Introduction to PyTorch Lightning

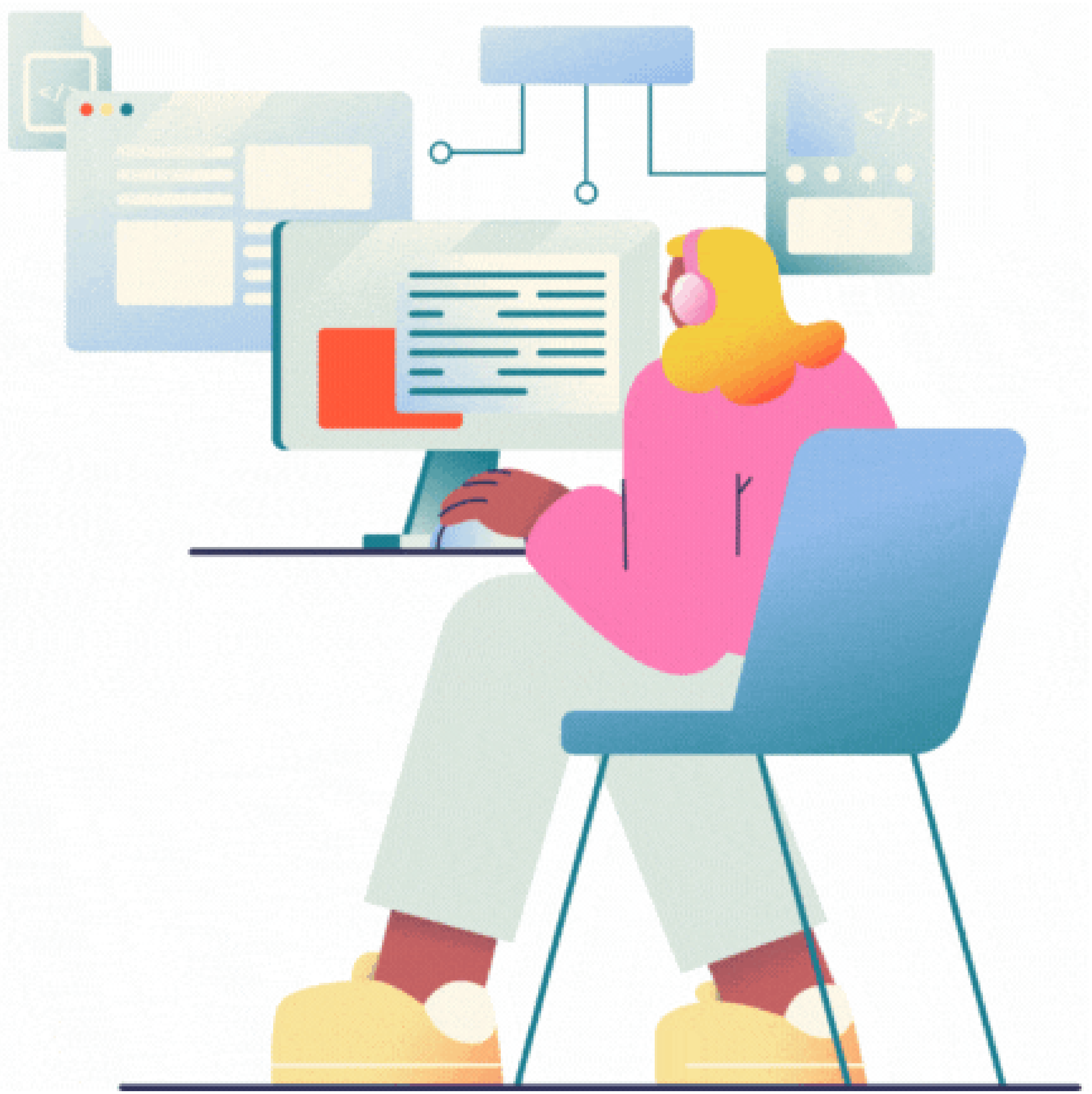
SCALABLE AI MODELS WITH PYTORCH LIGHTNING

Sergiy Tkachuk
Director, GenAI Productivity

PyTorch & PyTorch Lightning

Standard PyTorch:

- Significant manual effort
- Writing explicit training loops
- GPU/TPU handling, logging, and checkpointing



PyTorch & PyTorch Lightning

PyTorch Lightning:

- Built on top of PyTorch
- Automates:
  - Training
  - Checkpointing
  - Logging
- Reduces boilerplate
- Improves scalability and reproducibility

Overview of PyTorch Lightning

- Example: global e-commerce streamlining workflows
  - Visual search model development
  - Automated training loops
  - Rapid iteration with minimal boilerplate

- Core components: `LightningModule` and `Trainer`

```
from lightning.pytorch import LightningModule
from lightning.pytorch import Trainer
```

Lightning structure

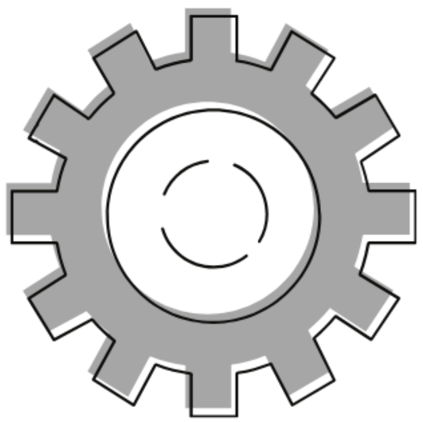
Key components:

- LightningModule: core model logic

Lightning structure

Key components:

- LightningModule: core model logic
- Lightning Trainer: orchestrates training



Lightning structure

Key components:

- LightningModule: core model logic
- Lightning Trainer: orchestrates training
- DataModule: organizes data pipelines
- Callbacks: automates events
- Logger: tracks experiments

LightningModule in action

Key points:

- `__init__` : Defines model architecture
- `forward()` : Pass data through the model
- `training_step()` : Define training
- Custom hooks available

```
import lightning.pytorch as pl

class LightClassifier(pl.LightningModule):
    def __init__(self, model, criterion, optimizer):
        super().__init__()
        self.model = model
        self.criterion = criterion
        self.optimizer = optimizer
    def forward(self, x):
        return self.model(x)
    def training_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = self.criterion(logits, y)
        return loss
```

Lightning Trainer in action

Key points:

- Manages training loop
- Supports distributed training
- Handles callbacks & logging
- Optimizes resource usage

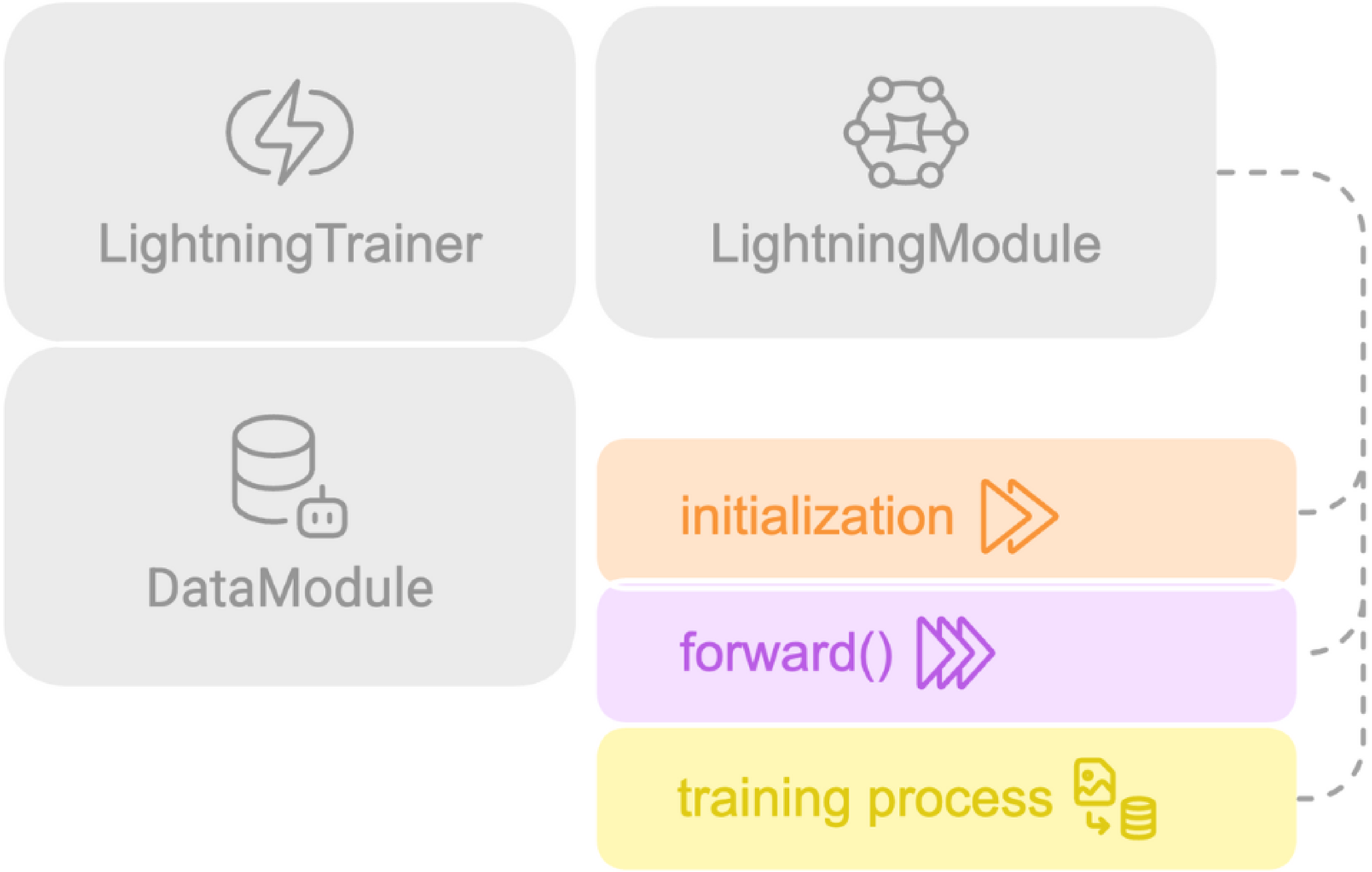
```
model = LightClassifier()



trainer = Trainer(max_epochs=10, accelerator="gpu", devices=1)
trainer.fit(model, train_dataloader, val_dataloader)
```

Introducing the Afro-MNIST dataset

A set of synthetic MNIST-style datasets for four orthographies used in Afro-Asiatic and Niger-Congo languages: Ge'ez (Ethiopic), Vai, Osmanya, and N'Ko.

[1] Wu, Daniel J., Andrew C. Yang, and Vinay U. Prabhu. "Afro-MNIST: Synthetic generation of MNIST-style datasets for low-resource languages." arXiv preprint arXiv:2009.13509 (2020).

PyTorch Lightning recap



Let's practice!

SCALABLE AI MODELS WITH PYTORCH LIGHTNING

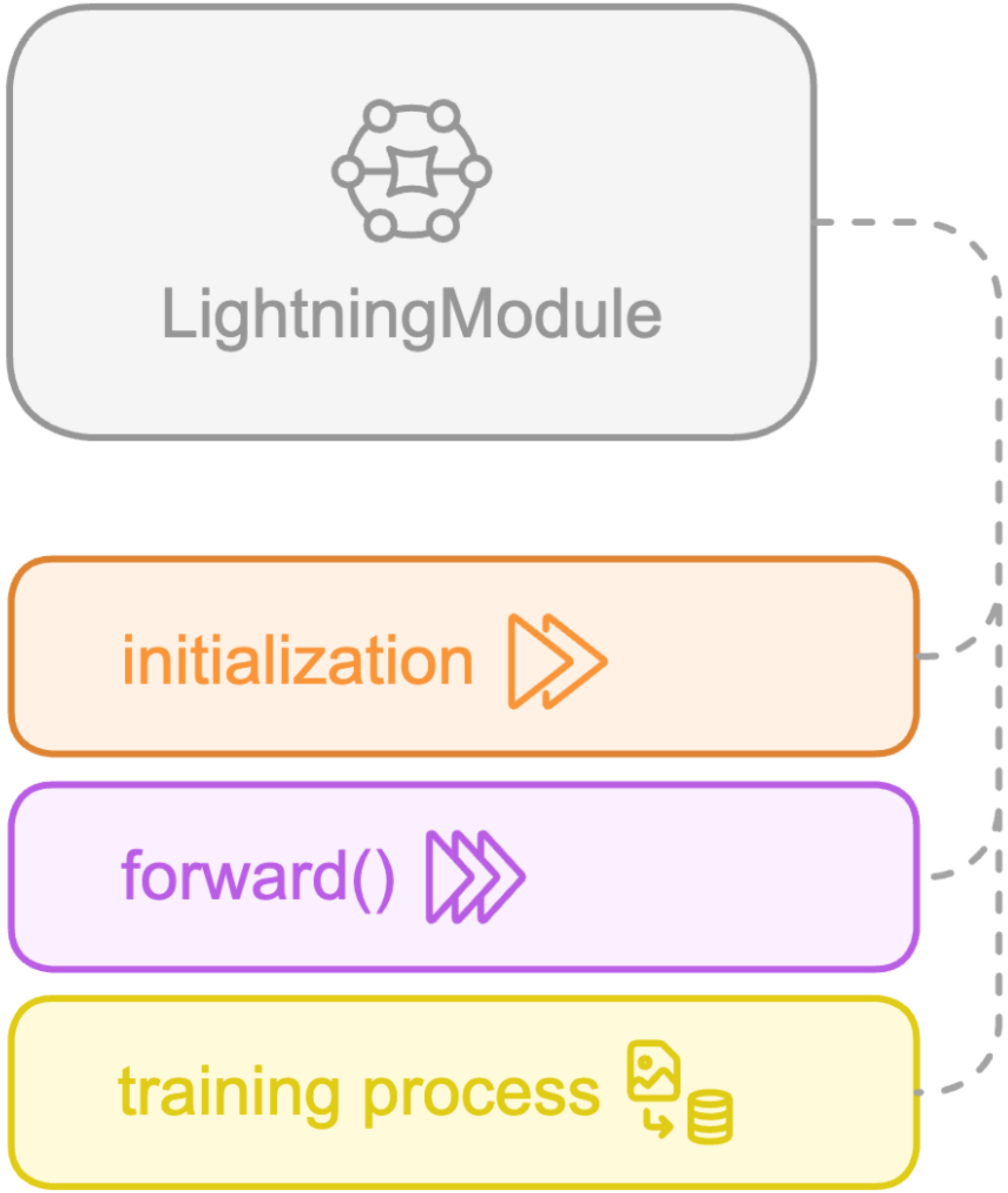
Defining models with LightningModule

SCALABLE AI MODELS WITH PYTORCH LIGHTNING

Sergiy Tkachuk
Director, GenAI Productivity

LightningModule in focus

1. Encapsulates your model architecture
2. Organizes training logic into a single, manageable unit
3. Blueprint that brings order and clarity to deep learning projects



Defining the init method

Key tasks:

- Model's initialization
- `super()` :
  - Automated handling of training loops
  - Logging
  - Checkpointing
- Model layers defined after initialization
- Modular and easy to maintain

```python
import lightning.pytorch as pl
import torch.nn as nn

class ClassificationModel(pl.LightningModule):
    def __init__(self, input_dim,
                 hidden_dim, num_class):
          # Initialize parent class
        super().__init__()
        # First layer
        self.layer1 = nn.Linear(input_dim,
                                hidden_dim)
        # Activation function
        self.relu = nn.ReLU()
        # Output layer
        self.layer2 = nn.Linear(hidden_dim,
                                num_class)
```

Implementing the forward method

Key steps:

- Define data flow through network
- Process input through layers sequentially
  - Linear transformation
  - Activation
  - Last layer and output

```
import lightning.pytorch as pl
import torch.nn as nn

class ClassificationModel(pl.LightningModule):
    def __init__(self, input_dim,
                 hidden_dim, num_class):
        ...
    def forward(self, x):
        x = self.layer1(x) # Pass input
        x = nn.ReLU(x) # Apply activation
        x = self.layer2(x) # Compute output
        return x # Return result
```

Example: classifying hand written digits

```
import lightning.pytorch as pl
from torch.utils.data import DataLoader
from torchvision.datasets import MNIST
from torchvision import transforms
transform = transforms.ToTensor()
train_ds = MNIST(root='.', train=True,  download=True, transform=transform)
test_ds  = MNIST(root='.', train=False, download=True, transform=transform)

train_loader = DataLoader(train_ds, batch_size=64, shuffle=True)
test_loader  = DataLoader(test_ds,  batch_size=64)

model = ClassificationModel(input_dim=28*28, hidden_dim=128, num_class=10)

trainer = pl.Trainer(max_epochs=3, accelerator='auto')
trainer.fit(model, train_loader, test_loader)
```

Integrating the model with classification tasks

- Focus on classification use case
- Entire flow within LightningModule
- Output raw outputs for softmax activation
- Integration with Lightning Trainer

```
class ClassificationModel(pl.LightningModule):
  def __init__(self, input_dim,
               hidden_dim, output_dim):
    super().__init__()
    self.hid = nn.Linear(input_dim, hidden_dim)
    self.out = nn.Linear(hidden_dim, output_dim)

  def forward(self, x):
    x = self.hidden(x)
    x = nn.ReLU(x)
    x = self.output(x)
    return x
```

Let's practice!

SCALABLE AI MODELS WITH PYTORCH LIGHTNING

Implementing training logic

SCALABLE AI MODELS WITH PYTORCH LIGHTNING

Sergiy Tkachuk
Director, GenAI Productivity

Defining the training step

- Process input and label batch
- Compute predictions with forward pass
- Calculate cross entropy loss for classification
- Log training loss for monitoring

```
def training_step(self, batch, batch_idx):
    x, y = batch
    y_hat = self(x)
    loss = cross_entropy(y_hat, y)
    self.log("train_loss", loss)
    return loss
```

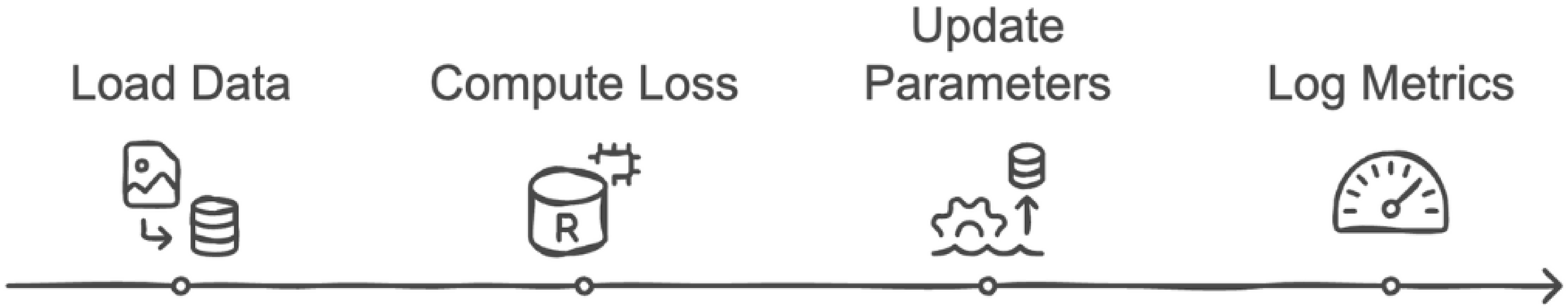
Configuring optimizers

- Select an appropriate optimizer for updates
- Link model parameters for gradient computation
- Set a suitable learning rate for convergence
- Return the optimizer instance for Lightning integration

```
def configure_optimizers(self):
    optimizer = torch.optim.Adam(self.parameters(), lr=1e-3)
    return optimizer
```

Training with Lightning Trainer

- Integrate training logic with Lightning Trainer
- Manage training loops and epochs automatically
- Monitor performance metrics in real time



Using trainer.fit and trainer.validate

- Start training with trainer.fit method
- Validate model with trainer.validate method

```
trainer.fit(model, train_dataloader)
trainer.validate(model, val_dataloader)
```

- Automate training and validation cycles
- Monitor metrics during both phases

Complete training logic example

- Define a custom LightningModule with a classifier
- Implement training_step to compute and log loss
- Configure optimizers to update model parameters
- Train and validate the model

```python
class LightClassifier(pl.LightningModule):
    def __init__(self):
        super().__init__()
        self.layer=torch.nn.Linear(28 * 28, 10)
    def forward(self, x):
        return self.layer(x.view(x.size(0), -1))
    def training_step(self, batch, batch_idx):
        ...
    def configure_optimizers(self):
        params=self.parameters()
        optimizer=torch.optim.Adam(params,lr=1e-3)
        return optimizer
model = LightClassifier() # Define classifier model
trainer = Trainer(max_epochs=5) # Define trainer
trainer.fit(model, train_dataloader)
trainer.validate(model, val_dataloader)
```

Industry applications

Why training logic matters?

- Ensure precise loss tracking for quality control
- Optimize training pipelines for scalable deployment

Real-world examples:

- Enhance image analysis in healthcare diagnostics
- Support fraud detection in financial services

Industry applications

Why training logic matters?

- Ensure precise loss tracking for quality control
- Optimize training pipelines for scalable deployment

Real-world examples:

- Enhance image analysis in healthcare diagnostics
- Support fraud detection in financial services

Let's practice!