Category	Sub-category	Code Smell	Happens in other frmework?	Framework problem?	Paper Name	Description	Consequences	Name Best Practice	Best Parctice (from the paper)	Bad Pratice	Good Practice

Resource Management Concerns This category refers to issues that arise from inefficient use or release of memory and computational resources.	**Tensor Inefficient Operations** This subcategory includes patterns where tensor operations are used inefficiently, such as excessive concatenations, repeated cloning, or unnecessary creation of temporary tensors. These operations can increase memory usage, create redundant computational graphs, and degrade runtime performance.	Tensor Over-Concatenation			Overuse or frequent concatenation of tensors along a specified dimension	This smell refers to the repeated concatenation of tensors during a loop or over time without preallocating memory. Each concatenation operation creates a new tensor and copies existing data.	- Out Of Memory (OOM) Errors
							- Slowdowns in training or inference
								Use In-Place Tensor Operations	- Employing In-Place Operations or Functions to
									Minimize Tensor Creation. Examples: torch.add_(), torch.mul_(), or tensor. resize_(), where the underscore (_) signifies that the operation is performed directly on the tensor, modifying it in place.
										`results = torch.empty(0, 10)`
										
										`for _ in range(100):`
										`    new_tensor = torch.rand(1, 10)`
										`    results = torch.cat((results, new_tensor), dim=0)  # BAD:`
										`new tensor every time`
											`results = torch.empty(100, 10)  # GOOD: preallocate`
											`memory`
											
											`for i in range(100):`
											`    new_tensor = torch.rand(1, 10)`
											`    results[i] = new_tensor`

		Unnecessary Dim Retention	**T (similar to SML)**		Unnecessary retention of dimensions	This smell refers to keeping redundant size-1 dimensions after reduction operations, such as averaging or summing over an axis. These unused dimensions increase tensor shape unnecessarily.	- Out Of Memory (OOM) Errors
							- Larger-than-expected tensor shapes
							- Slower performance during operations like matrix multiplication or broadcasting due to larger tensor shapes
								Dimension Squeezing	- Squeezing Unnecessary Dimensions	`x = torch.randn(32, 1, 10)`
										`mean = x.mean(dim=1, keepdim=True)  # BAD: retains [32, 1, 10]`
										`unnecessarily`
											`x = torch.randn(32, 1, 10)`
											`mean = x.mean(dim=1)  # GOOD: result shape is [32, 10]`

		Inefficient GPU Matrix Ops		Possibly Framework	Performing matrix multiplication on two 2D tensors directly on the GPU without proper memory management.	This smell refers to performing large matrix multiplications directly on the GPU without considering memory limits, tensor shapes, or hardware efficiency, leading to excessive memory usage and potential performance issues.	- Excessive memory consumption
							- CUDA memory leaks
								Offload Non-Critical Operations to CPU	- Moving Operations to CPU for Efficient GPU Resource Utilization	`a = torch.randn(20000, 20000, device='cuda')`
										`b = torch.randn(20000, 20000, device='cuda')`
										
										`result = torch.matmul(a, b)  # BAD: easily causes OOM on GPU`
											`a = torch.randn(20000, 20000).to('cpu')  # GOOD: stays`
											`on CPU`
											`b = torch.randn(20000, 20000).to('cpu')`
											
											`result = torch.matmul(a, b)  # No GPU memory consumed`

	Tensor Storage Mismanagement This subcategory refers to storing tensors in long-lived containers, such as class attributes or global variables, which unnecessarily extends their lifecycle.	Improper Tensor Retention			Directly saving tensors in a data structure without uti-
					lizing proper context management
						This smell refers to storing tensors that require gradients in long-lived data structures, such as lists, dictionaries, or global variables, without proper context management. When these tensors are saved without detachment or context management, their associated computation graphs are retained unnecessarily.	- Excessive memory consumption
							- Potentially causing out-of-memory errors.
								Store Intermediate Tensors Explicitly During Backward Pass	- Proper Storage of Intermediate Results with
									save_for_backward, to prevent unintended tensor retention, reduce memory overhead, and enhance the overall stability and efficiency of deep learning models.
										`outputs = []`
										
										`for batch in dataloader:`
										`    x = batch.to(device)`
										`    out = model(x)  # has grad_fn`
										`    outputs.append(out)  # BAD: still attached to computation`
										`graph`
										
										`loss = compute_loss(outputs)`
										`loss.backward()`
											`outputs = []`
											
											`for batch in dataloader:`
											`    x = batch.to(device)`
											`    out = model(x)`
											`    outputs.append(out.detach().cpu())  # GOOD:`
											`detaches from graph and moves to CPU`

		Forward-Pass Tensor Stored as Class Attribute			Using a class attribute to hold a tensor reference within the forward pass	This smell refers to storing intermediate tensors as class attributes during the forward pass. Doing so retains references to these tensors beyond their useful scope, preventing memory from being released.	- Unexpectedly High GPU Memory Usage
							- Out Of Memory (OOM) Errors
								Local Variable Usage	- Using Local Variables for Temporary Tensors to Prevent Memory Leaks	`import torch`
										`import torch.nn as nn`
										
										`class LeakyModel(nn.Module):`
										`    def __init__(self):`
										`        super().__init__()`
										`        self.linear = nn.Linear(10, 5)`
										
										`    def forward(self, x):`
											`import torch`
											`import torch.nn as nn`
											
											`class CleanModel(nn.Module):`
											`    def __init__(self):`
											`        super().__init__()`
											`        self.linear = nn.Linear(10, 5)`
											
											`    def forward(self, x):`

	Tensor/Object Retention This subcategory refers to the unintended accumulation of tensors or model-related objects. Common causes include appending non-detached tensors to lists or storing outputs in persistent logs, which prevents memory from being released.	Lingering References	**T (UTR)**		Lingering tensor references	This smell refers to when unused PyTorch tensors remain in memory due to persistent references. This prevents garbage collection, leading to memory buildup, especially in cases like DQN replay memory, where improper management can cause excessive memory consumption.	- Excessive memory consumption	Use NumPy to Prevent Tensor Retention	• Convert tensors to NumPy arrays before storing them in replay memory. This prevents tensors from maintaining unintentioanl strong references in the memory.
									• For GPU tensors, offload them to the CPU using .cpu() prior to storage, ensuring that GPU memory is freed for future use.
									• Periodically calling torch.cuda. empty_cache()
									helps to clear unused memory, further preventing resource leaks.
									• Using local variables instead of class attributes to hold temporary tensors or intermediate
										`replay_buffer = []`
										
										`for data in dataloader:`
										`    output = model(data)  # Tensor on GPU`
										`    replay_buffer.append(output)  # BAD: stores GPU tensor`
										`directly`
											`replay_buffer = []`
											
											`for data in dataloader:`
											`    with torch.no_grad():`
											`        output = model(data).detach().cpu().numpy()  #`
											`GOOD: no grad, off GPU`
											`    replay_buffer.append(output)`

		Unreleased Tensor/Model References	**T (UTR)**		Improper handling of resources such as tensors and models	This smell refers to retaining tensors or model outputs that are still attached to the computation graph, often by storing them without detachment. When these references are kept outside the forward pass, the underlying graph and gradient data persist in memory.	- Out-of-memory (OOM) errors
							- Rapidly increasing GPU memory usage over time
							- Build up of computational graphs
								Proper Tensor Detachment	- Proper Detachment of Tensors and Hidden Layers to Prevent Computational Graph Retention (e.g., loss.detach())	`losses = []`
										
										`for data, target in dataloader:`
										`    output = model(data)`
										`    loss = loss_fn(output, target)`
										`    losses.append(loss)  # BAD: retains entire graph`
											`losses = []`
											
											`for data, target in dataloader:`
											`    output = model(data)`
											`    loss = loss_fn(output, target)`
											`    losses.append(loss.detach())  # GOOD: frees graph`
											`memory`

		Unreleased Hook Memory		Possibly Framework	Failure to release GPU memory after creating instances of a module with a registered forward hook	This smell refers to the use of forward hooks without properly releasing the memory they consume. When hooks are registered but not removed, they retain references to inputs or outputs across iterations, leading to memory accumulation.	- Excessive memory consumption
							- Out-of-memory (OOM) errors
							- Potentially lead to the complete exhaustion of GPU memory
							- Compromising the stability and efficiency of both model training and inference processes
								Unregister Hooks and Avoid Self-References	- Proper un-registration of Forward Hooks
									- Optionally, minimizing the use of self within hooks prevents the creation of unintended references to the model
										`# Bad: Hook registered but never removed`
										`model = nn.Linear(10, 5).cuda()`
										`model.register_forward_hook(lambda m, i, o: print("hook"))`
										`for _ in range(100):`
										`    model(torch.randn(32, 10).cuda())`
											`# Good: Hook removed after use`
											`hook = model.register_forward_hook(lambda m, i, o:`
											`print("hook"))`
											`for _ in range(100):`
											`    model(torch.randn(32, 10).cuda())`
											`hook.remove()`

		Accumulated Object References	**K (IMR); T/K UR**		Improperly handling accumulated references	This smell refers to unintentionally keeping tensors or objects in memory across iterations by storing them in ways that prevent proper release after use.	- Out Of Memory (OOM) Errors
							- Memory is not released after an epoch or training loop finishes
								Static Methods to Avoid Object Ac	- Using Static Methods for Forward and Backward Computations to Prevent Object Accumulation	`saved = []`
										
										`class CustomFn(torch.autograd.Function):`
										`    @staticmethod`
										`    def forward(ctx, x):`
										`        ctx.save_for_backward(x)`
										`        saved.append(x)  # BAD: accumulating tensor references`
										`        return x * 2`
										
										`    @staticmethod`	

		Circular Buffer References		Possibly Framework	Creating circular references between buffers and objects	This smell refers to situations where two components, such as a model object and one of its internal buffers, hold direct references to each other, forming a circular dependency.	- Out Of Memory (OOM) Errors
							- Increased GPU/CPU memory consumption
							- Objects not getting garbage collected
							- Memory not released after training or inference ends
								Breaking Reference Cycles	- Breaking Circular References with weakref.ref for Proper Memory Deallocation	`import torch`
										`import torch.nn as nn`
										
										`class LeakyModule(nn.Module):`
										`    def __init__(self):`
										`        super().__init__()`
										`        buffer = torch.zeros(1000, 1000).to("cuda")  # Large`
										`tensor on GPU`
											`import torch`
											`import torch.nn as nn`
											`import weakref`
											
											`class SafeModule(nn.Module):`
											`    def __init__(self):`
											`        super().__init__()`
											`        buffer = torch.zeros(1000, 1000).to("cuda")  #`

Improper Resource and Cache Cleanup This subcategory refers to the failure to release temporary resources, such as caches, buffers, or forward hooks, after they are no longer needed. This leads to gradually increasing memory usage and resource exhaustion during training or inference.	Unreleased GPU Memory	**T(GRMF)**	Possibly Framework	Running the model without properly releasing GPU memory can lead to inefficient memory usage and potential leaks	This smell refers to running a model without explicitly managing GPU memory, failing to release unused tensors, or caches or forward hooks which leads to inefficient memory utilization and can cause memory leaks or \texttt{CUDA out of memory} errors.	- Excessive memory consumption
						- Out-of-memory (OOM) errors
						- Undermining the stability and efficiency of the model's operations, particularly when processing large datasets or performing extended inference tasks.
						- System slowdowns and crashes
							Explicit GPU Memory Release	- Call torch.cuda.empty_cache() to periodically to release unused GPU memory
								- Disabling gradient tracking during inference with \texttt{torch. no_grad()}
									for data in dataloader:
									    output = model(data) # BAD: output not deleted or detached
									    # No cleanup after usage
										for data in dataloader:
										    with torch.no_grad(): # GOOD: avoids building computation graph
										        output = model(data)
										
										    del output # Free the tensor
										    torch.cuda.empty_cache() # Optional: release unused GPU memory

	Tracing Inside Loop Without Cleanup	**K (IMR)**	possibly, torch.jit.trace() has a memory leak.	Repeated tracing in a loop without freeing resources or managing traced models appropriately	This smell refers to tracing models repeatedly in a loop without releasing earlier traces causes memory buildup in RAM and GPU, leading to inefficient resource use and performance slowdowns.	- Gradual GPU or CPU Memory Bloat
						- Out-Of-Memory (OOM) Errors
							Subprocess-Based Isolation	- Leveraging Subprocesses for Better Resource Handling	import torch
									import torch.nn as nn
									
									model = nn.Linear(10, 5).cuda()
									example_input = torch.randn(1, 10).cuda()
									
									# BAD PRACTICE: Tracing inside a loop without cleanup
									for _ in range(1000):
										import torch
										import torch.nn as nn
										
										model = nn.Linear(10, 5).cuda()
										example_input = torch.randn(1, 10).cuda()
										
										# GOOD PRACTICE: Trace once outside the loop
										traced = torch.jit.trace(model, example_input)

	Unreleased Shell References			Using the Python shell to create variables	This smell refers to when in long-running IPython sessions, undeleted variables can accumulate and cause memory issues. Using %xdel helps fully remove variables and their references, preventing memory leaks and improving resource efficiency.	- Degraded system performance	Interactive Resource Cleanup	- Using %xdel to Free Resources	import numpy as np
									
									# Create a large array
									X = np.random.random((10000, 10000))
									
									# Display the array
									X
									# In IPython, when you display a variable without explicitly printing it, the output is stored in the Out cache. This means
										import numpy as np
										
										# Create a large array
										X = np.random.random((10000, 10000))
										
										# Proper cleanup in IPython
										%xdel X # GOOD: Removes variable and clears references from the Out cache

	Using del Without Freeing Memory			Assuming that calling del on variables is enough to free memory	Simply using del in PyTorch doesn't guarantee memory is freed, as the computation graph may still hold references. Without clearing these, memory can accumulate, leading to performance issues or out-of-memory errors.	- Performance degradation - Out-Of-Memory (OOM) Errors	Proper Memory Release	- Ensure proper memory release	import torch x = torch.randn(10000, 10000, requires_grad=True).cuda() y = x * 2 z = y.mean() del y # BAD: Memory still held due to computation graph linking x → y → z	import torch x = torch.randn(10000, 10000, requires_grad=True).cuda() y = x * 2 z = y.mean() # GOOD PRACTICE: Detach or stop gradient tracking before delete if memory isn't needed

	Unmanaged Memory Cache			Maintaining memory allocated as a form of cache without properly releasing it when it's no longer needed.	This smell refers to when intermediate results, model outputs, or activations are cached (intentionally or by the framework) to speed up repeated computations or access. However, if these cached tensors are not explicitly deleted, detached, or cleared, they remain in memory, even if they are no longer used,contributing to memory issues.	- Gradual GPU Memory Growth
						- Out Of Memory (OOM) Errors
							Manual Cache Release	- Manually release GPU memory cache using torch.cuda.empty_cache() when necessary	cached_outputs = []
									for data in dataloader:
									    output = model(data) # BAD: accumulates in list
									    cached_outputs.append(output) # Never cleared or detached
										cached_outputs = []
										for data in dataloader:
										    output = model(data).detach().cpu() # GOOD: remove graph & offload from GPU
										    cached_outputs.append(output)
										
										# Optional: manually free GPU cache if needed
										torch.cuda.empty_cache()

Debug Artifacts This subcategory refers to leftover debugging constructs, such as print statements, assertion checks, and verbose logging, that remain in training or production code. These artifacts introduce unnecessary overhead and clutter the codebase.	Dead Code			Leaving debugging-related code in production or training code.	This smell refers to leaving debugging-specific constructs in the training or production code after the debugging phase has ended. Examples include verbose logging, assertion checks, diagnostic tools, or test-only control structures.	- Excessive memory consumption
						- Slowed down execution
							Removing Debug Artifacts	- Remove debugging-related code	for data, target in train_loader:
									    with torch.autograd.detect_anomaly(): # BAD: slows training
									        output = model(data)
									        loss = loss_fn(output, target)
									        loss.backward()
									        optimizer.step()
										for data, target in train_loader:
										    # Production-ready: no debugging overhead
										    output = model(data)
										    loss = loss_fn(output, target)
										    loss.backward()
										    optimizer.step()

Gradient Management: This subcategory refers to incorrect handling of automatic gradient tracking. It includes issues such as interrupting the flow of	Unnecessary Gradient Tracking		Possibly Framework	Over-relying on gradient tracking management, or not applied where it should be	This smell refers to the failure to disable gradient tracking during phases where gradients are not needed, such as inference or evaluation. If gradient tracking is left enabled, the system continues to build and store computation graphs, consuming memory and computational resources unnecessarily.	- Increased GPU Memory Usage During Inference
						- OOM (Out of Memory) Errors on Large Batches
							Context Manager Usage	- Employing context manager statement	model.eval()
									for data in val_loader:
									    output = model(data) # BAD: gradients are tracked by default
										model.eval()
										with torch.no_grad(): # GOOD: disables gradient tracking
										    for data in val_loader:
										        output = model(data)

	Uncleared Gradients			Not clearing gradients properly after multiple backward passes, or creating unnecessary additional computational graphs for gradient computation.	This smell refers to failing to properly reset gradients or unnecessarily enabling higher-order gradient tracking during backpropagation. Specifically, using the setting to retain the computation graph for gradient computation, often triggered by enabling higher-order derivative tracking, can lead to memory being occupied by intermediate tensors and graph structures that are no longer needed.	- Increasing GPU memory usage during training loops.
						
						- Slowed training performance
						
						- Gradients accumulate when not intended.
							Fine-Tune Gradients with torch. autograd.grad	- Avoid create_graph=True in the backward pass, instead use torch. autograd.grad for finer control over gradient computation	for data, target in dataloader:
									    output = model(data)
									    loss = loss_fn(output, target)
									    loss.backward(create_graph=True) # BAD: unnecessary graph retention
									    optimizer.step()
										for data, target in dataloader:
										    optimizer.zero_grad() # GOOD: clears gradients
										    output = model(data)
										    loss = loss_fn(output, target)
										    loss.backward() # GOOD: no extra graph unless needed
										    optimizer.step()

	Improper Gradient Use in Normalization Layers			Assigning a tensor with gradient information directly to tensors for a normalization layer that should not track gradients.	This smell refers to when tensors that are part of the computation graph are assigned to internal state variables in normalization layers, such as running means or variances. These variables are designed to hold long-term statistical summaries and are not meant to track gradients. Assigning gradient-tracking tensors to them causes PyTorch to retain the associated computation graph unnecessarily.	- Out Of Memory (OOM) errors,
						- Gradual GPU memory growth over time
							Gradient Detachment for Running Stats	- Detaching Gradients Before Assignment to self.running_mean and self.running_covar	class BadNormLayer(torch.nn.Module):
									    def __init__(self):
									        super().__init__()
									        self.running_mean = torch.zeros(10)
									
									    def forward(self, x):
									        mean = x.mean(dim=0) # has grad
									        self.running_mean = mean # BAD: attaches to computation graph
									        return x - mean
										class GoodNormLayer(torch.nn.Module):
										    def __init__(self):
										        super().__init__()
										        self.running_mean = torch.zeros(10)
										
										    def forward(self, x):
										        mean = x.mean(dim=0)
										        self.running_mean = mean.detach() # GOOD: no gradient tracking
										        return x - mean

Graph and Gradient Management Issues This category refers to errors in handling gradient computation and the underlying computational graph.	interrupting the flow of gradients too early, disabling gradient updates when they are needed, or failing to activate gradient tracking on model parameters. These mistakes prevent models from learning properly.	Mishandling training gradient			Handling the training model's gradient the same way as inference	This smell refers to failing to disable gradient tracking during inference, treating it the same as training. Since gradients are not needed during inference, allowing PyTorch to track them by default leads to unnecessary memory consumption and computational overhead.	- Slower Inference Time - Out-of-memory (OOM) errors - Slower Inference Time	Apply torch.no_grad to Disable Gradients During Inference	- Using torch.no_grad for Inference and Gradient Management	# BAD: Gradients are tracked even during inference model.eval() for data in val_loader: output = model(data) predictions = torch.argmax(output, dim=1)	# GOOD: Gradient tracking is disabled model.eval() with torch.no_grad(): for data in val_loader: output = model(data) predictions = torch.argmax(output, dim=1)

		Missing Gradient Tensors			Not storing tensors that are needed for computing gra-
					dients during the backward pass
						This smell refers to the failure to retain intermediate tensors that are necessary for computing gradients during backpropagation in neural network training	- Excessive memory consumption
							- Slow down the inference process
								Proper Tensor Management with ctx.save_for_backward()	- Use ctx.save_for_backward() in custom autograd functions to ensure tensors are managed correctly. Without it, tensors critical for gradient computation may not be properly tracked or released, leading to an accumulation of unreferenced tensors that persist in memory.	class MyModel(torch.nn.Module):
										def forward(self, x):
										x = x.detach() # BAD: removes tensor from the graph
										return x**2
											class MyModel(torch.nn.Module):
											def forward(self, x):
											return x**2 # GOOD: keeps tensor in the graph

		Accumulating Gradients in Loop			Compute the gradients of the loss with respect to the model parameters in a loop without resetting gradients or detaching tensors	Computing gradients in a loop without resetting or detaching them causes gradient buildup, leading to increased memory usage and degraded performance.	- Gradients from earlier iterations accumulate
							- Degraded system performance
								Clear Gradients at Training Loop Start	- Reset gradients at the beginning of each training iteration	for data, target in dataloader:
										output = model(data)
										loss = loss_fn(output, target)
										loss.backward() # BAD: Gradients accumulate
										optimizer.step()
											for data, target in dataloader:
											optimizer.zero_grad() # GOOD: Clears old gradients
											output = model(data)
											loss = loss_fn(output, target)
											loss.backward()
											optimizer.step()

		Nested Second Derivative Calls			Obtaining the second derivative by nesting calls	This smell refers to computing higher-order derivatives by repeatedly nesting calls to automatic differentiation functions. When the second derivative (or higher) is obtained through nested differentiation calls, PyTorch constructs new computation graphs at each step. These graphs are retained in memory unless explicitly handled.	- Gradual GPU or CPU Memory Bloat
							- Out-of-Memory (OOM) errors
								Avoid Nested grad() Calls	- Avoid Nesting grad() Calls	import torch
										from torch.autograd import grad
										
										x = torch.tensor(2.0, requires_grad=True)
										
										# First derivative
										y = x**3
										dy_dx = grad(y, x, create_graph=True)[0]
										
										# BAD: Nested grad() for second derivative
										d2y_dx2 = grad(dy_dx, x, create_graph=True)[0]	

	Graph Management: This subcategory refers to keeping computational structures in memory for longer than needed. When computation graphs are not released between training steps, they can accumulate and waste memory, reducing efficiency over time.	Graph Retention After Backward Pass			Unnecessarily retaining the computational graph after the backward pass	This smell refers to keeping the computation graph after a backward pass consumes extra memory and can cause leaks, especially in large models or datasets. It stores intermediate values that aren't needed once gradients are computed.	- Out-of-memory (OOM) errors
							- Degraded system performance
								Clear Graph and Backpropagate Immediately	- Avoid using retain_graph=True unless essential
									- Instead, computation graphs should be cleared after use, and the backward pass should be performed immediately after computing the loss for each iteration or batch.
										outputs = []
										
										for batch in dataloader:
										input = batch.to(device)
										
										output = model(input) # output has grad_fn, attached to the graph
										
										# BAD: Appending output still attached to the computation graph
										outputs.append(output)
											for batch in dataloader:
											input = batch.to(device)
											output = model(input)
											
											loss = compute_loss(output) # GOOD: compute and backprop immediately
											loss.backward()

Training Pipeline Management This category refers to inefficiencies and misconfigurations in the broader structure of the training process. It includes poor choices in model architecture, loop design, data loading strategies, and hyperparameter tuning	**Model Lifecycle Management:** Smells in this subcategory arise when model instances, computation graphs, optimizers, or callback objects are not properly initialized, reset, or released between training or evaluation runs, such as failing to call "clear_session()", reusing model objects without resetting weights or graphs, reusing optimizer instances without reinitializing their internal state (e.g., momentum buffers), or carrying over callbacks that retain tensors from prior runs.	Encoder-Decoder Inside Training Loop	**IMR K**		The encoder and decoder inside the training loop	This smell refers to repeatedly instantiating the encoder and decoder modules inside the training loop. Doing so causes new model instances to be created at every iteration, each tied to a fresh computation graph. This prevents proper release of memory and leads to accumulation of intermediate results.	- Out-of-memory (OOM) errors
							- Gradual GPU or CPU Memory Increase
							- Slower Training Over Time
								Avoid Reinitializing Encoder and Decoder Inside Training Loop	- Avoiding Reinitialization of the Encoder and Decoder Inside the Training Loop	for epoch in range(num_epochs):
										for batch in dataloader:
										input = batch.to(device)
										
										encoder = Encoder().to(device) # BAD: re-instantiating in loop
										decoder = Decoder().to(device)
										
										enc_out = encoder(input) # Builds new computation graph each time
										output = decoder(enc_out)
										
										loss = loss_fn(output, target)
										loss.backward()
										optimizer.step()
										optimizer.zero_grad()
											# GOOD: Initialize encoder and decoder once
											encoder = Encoder().to(device)
											decoder = Decoder().to(device)
											
											for epoch in range(num_epochs):
											for batch in dataloader:
											input, target = batch
											input = input.to(device)
											target = target.to(device)
											
											enc_out = encoder(input)
											output = decoder(enc_out)
											
											loss = loss_fn(output, target)
											loss.backward()
											optimizer.step()
											optimizer.zero_grad()

	Data Loading and Preprocessing: This subcategory refers to inefficient design of input pipelines. Repeatedly creating data loaders, using unsafe loading patterns, or poorly batching input can cause memory problems and reduce training throughput.	Zipped or Cycled DataLoader	**T (DIR)**		Zipping or Cycling the image DataLoader	This smell refers to combining or endlessly iterating over DataLoaders in a way that causes inefficient data loading and memory usage. When DataLoaders are paired without regard to batch alignment or are used in loops without clear stopping conditions, it can lead to mismatched batches, poor synchronization, and repeated loading of the same data.	- High CPU or Disk I/O Usage
							- Increasing Memory Usage Over Time
								Avoiding zipping or cycling the DataLoader	- Avoiding zipping or cycling the DataLoader when handling data	from torch.utils.data import DataLoader, TensorDataset
										import torch
										
										X = torch.randn(100, 10)
										y = torch.randint(0, 2, (100,))
										dataset = TensorDataset(X, y)
										loader = DataLoader(dataset, batch_size=16, num_workers=2)
										
										# BAD PRACTICE: Infinite cycling over DataLoader
										loader_iter = iter(itertools.cycle(loader))
										
										for i in range(200):
										xb, yb = next(loader_iter)
											from torch.utils.data import DataLoader, TensorDataset
											import torch
											
											X = torch.randn(100, 10)
											y = torch.randint(0, 2, (100,))
											dataset = TensorDataset(X, y)
											loader = DataLoader(dataset, batch_size=16, num_workers=2, shuffle=True)
											
											# GOOD PRACTICE: Proper epoch-based iteration
											for epoch in range(5):
											for xb, yb in loader:
											_ = xb @ torch.randn(10, 1) # Simulate

		Redundant DataLoader Instantiation	**T (DIR)**		Improper management of DataLoader with multiple workers can result in inefficient resource utilization.	This smell refers to repeatedly creating a DataLoader with multiple worker processes inside the training loop. Each instantiation spawns new worker processes without releasing the previous ones. This behavior undermines the advantages of parallel data loading and can severely impact training efficiency.	- High CPU Usage
							- Slower Training or Inference
							- Delay the data loading process
							- Undermining the effectiveness of parallel processing
							- Ultimately impeding the efficiency of model training or inference tasks
								Optimize DataLoader Workers and Persistence for Efficiency	- Optimizing DataLoader Workers and Persistence for Resource Efficiency.	data = torch.randn(1000, 10)
										labels = torch.randint(0, 2, (1000,))
										dataset = TensorDataset(data, labels)
										
										def train():
										for epoch in range(100):
										# BAD: Reinitializing DataLoader every epoch with multiple workers
										loader = DataLoader(dataset, batch_size=32, shuffle=True, num_workers=4)
										
										for batch in loader:
										x, y = batch
											from torch.utils.data import DataLoader, TensorDataset
											import torch
											
											data = torch.randn(1000, 10)
											labels = torch.randint(0, 2, (1000,))
											dataset = TensorDataset(data, labels)
											
											# GOOD: Initialize DataLoader once
											loader = DataLoader(dataset, batch_size=32, shuffle=True, num_workers=4, persistent_workers=True)
											
											def train():
											for epoch in range(100):

	Hyperparameter Configuration This subcategory refers to choosing values for training settings, such as learning rate or batch size, that are poorly suited for the task. Incorrect hyperparameters often result in training failure, wasted resources, or poor generalization.	Oversized Batch Handling	**K(MBM)**		Not Properly Managing Large Tensors, Long Sequences,
					and Large Batch Sizes
						This smell refers to developers allocating or processing excessively large tensors, long input
						sequences, or large batch sizes without considering memory limitations, compute efficiency, or hardware constraints.
							- Excessive memory consumption	Batch adjesment and Sequence Optimization	- Reducing sequence length and batch size
									- Using mixed precision with torch.cuda.amp.autocast(), and detaching or processing large tensors in smaller chunks
										`inputs = torch.randn(1024, 3, 224, 224, device='cuda')  # BAD: very large batch`
										`outputs = model(inputs)`
											`# Mixed precision reduces memory usage with minimal accuracy drop`
											`with torch.cuda.amp.autocast():`
											`    inputs = torch.randn(256, 3, 224, 224, device='cuda')  # Smaller batch`
											`    outputs = model(inputs)`

Loop Lifecycle Mismanagement This category captures memory or performance issues that arise from improper control of loop behavior	**Resource Instantiation Inside Loop** This subcategory refers to loops that repeatedly create new resources, such as models, communication groups, or tensors, without properly deallocating previous ones.	Repeated Group Creation Inside Loop	**(T/K) UR - but the objects in the loop differ**		Improper use of a loop by creating a new communication group in each iteration	This smell refers to the repeated creation of communication groups within a loop, typically in distributed training environments. A communication group defines the set of processes involved in collective operations. When a new group is created in every iteration without deallocating the previous one, references to old groups accumulate in memory.	- Gradual GPU or CPU Memory Bloat
							- Out-Of-Memory (OOM) Errors
							- Gradients from earlier iterations accumulate
								Initialize Groups Outside the Loop and Reuse Them	- Initialize the group once outside the loop and reuse it as needed.	`import torch.distributed as dist`
										
										`# BAD: Creating a new communication group inside the loop`
										`for epoch in range(10):`
										`    group = dist.new_group([0, 1])  # Memory leak risk if not cleaned up`
										`    # Use group for collective ops (e.g., dist.all_reduce(..., group=group))`
											`import torch.distributed as dist`
											
											`# GOOD: Create the communication group once`
											`group = dist.new_group([0, 1])`
											
											`for epoch in range(10):`
											`    # Reuse the same group`
											`    dist.all_reduce(torch.tensor(1).cuda(), op=dist.ReduceOp.SUM, group=group)`

	Unbounded or Infinite Looping This subcategory refers to loop structures that lack a termination condition, leading to unbounded execution.	Unbounded Loop	**(T/K) UR - but in my case there is no file in the loop**		Improper handling of an endless loop	This smell refers to a loop that lacks a proper termination condition, causing it to run indefinitely. Such unbounded execution often occurs when iterating over data or training steps without defining clear stopping criteria.	- Iterates over the dataset or training steps without termination.
							- High CPU/GPU Usage
								Avoid Infinite Loops	- Eliminating Unnecessary Endless Loops
									- Avoid itertools.cycle unless an infinite iteration is explicitly required
										`data = torch.randn(100, 10)`
										`labels = torch.randint(0, 2, (100,))`
										`dataset = TensorDataset(data, labels)`
										`loader = DataLoader(dataset, batch_size=16)`
										
										`# BAD: Infinite loop without any stopping condition`
										`for batch in itertools.cycle(loader):`
										`    # Training logic here`
										`    pass  # Loop never ends - causes high CPU/GPU usage`
											`from torch.utils.data import DataLoader, TensorDataset`
											`import torch`
											
											`data = torch.randn(100, 10)`
											`labels = torch.randint(0, 2, (100,))`
											`dataset = TensorDataset(data, labels)`
											`loader = DataLoader(dataset, batch_size=16, shuffle=True)`
											
											`# BEST PRACTICE: epoch-based training loop`
											`for epoch in range(5):  # Controlled number of epochs`
											`    for batch in loader:`
											`        # Training logic here`
											`        pass`