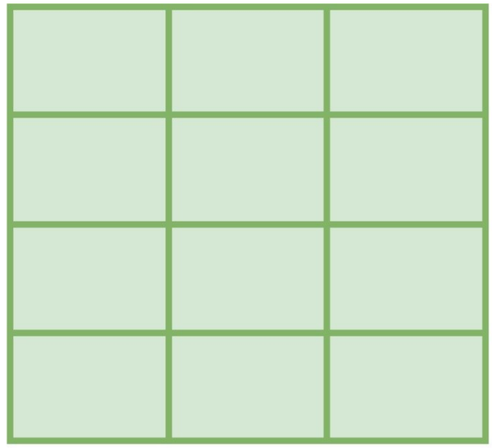
EMA of squared gradents

- Each square is a parameter, and each color is a state

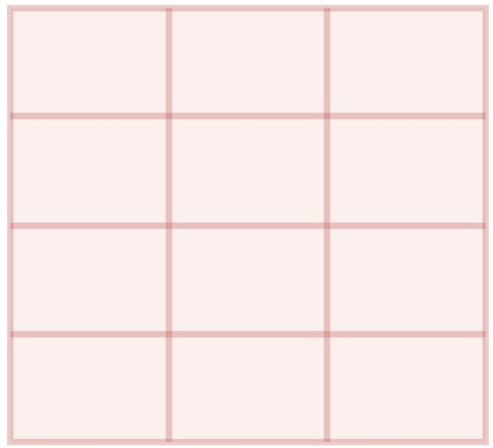
Memory usage of AdamW



Parameter gradients



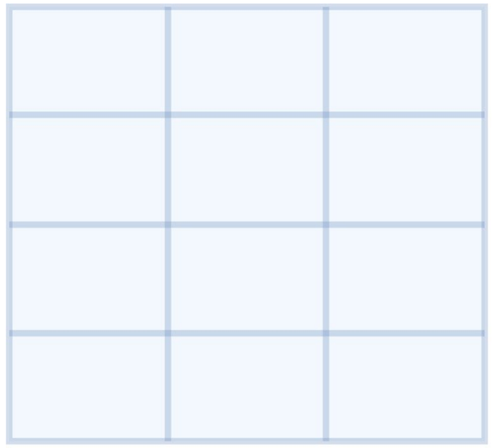
EMA of gradients



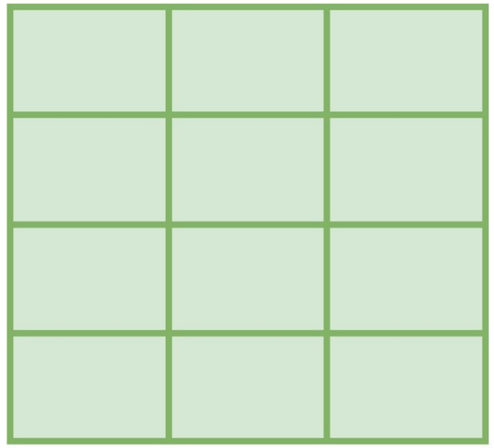
EMA of squared gradents

- Each square is a parameter, and each color is a state

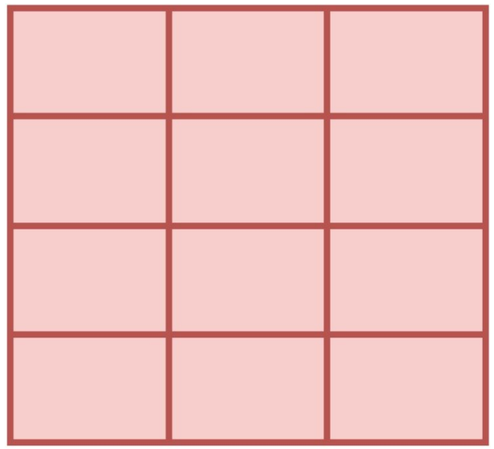
Memory usage of AdamW



Parameter gradients



EMA of gradients



EMA of squared gradents

- Each square is a parameter, and each color is a state
- Memory per parameter = 8 bytes = 4 bytes per state * 2 states
- Total memory = Memory per parameter (8 bytes) * Number of parameters

Estimate memory usage of AdamW

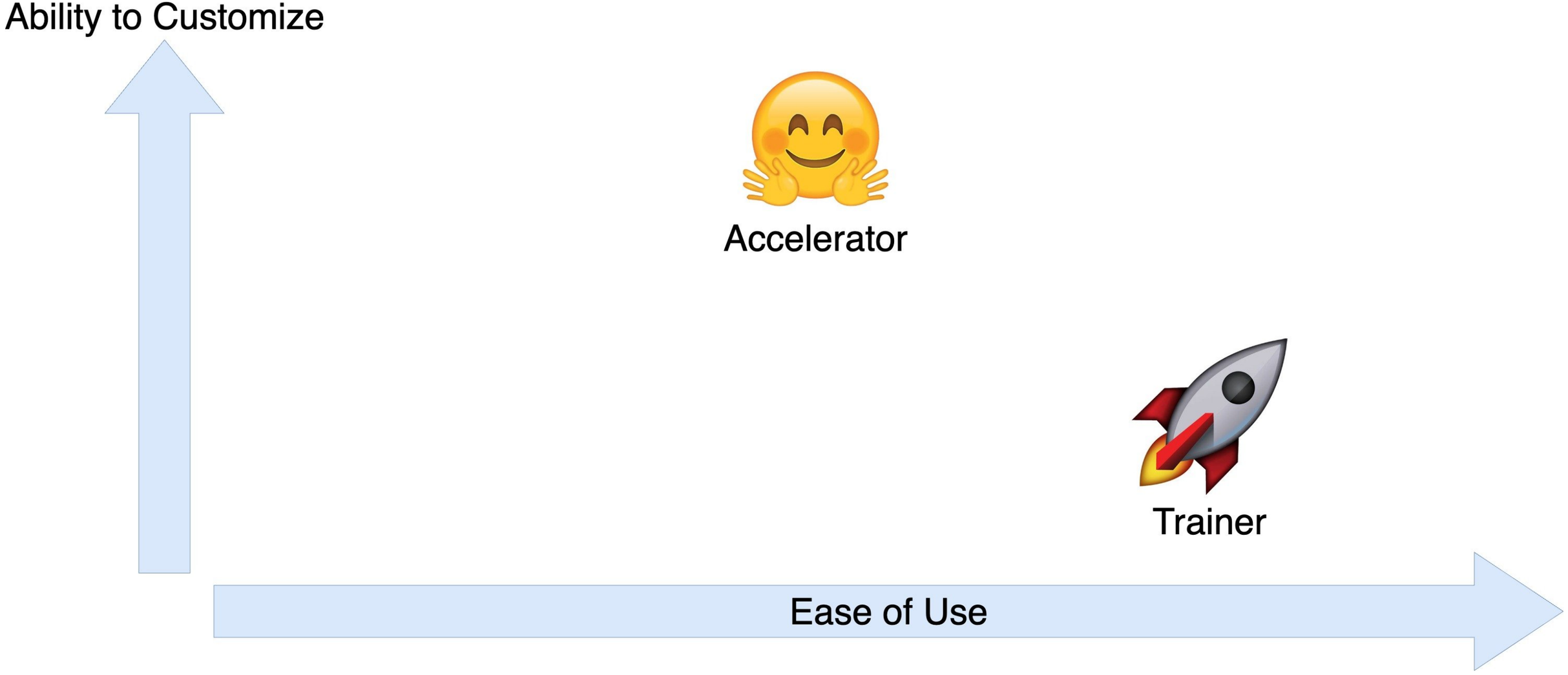
```
model = AutoModelForSequenceClassification.from_pretrained(
    "distilbert-base-cased", return_dict=True)
num_parameters = sum(p.numel() for p in model.parameters())
print(f"Number of model parameters: {num_parameters:,}")
```

```
Number of model parameters: 65,783,042
```

```
estimated_memory = num_parameters * 8 / (1024 ** 2)
print(f"Estimated memory usage of AdamW: {estimated_memory:.0f} MB")
```

```
Estimated memory usage of AdamW: 502 MB
```

Trainer and Accelerator



Implement AdamW with Trainer

```
from torch.optim import AdamW

optimizer = AdamW(params=model.parameters())

trainer = Trainer(model=model,
                  args=training_args,
                  train_dataset=train_dataset,
                  eval_dataset=validation_dataset,
                  compute_metrics=compute_metrics,
                  optimizers=(optimizer, lr_scheduler))
trainer.train()
```

```
{'epoch': 1.0, 'eval_accuracy': 0.7, 'eval_f1': 0.8}
```

Implement AdamW with Accelerator

```
from torch.optim import AdamW

optimizer = AdamW(params=model.parameters())

for batch in train_dataloader:
    inputs, targets = batch["input_ids"], batch["labels"]
    outputs = model(inputs, labels=targets)
    loss = outputs.loss
    accelerator.backward(loss)
    optimizer.step()
    lr_scheduler.step()
    optimizer.zero_grad()
    print(f"Loss = {loss}")
```

```
Loss = 0.7
```

Inspecting the optimizer state

```
optimizer_state = optimizer.state.values()
print(optimizer_state)
```

```
dict_values([{'step': tensor(3.),
              'exp_avg': tensor([[0., 0., 0.,  ..., 0., 0., 0.], ...]),
              'exp_avg_sq': tensor([[0., 0., 0.,  ..., 0., 0., 0.], ...])}, ...])
```

Computing the optimizer size

```
def compute_optimizer_size(optimizer_state):
    total_size_megabytes, total_num_elements = 0, 0
    for params in optimizer_state:
        for name, tensor in params.items():
            tensor = torch.tensor(tensor)
            num_elements = tensor.numel()
            element_size = tensor.element_size()
            total_num_elements += num_elements
            total_size_megabytes += num_elements * element_size / (1024 ** 2)
    return total_size_megabytes, total_num_elements
```

Computing the optimizer size

```
total_size_megabytes, total_num_elements = \
    compute_optimizer_size(trainer.optimizer.state.values())
print(f"Number of optimizer parameters: {total_num_elements:,}")
```

```
Number of optimizer parameters: 131,566,188
```

```
print(f"Optimizer size: {total_size_megabytes:.0f} MB")
```

```
Optimizer size: 502 MB
```

Let's practice!

EFFICIENT AI MODEL TRAINING WITH PYTORCH

Memory-efficient training with Adafactor

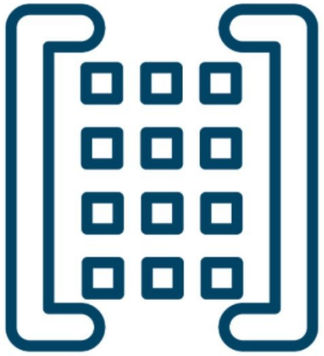
EFFICIENT AI MODEL TRAINING WITH PYTORCH

Dennis Lee
Data Engineer

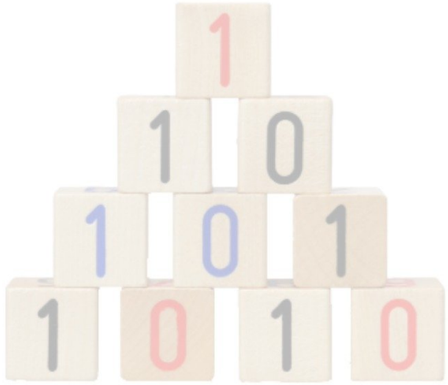
datacamp

Optimizers for training efficiency

AdamW



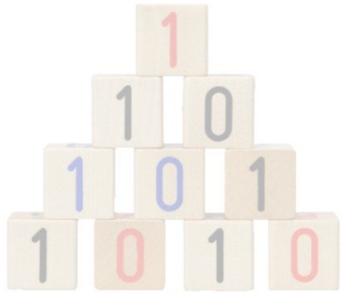
Adafactor



8-bit Adam

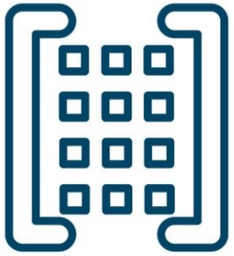
Optimizer tradeoffs

More parameters



8-bit Adam

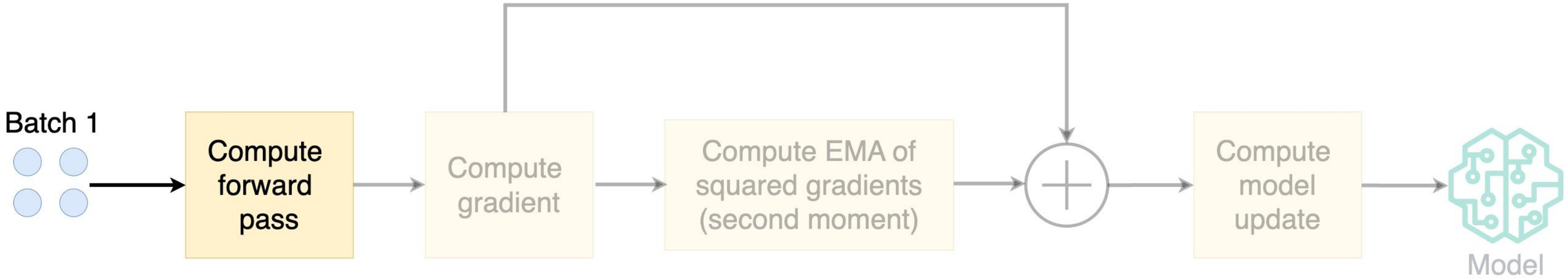
AdamW



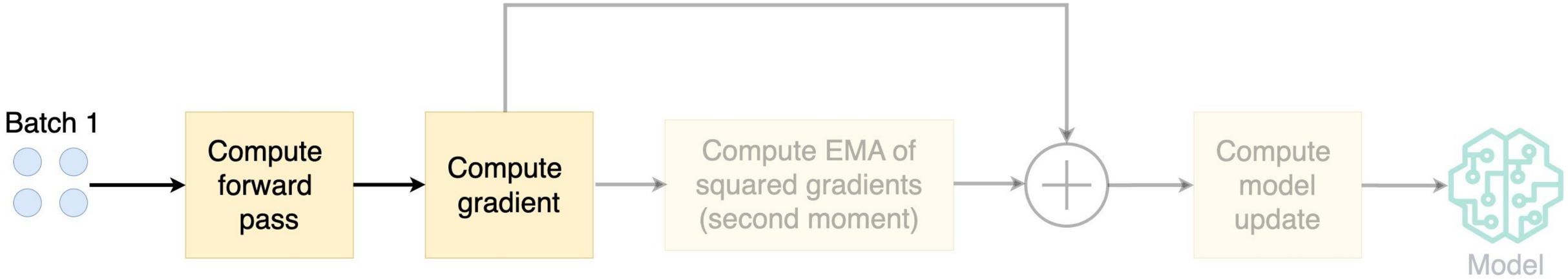
Adafactor

Higher precision

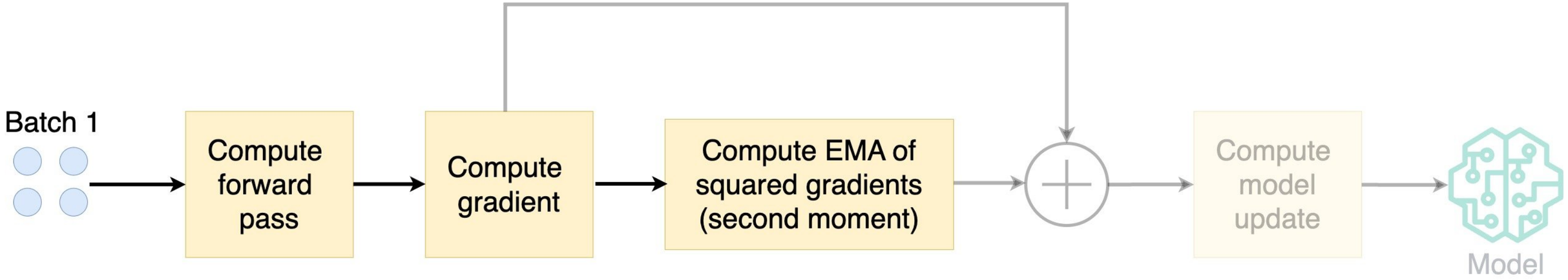
How does Adafactor work?



How does Adafactor work?

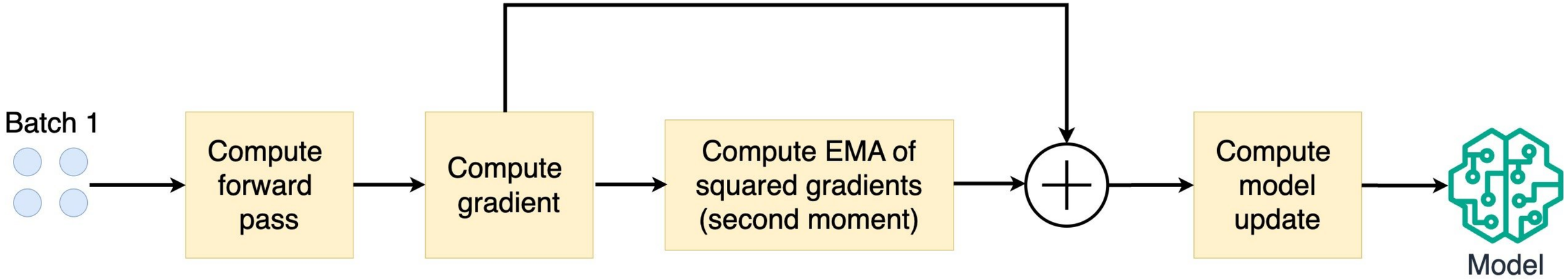


How does Adafactor work?



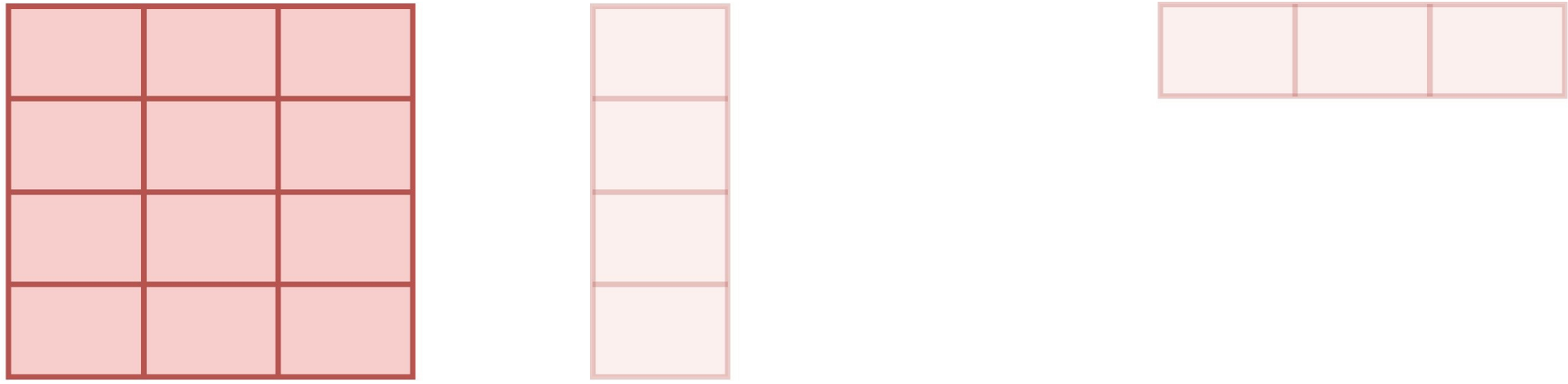
- EMA: exponential moving average
- Second moment: EMA of the squared gradients

How does Adafactor work?



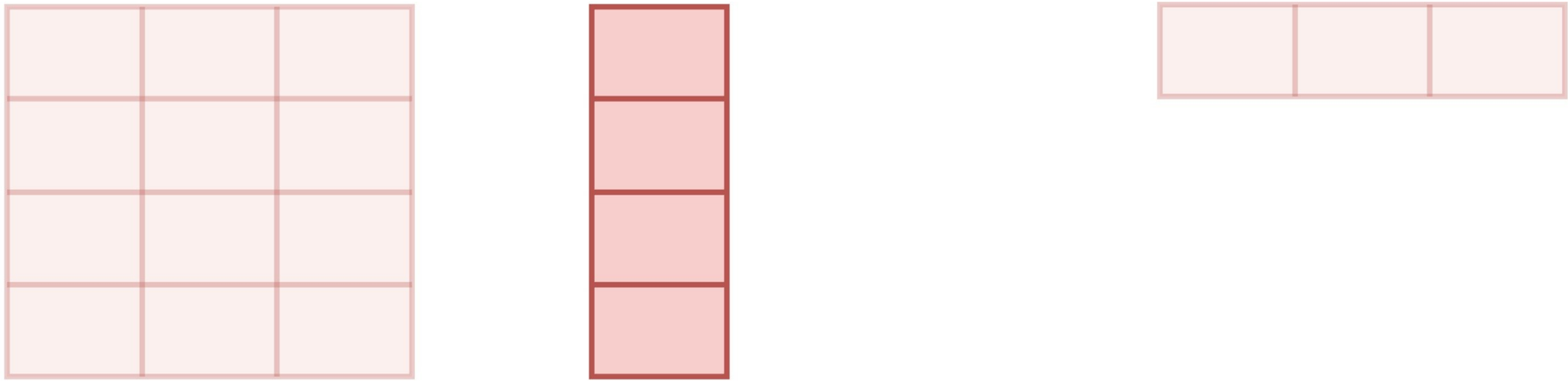
- EMA: exponential moving average
- Second moment: EMA of the squared gradients

How does Adafactor save memory?



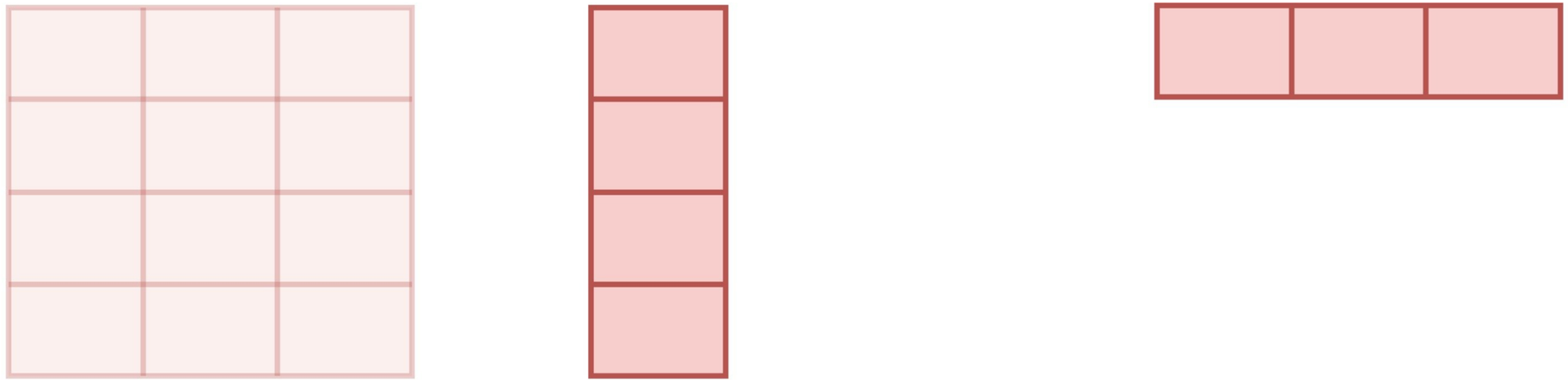
- Save memory by not storing the second moment matrix

How does Adafactor save memory?



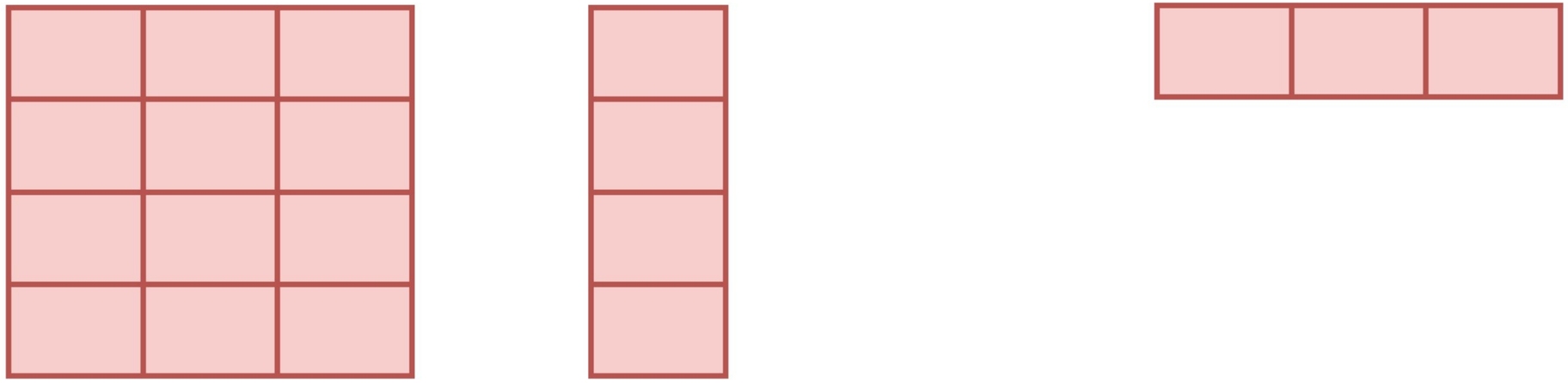
- Save memory by not storing the second moment matrix
- Instead, store the column sum and row sum of the matrix

How does Adafactor save memory?



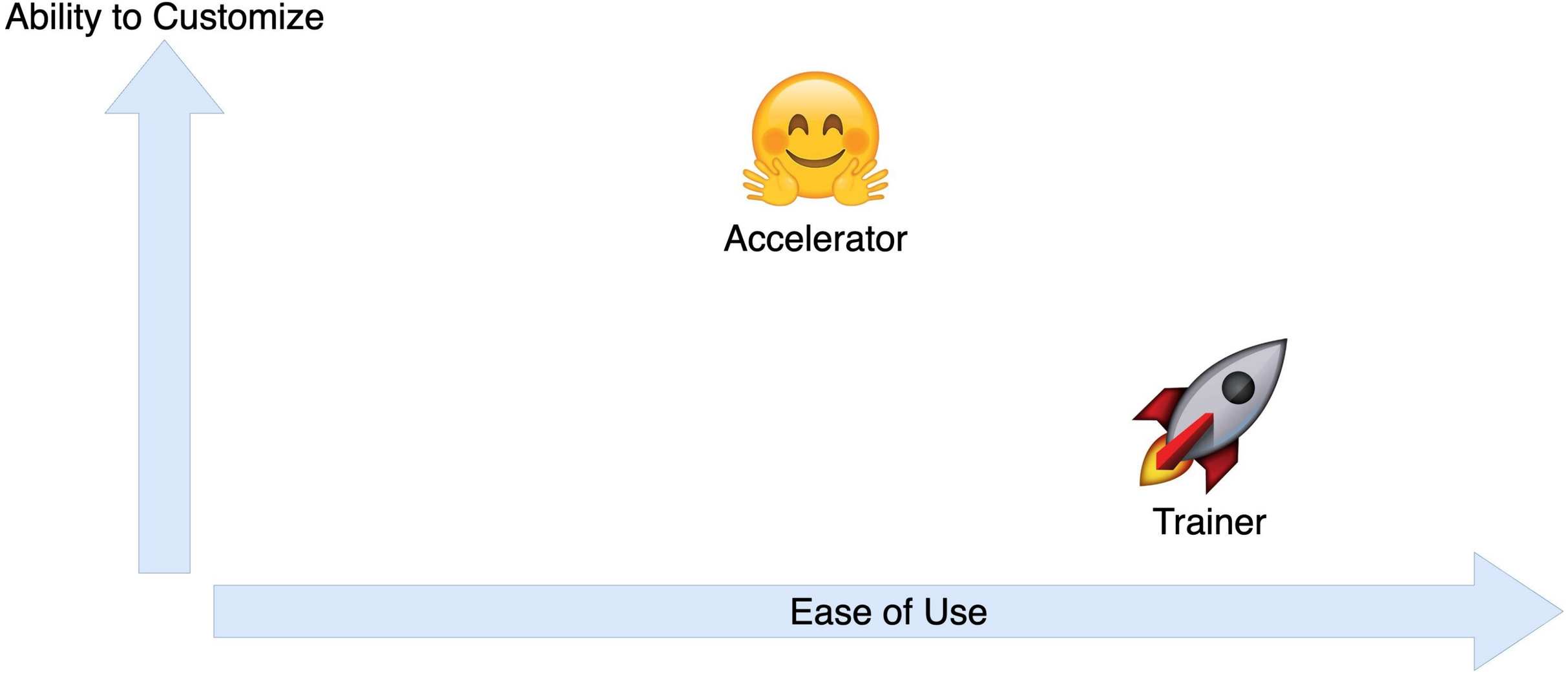
- Save memory by not storing the second moment matrix
- Instead, store the column sum and row sum of the matrix

How does Adafactor save memory?



- Save memory by not storing the second moment matrix
- Instead, store the column sum and row sum of the matrix
- Estimate full matrix by multiplying column sum and row sum

Trainer and Accelerator implementation



Implement Adafactor with Trainer

```
training_args = TrainingArguments(output_dir="./results",
                                  evaluation_strategy="epoch",
                                  optim="adafactor")
trainer = Trainer(model=model,
                  args=training_args,
                  train_dataset=train_dataset,
                  eval_dataset=validation_dataset,
                  compute_metrics=compute_metrics)
trainer.train()
```

```
{'epoch': 1.0, 'eval_accuracy': 0.6, 'eval_f1': 0.5}
```

Implement Adafactor with Accelerator

```
# Assumes PyTorch 2.5 or higher
from torch.optim import Adafactor
optimizer = Adafactor(params=model.parameters(), lr=lr)

for batch in train_dataloader:
    inputs, targets = batch["input_ids"], batch["labels"]
    outputs = model(inputs, labels=targets)
    loss = outputs.loss
    accelerator.backward(loss)
    optimizer.step()
    lr_scheduler.step()
    optimizer.zero_grad()
    print(f"Loss = {loss}")
```

```
Loss = 0.71
```

Inspect the optimizer state

- Access the `optimizer` through its `state`

```
optimizer_state = optimizer.state.values()
```

- Or access the `optimizer` through `trainer`

```
optimizer_state = trainer.optimizer.state.values()
```

```
print(optimizer_state)
```

```
dict_values([{'step': tensor(3.),
              'exp_avg_sq_row': tensor([1.0000e-30, 1.0000e-30, 1.0000e-30,  ...]),
              'exp_avg_sq_col': tensor([4.6147e-11, 5.5115e-11, 1.6338e-10, ...])}, ...])
```

Compute memory usage of Adafactor

```
total_size_megabytes, total_num_elements = \
    compute_optimizer_size(trainer.optimizer.state.values())
print(f"Number of Adafactor parameters: {total_num_elements:,}")
print(f"Adafactor size: {total_size_megabytes:.0f} MB")
```

```
Number of Adafactor parameters: 178,712
Adafactor size: 1 MB
```

- Compare to AdamW: Adafactor uses much less memory!

```
Number of AdamW parameters: 131,566,188
AdamW size: 502 MB
```

Let's practice!

EFFICIENT AI MODEL TRAINING WITH PYTORCH

Mixed precision training with 8-bit Adam

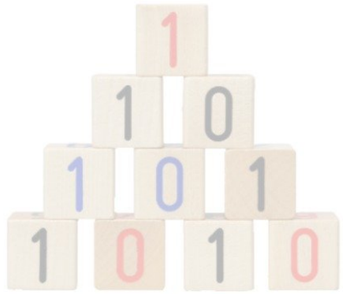
EFFICIENT AI MODEL TRAINING WITH PYTORCH

Dennis Lee
Data Engineer

datacamp

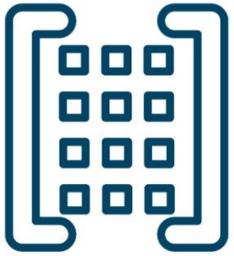
Optimizers for training efficiency

More parameters



8-bit Adam

AdamW

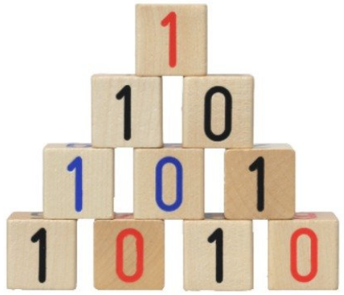


Adafactor

Higher precision

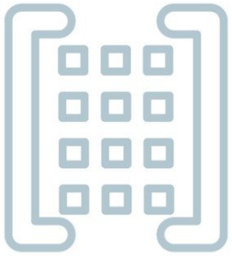
Optimizers for training efficiency

More parameters



8-bit Adam

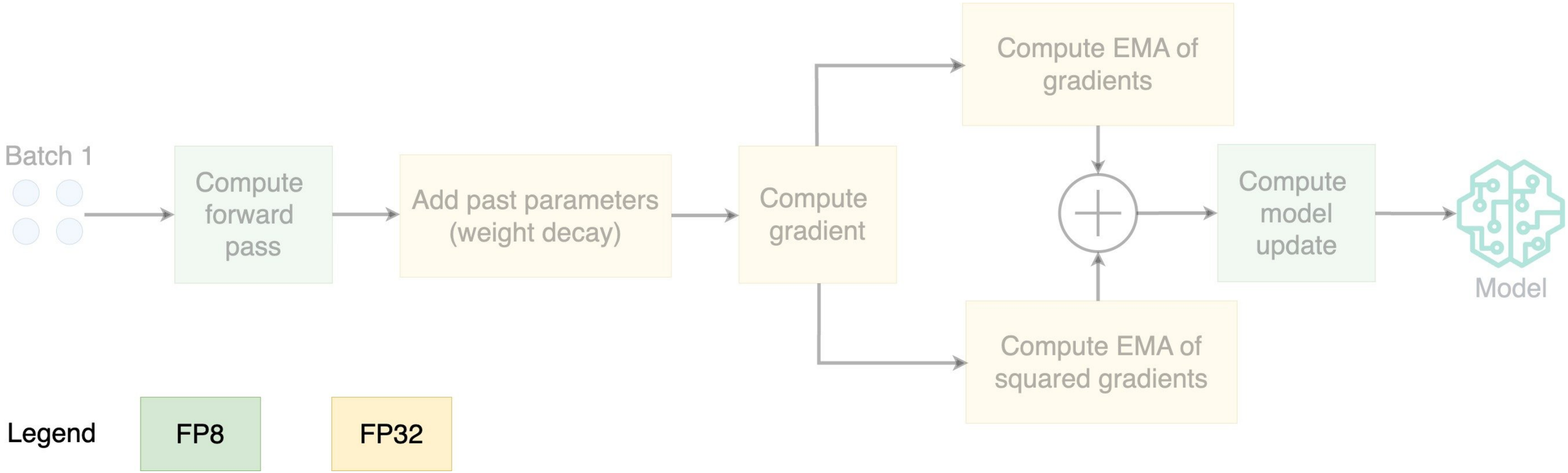
AdamW



Adafactor

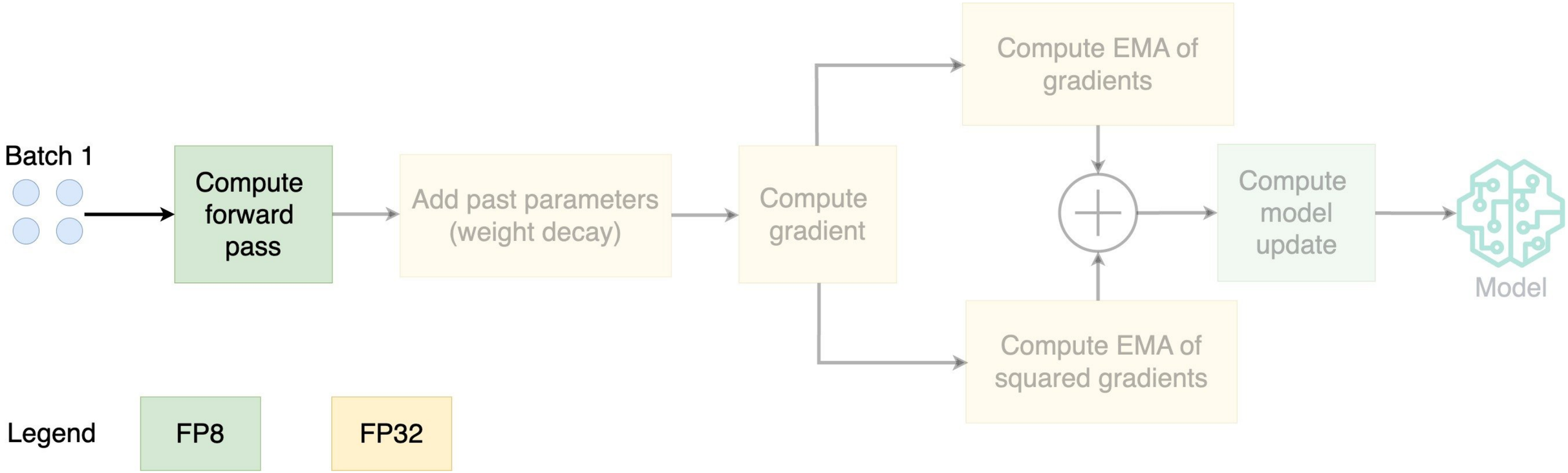
Higher precision

How does 8-bit Adam work?



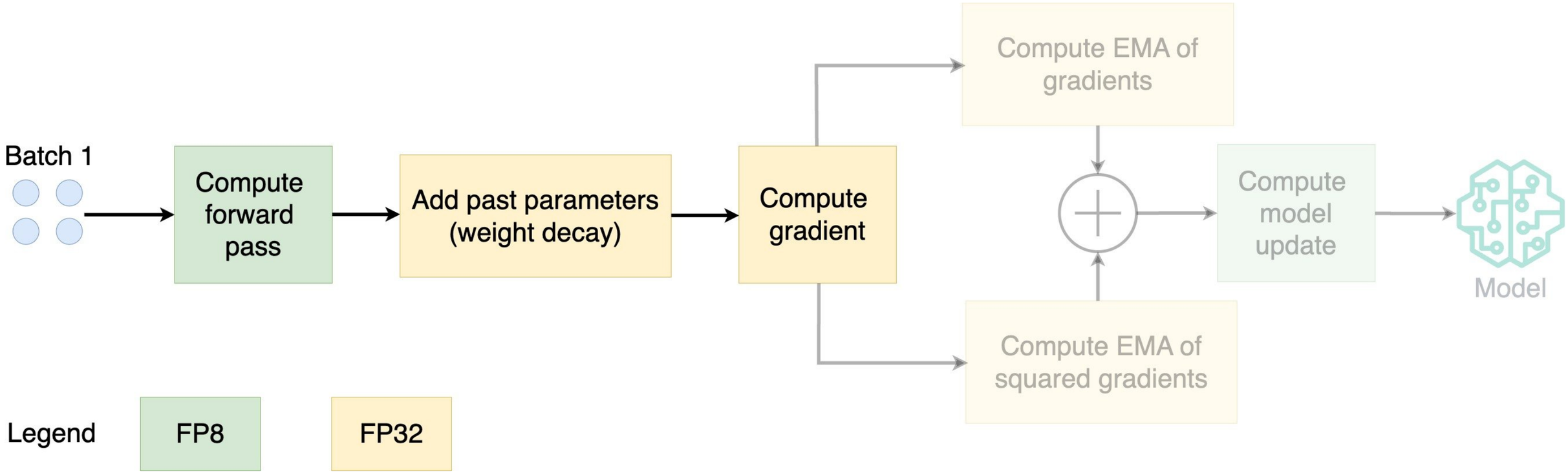
- Store parameters in FP8; optimize in FP32

How does 8-bit Adam work?



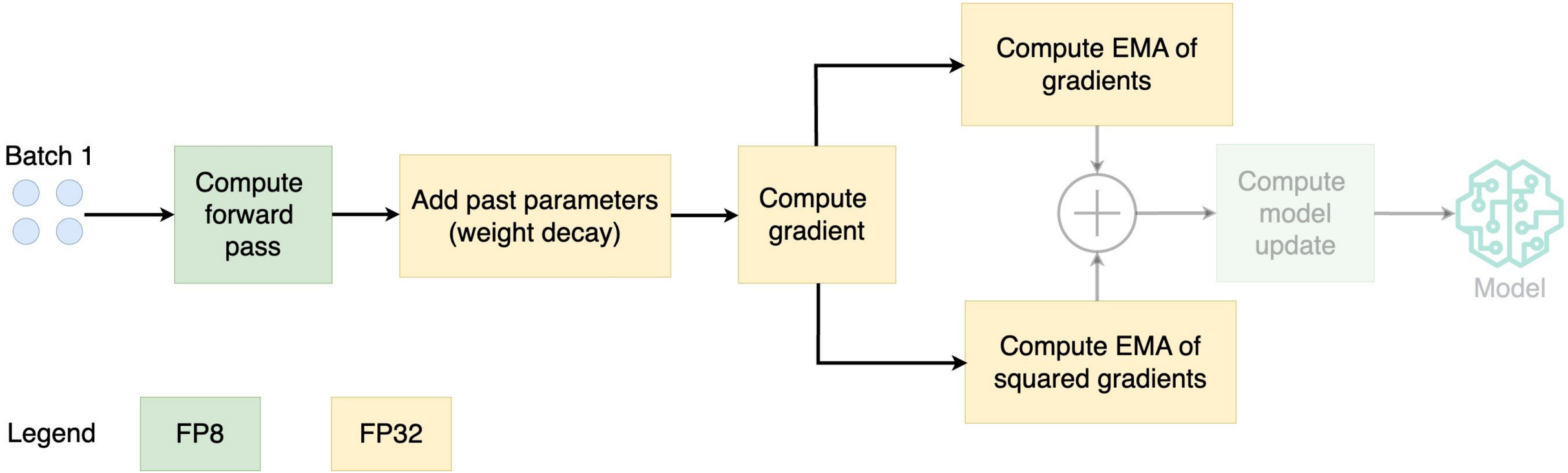
- Store parameters in FP8; optimize in FP32

How does 8-bit Adam work?



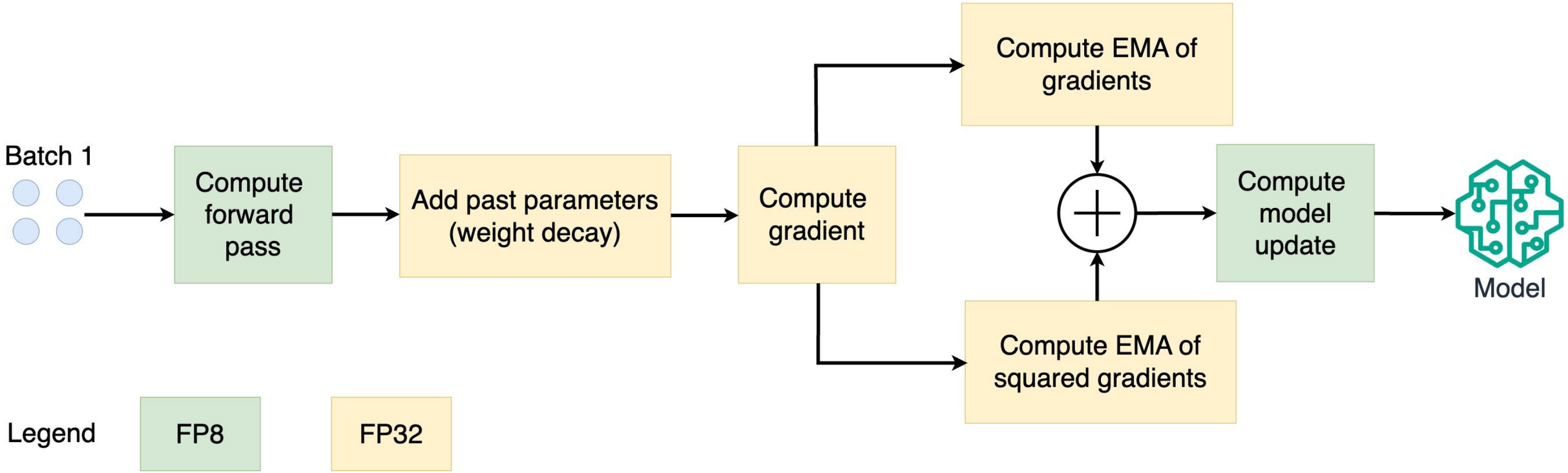
- Store parameters in FP8; optimize in FP32

How does 8-bit Adam work?



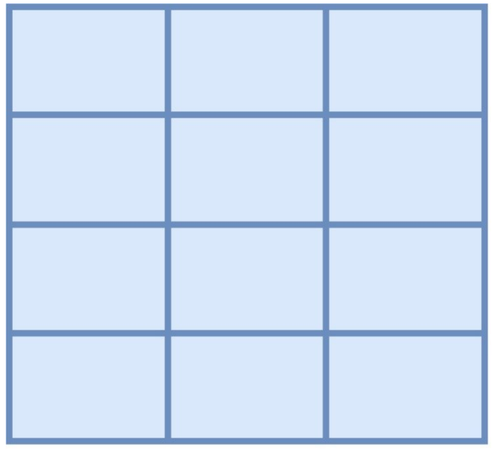
- Store parameters in FP8; optimize in FP32
- EMA: exponential moving average
- Compute the EMA of the gradients and squared gradients

How does 8-bit Adam work?

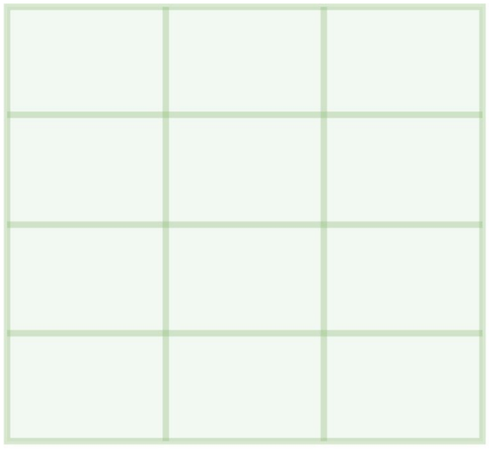


- Store parameters in FP8; optimize in FP32
- EMA: exponential moving average
- Compute the EMA of the gradients and squared gradients

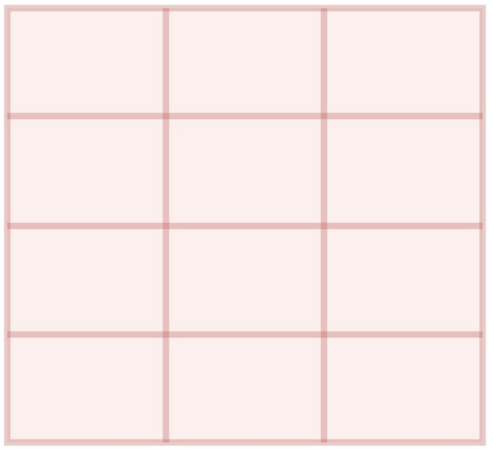
How does 8-bit Adam save memory?



Parameter gradients

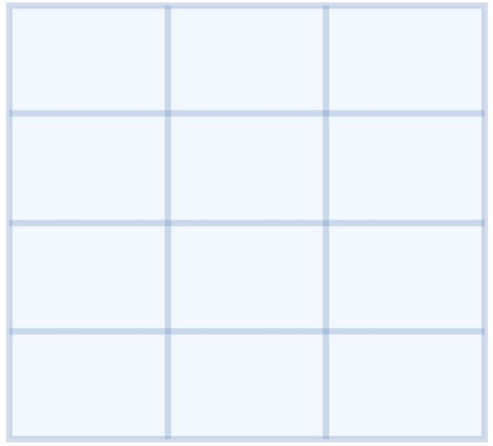


EMA of gradients

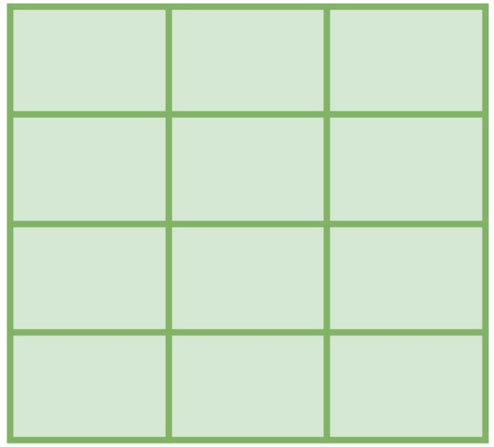


EMA of squared gradents

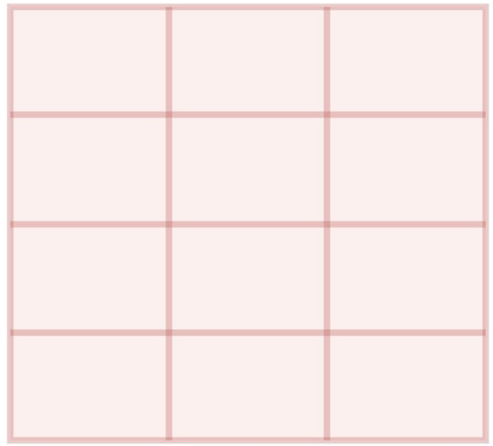
How does 8-bit Adam save memory?



Parameter gradients

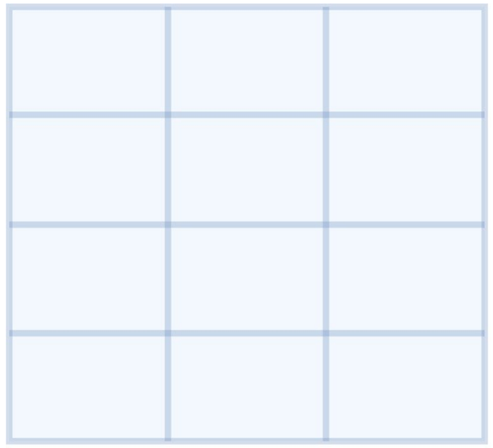


EMA of gradients

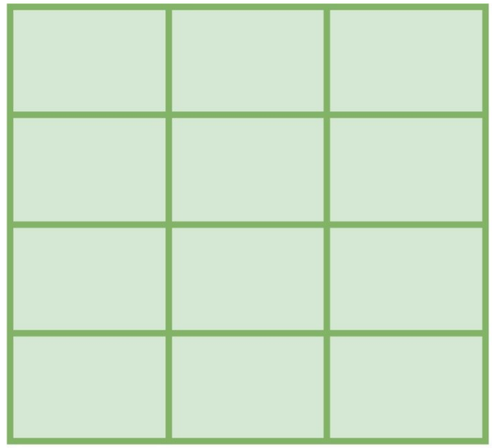


EMA of squared gradents

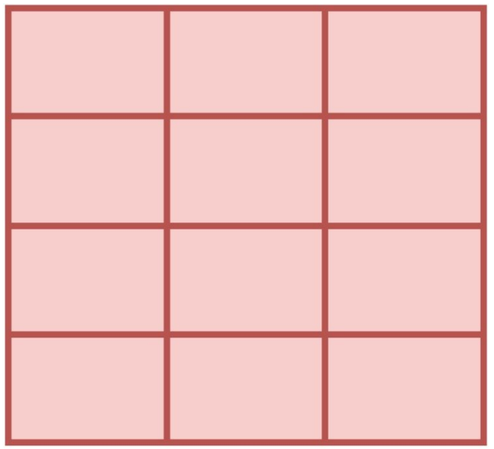
How does 8-bit Adam save memory?



Parameter gradients



EMA of gradients



EMA of squared gradents

- Each square is a parameter, and each color is a state
- Memory per parameter = 2 bytes = 1 byte per state * 2 states
- Total memory = Memory per parameter (2 bytes) * Number of parameters

Estimate memory usage of 8-bit Adam

```
model = AutoModelForSequenceClassification.from_pretrained(
    "distilbert-base-cased", return_dict=True)
num_parameters = sum(p.numel() for p in model.parameters())
print(f"Number of model parameters: {num_parameters:,}")
```

```
Number of model parameters: 65,783,042
```

```
estimated_memory = num_parameters * 2 / (1024 ** 2)
print(f"Estimated memory usage of 8-bit Adam: {estimated_memory:.0f} MB")
```

```
Estimated memory usage of 8-bit Adam: 125 MB
```

Set up the 8-bit Adam optimizer

```
import bitsandbytes as bnb
from transformers.trainer_pt_utils import get_parameter_names

args = TrainingArguments(output_dir="./results")
decay_parameters = get_parameter_names(model, [nn.LayerNorm])
decay_parameters = [name for name in decay_parameters if "bias" not in name]
```

- Weight decay prevents overfitting
- `decay_parameters` : specify parameters for applying weight decay
- `get_parameter_names` : return parameter names; ignore `nn.LayerNorm` layers
- Remove `bias` parameters from `decay_parameters`
- Don't apply weight decay to normalization layers and biases

Set up the 8-bit Adam optimizer

```
optimizer_grouped_parameters = [{"params": [p for n, p in model.named_parameters()
                                            if n in decay_parameters],
                                 "weight_decay": args.weight_decay},
                                {"params": [p for n, p in model.named_parameters()
                                            if n not in decay_parameters],
                                 "weight_decay": 0.0}]
adam_bnb_optim = bnb.optim.Adam8bit(optimizer_grouped_parameters,
                                    betas=(args.adam_beta1, args.adam_beta2),
                                    eps=args.adam_epsilon,
                                    lr=args.learning_rate)
```

- `optimizer_grouped_parameters` : One group applies weight decay; the other does not
- `beta1` , `beta2` : Decay rates of 1st and 2nd moments; higher = stable, slow training

Implement 8-bit Adam with Trainer

```
trainer = Trainer(model=model,
                  args=training_args,
                  train_dataset=train_dataset,
                  eval_dataset=validation_dataset,
                  optimizers=(adam_bnb_optim, None),
                  compute_metrics=compute_metrics)
trainer.train()
```

```
{'epoch': 1.0, 'eval_loss': 0.63, 'eval_accuracy': 0.67, 'eval_f1': 0.62}
{'epoch': 2.0, 'eval_loss': 0.61, 'eval_accuracy': 0.71, 'eval_f1': 0.66}
```

Implement 8-bit Adam with Accelerator

```
model, adam_bnb_optim, train_dataloader, lr_scheduler = \
    accelerator.prepare(model, adam_bnb_optim, train_dataloader, lr_scheduler)

for batch in train_dataloader:
    inputs, targets = batch["input_ids"], batch["labels"]
    outputs = model(inputs, labels=targets)
    loss = outputs.loss
    accelerator.backward(loss)
    trainer.optimizer.step()
    lr_scheduler.step()
    adam_bnb_optim.zero_grad()
    print(f"Loss = {loss}")
```

```
Loss = 0.75
```

Compute memory usage of 8-bit Adam

```
total_size_megabytes, total_num_elements = \
    compute_optimizer_size(trainer.optimizer.state.values())
print(f"Number of 8-bit Adam parameters: {total_num_elements:,}")
print(f"8-bit Adam size: {total_size_megabytes:.0f} MB")
```

```
Number of 8-bit Adam parameters: 131,566,188
8-bit Adam size: 128 MB
```

- Compare with AdamW: 8-bit Adam uses 1/4 of the memory

```
Number of AdamW parameters: 131,566,188
AdamW size: 502 MB
```

Let's practice!

EFFICIENT AI MODEL TRAINING WITH PYTORCH

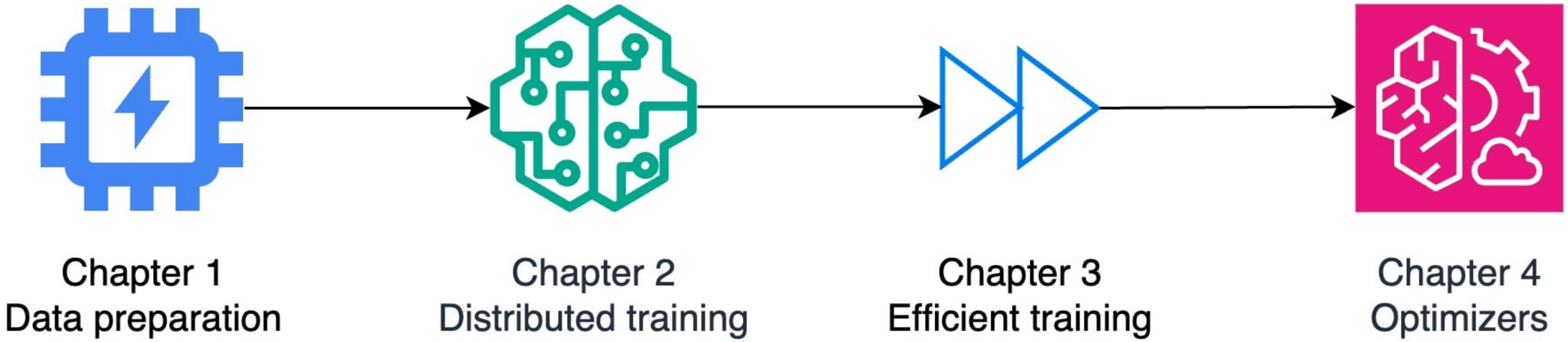
Congratulations!

EFFICIENT AI MODEL TRAINING WITH PYTORCH

Dennis Lee
Data Engineer

datacamp

Course journey



- Train models across multiple devices
- Ready to tackle large models with billions of parameters
- Challenges: hardware constraints, lengthy training times, memory limitations

Data preparation

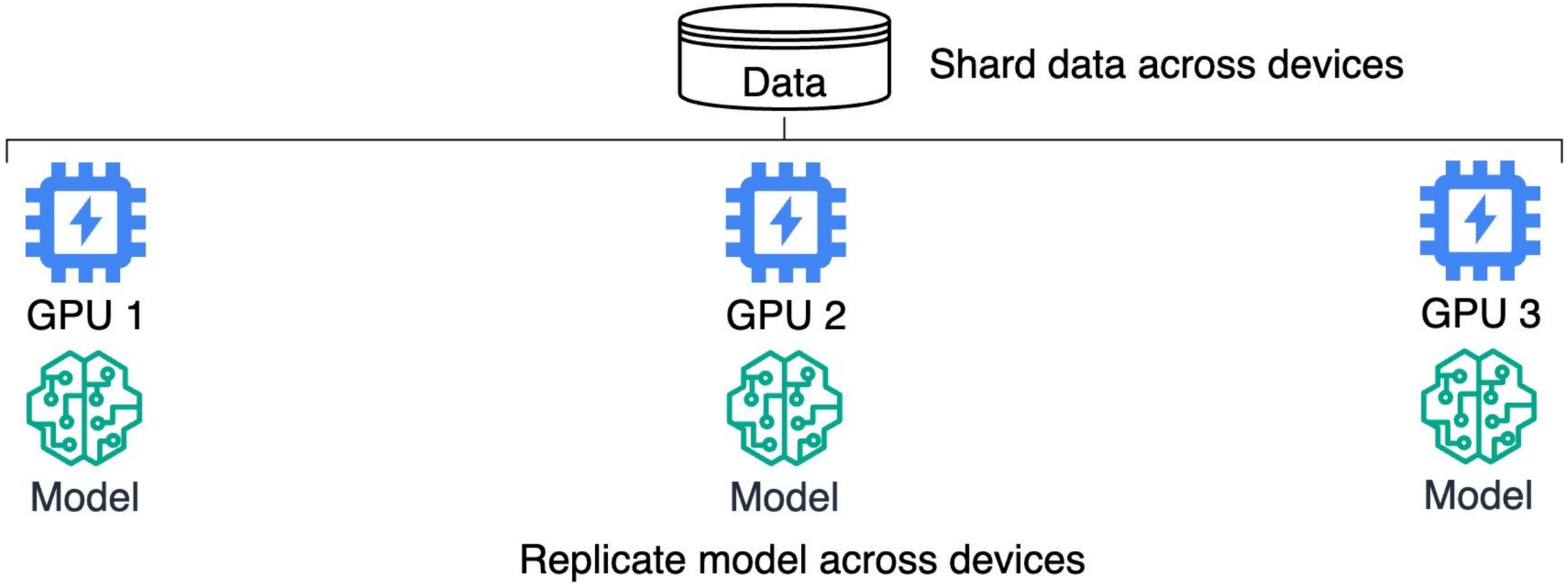
Chapter 1
Data preparation

Chapter 2
Distributed training

Chapter 3
Efficient training

Chapter 4
Optimizers

Distribute data and model across devices



Distributed training

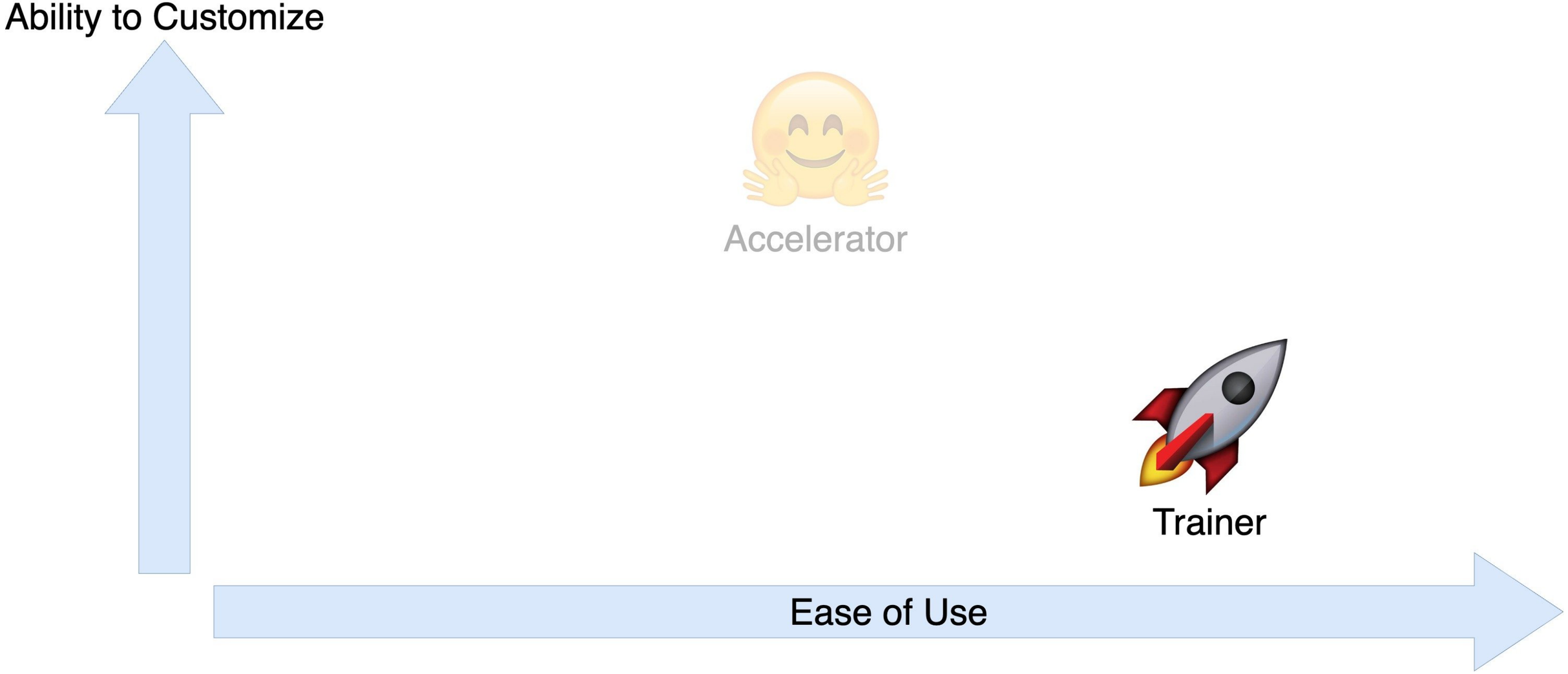
Chapter 1
Data preparation

Chapter 2
Distributed training

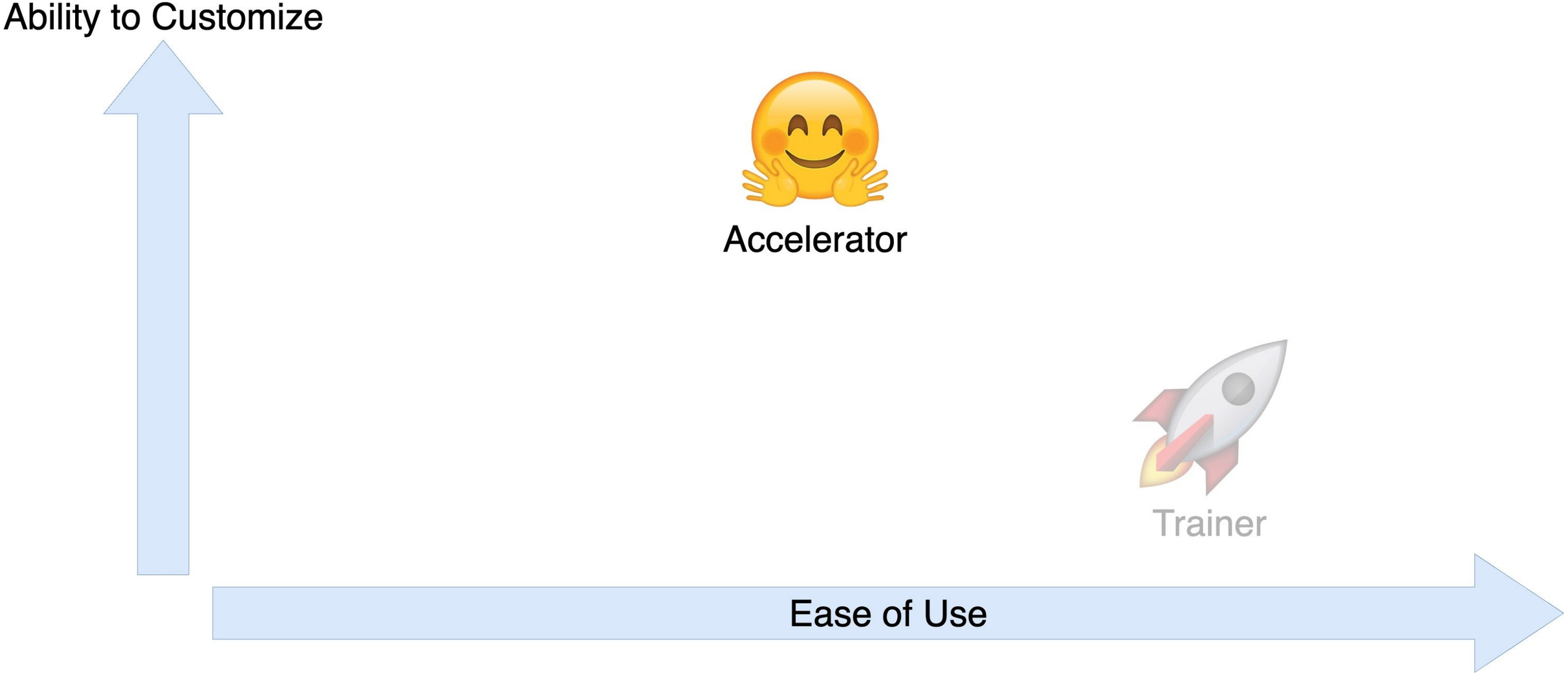
Chapter 3
Efficient training

Chapter 4
Optimizers

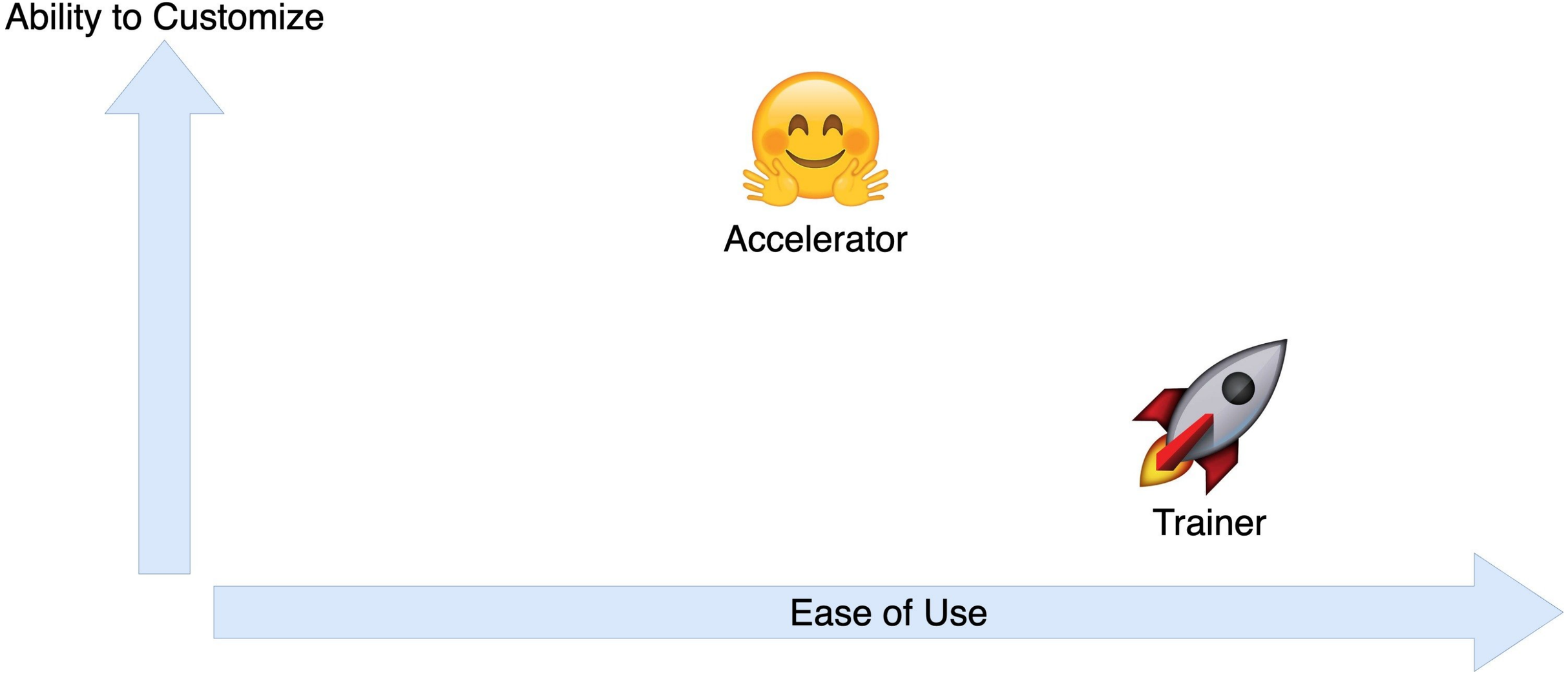
Trainer and Accelerator interfaces



Trainer and Accelerator interfaces



Trainer and Accelerator interfaces



Efficient training

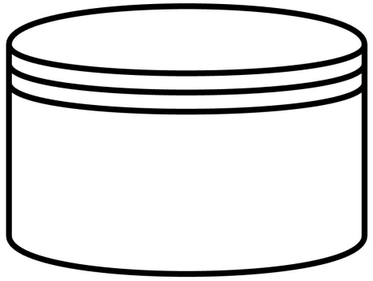
Chapter 1
Data preparation

Chapter 2
Distributed training

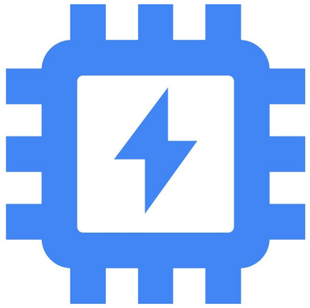
Chapter 3
Efficient training

Chapter 4
Optimizers

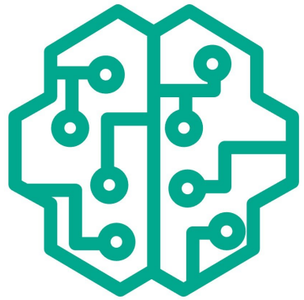
Drivers of efficiency



Memory Efficiency

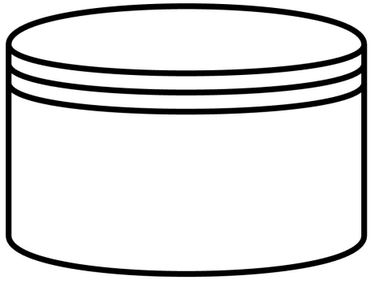


Communication Efficiency



Computational Efficiency

Drivers of efficiency



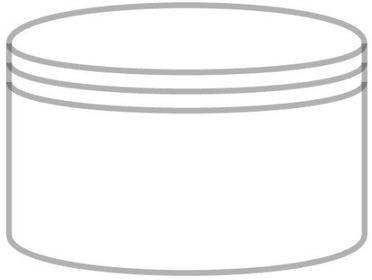
Memory Efficiency

Communication Efficiency

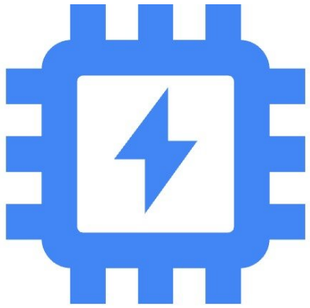
Computational Efficiency

- Memory efficiency
  - Gradient accumulation: train on larger batches
  - Gradient checkpointing: decrease model footprint

Drivers of efficiency



Memory Efficiency

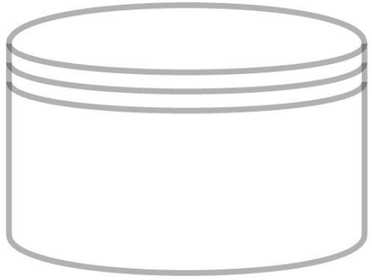


Communication Efficiency

Computational Efficiency

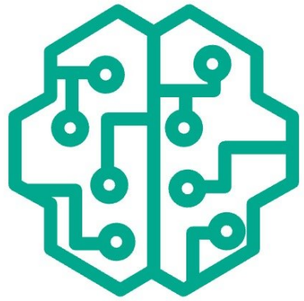
- Memory efficiency
  - Gradient accumulation: train on larger batches
  - Gradient checkpointing: decrease model footprint
- Communication efficiency: local SGD

Drivers of efficiency



Memory Efficiency

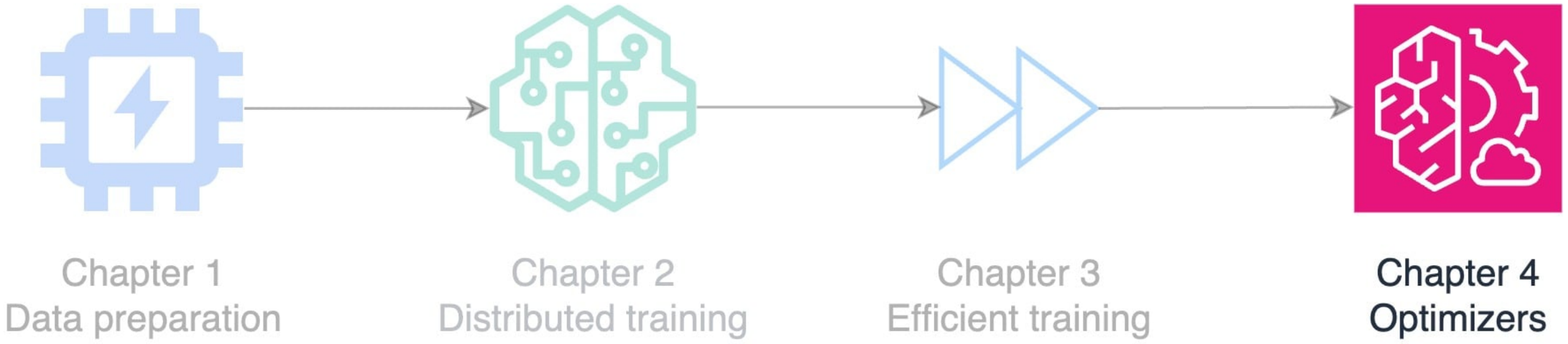
Communication Efficiency



Computational Efficiency

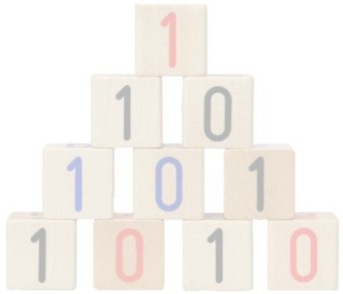
- Memory efficiency
  - Gradient accumulation: train on larger batches
  - Gradient checkpointing: decrease model footprint
- Communication efficiency: local SGD
- Computational efficiency: mixed precision training

Optimizers


Chapter 1
Data preparation
Chapter 2
Distributed training
Chapter 3
Efficient training
Chapter 4
Optimizers

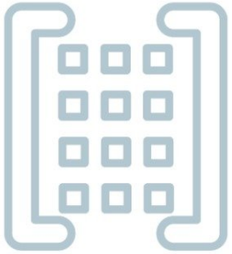
Optimizer tradeoffs

More parameters



8-bit Adam

AdamW

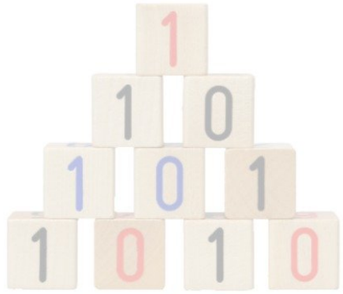


Adafactor

Higher precision

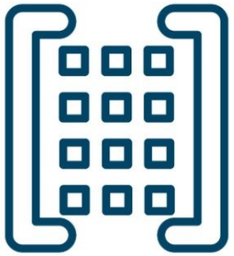
Optimizer tradeoffs

More parameters



8-bit Adam

AdamW

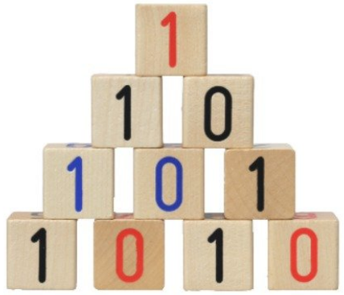


Adafactor

Higher precision

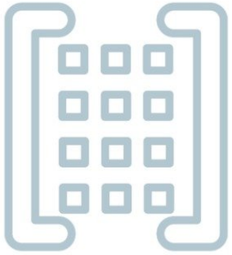
Optimizer tradeoffs

More parameters



8-bit Adam

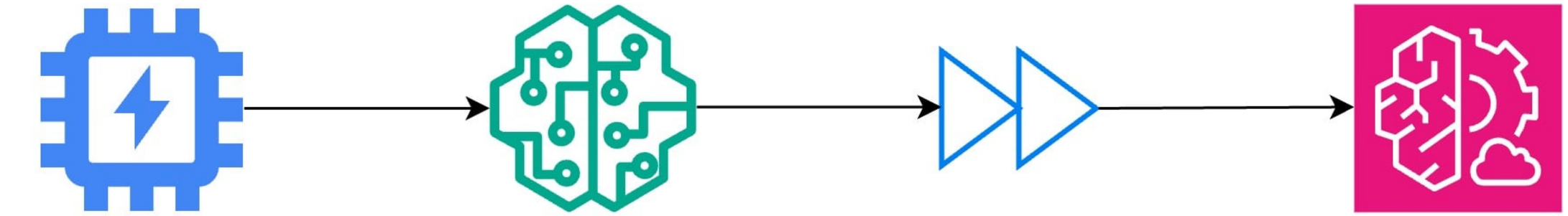
AdamW



Adafactor

Higher precision

Equipped to excel in distributed training



Chapter 1
Data preparation

Chapter 2
Distributed training

Chapter 3
Efficient training

Chapter 4
Optimizers

Kudos!

EFFICIENT AI MODEL TRAINING WITH PYTORCH