Categories	Sub-categories	Code Smells	Description	Potential Consequences	Name of Best Practice	Best Practice	Bad Code Example	Recommended Best Practice	Framework	Type	Static?

Resource Management: Smells in this category arise when resources, such as file handles, sessions, or CPU/GPU memory are not released properly or are consumed inefficiently.	**Memory Release:** Smells in this subcategory arise when system resources, such as file handles, sessions, GPU contexts, or background threads are opened but never properly closed or terminated. Over time, these lingering resources accumulate, exhausting available file descriptors, GPU/CPU memory, or operating-system handles.	Unclosed Resources (UR)	This smell refers to when a developer is running repeated training or evaluation loops without releasing or disposing of resources such as model objects, session data, or intermediate computation graphs after each iteration.	Accumulation of unused objects in the memory → memory leaks and reduced performance	Session Reset and Checkpointing	Clear Keras session and use Model Checkpoint Callback to release resources after each iteration.	for epoch in range(epochs):
							  # Every call to build_my_model() creates a new graph
							  model = build_my_model()
							  model.compile(optimizer='adam', loss='mse')
							  model.fit(dataset, epochs=1)
								import tensorflow as tf
								from tensorflow.keras.callbacks import ModelCheckpoint
								
								for epoch in range(epochs):
								  model = build_my_model()
								  model.compile(optimizer='adam', loss='mse')
								
								  if epoch > 0:
								  model.load_weights(f'checkpoint_epoch_{epoch-1}.h5')
								
								  checkpoint = ModelCheckpoint(
								  filepath=f'checkpoint_epoch_{epoch}.h5',
								  save_weights_only=True,
								  save_freq='epoch'
								  )
								
								  model.fit(
								  dataset,
								  epochs=1,
								  callbacks=[checkpoint]
								  )
								
								  tf.keras.backend.clear_session()
									Keras	Generic for Model Objects (also appears in Pytorch)/API-Specific for Session Data and Computation Graph
										
										One of the listed Pytorch smell: Accumulated Object References
											non-static

			This smell refers to when a developer creates new models, iterators, or graph nodes on each iteration of a training loop without releasing or deallocating previous resources.	1. accumulation of unnecessary resources → increased memory consumption
				2. Can cause memory leaks and performance degradation
					Avoid training inside a loop.	Avoid training inside a loop.	for epoch in range(10):
							  model.fit(x_train, y_train, epochs=1, batch_size=32)
								model.fit(
								  x_train,
								  y_train,
								  epochs=10,
								  batch_size=32
								)
									TensorFlow	Generic/also appears in Pytorch
										
										One of the listed Pytorch smell: Accumulated Object References
											non-static

			This smell refers to when tensors, variables, or intermediate results are created or modified on each loop iteration without proper deallocation.	1. Prevents the garbage collector from freeing their occupied memory → gradual increase in memory consumption
				2. Memory exhaustion → can cause significant memory leaks → impair system performance and efficiency
					Optimized TensorFlow Function Usage	Use "tf.function" for optimized memory management and resource cleanup in iterative workflows.	for epoch in range(EPOCHS):
							  loss = train_step_eager(x_batch, y_batch)
								import tensorflow as tf
								
								model = tf.keras.Sequential([
								  tf.keras.layers.Dense(128, activation='relu'),
								  tf.keras.layers.Dense(1)
								])
								loss_fn = tf.keras.losses.MeanSquaredError()
								optimizer = tf.keras.optimizers.Adam()
								
								@tf.function
								def train_step(x, y):
								  with tf.GradientTape() as tape:
								  y_pred = model(x, training=True)
								  loss = loss_fn(y, y_pred)
								  grads = tape.gradient(loss, model.trainable_variables)
								  optimizer.apply_gradients(zip(grads, model.trainable_variables))
								  return loss
								
								for epoch in range(EPOCHS):
								  for x_batch, y_batch in dataset:
								  loss = train_step(x_batch, y_batch)
									TensorFlow	Generic/also appears in Pytorch
										
										One of the listed Pytorch smell: Accumulated Object References
											non-static

			This smell refers to when a developer dynamically allocates new tensor objects in each loop iteration without properly deallocating or resetting them.	1. Unused memory allocations accumulate with each iteration
				2. Inefficient memory utilization → memory leaks, performance degradation, and potential out-of-memory (OOM) errors
					Resource Reuse and Memory Clearing	Reuse resources, manage memory growth, and efficiently load data.	memory_used = []
							for i in range(500):
							  # NEW dataset object & iterator on every iteration
							  data = (tf.data.Dataset.from_tensor_slices(
							  np.random.uniform(size=(10, 500, 500)))
							  .prefetch(64).repeat(-1).batch(3))
							  it = data.make_initializable_iterator()
							  next_el = it.get_next()
							
							  with tf.Session() as sess:
							  sess.run(it.initializer)
							  sess.run(next_el)
							
							  memory_used.append(psutil.virtual_memory().used / 2**30)
							  tf.reset_default_graph()
								data = (tf.data.Dataset.from_tensor_slices(
								  np.random.uniform(size=(10, 500, 500)))
								  .prefetch(64).repeat(-1).batch(3))
								it = data.make_initializable_iterator()
								next_el = it.get_next()
								
								with tf.Session() as sess:
								  sess.run(it.initializer)
								  for _ in range(500):
								  batch = sess.run(next_el)
									TensorFlow	Generic/also appears in Pytorch
										
										One of the listed Pytorch smell: Accumulated Object References
											non-static

			This smell refers to situations where a developer opens resources such as files without properly closing them. It can occur on its own or in combination with an infinite loop. When combined with an infinite loop, the resource allocations are repeated continuously without release, amplifying the severity of the issue.	1. Prevents the system from releasing resources → can cause resource leaks
				2. Can cause file handles to remain open → consuming memory and potentially locking the file → prevents further access or modifications.
				3. Accumulation of open file handles → system performance degradation or even exhaust the available file descriptors
					Loop Exit Condition Enforcement	Ensure all loops have a proper exit condition.	for path in file_paths:
							  f = open(path, 'r')
							  data = f.read()
							  process(data)
								for path in file_paths:
								  with open(path, 'r') as f:
								  data = f.read()
								  process(data)
									TensorFlow	Generic/also appears in Pytorch	non-static

		Unclosed-Session (US)	This smell refers to when a developer opens sessions or builds computational graphs without explicitly closing or resetting them.	1. Resources remain allocated → consuming memory even after if no longer needed
				2. Can cause memory leak
					Session and Resource Cleanup	Ensure proper session management and resource cleanup in TensorFlow.	for step in range(200):
							  with tf.Session() as sess:
							  # 'test' op was defined outside; feedForwardStep creates new ops each call
							  retour = sess.run(test, feed_dict={x: batch_data})
							  test2 = feedForwardStep(retour, W_to_output, b_output)
								for step in range(200):
								  with tf.Graph().as_default(), tf.Session() as sess:
								  retour = sess.run(test, feed_dict={x: batch_data})
								  test2 = feedForwardStep(retour, W_to_output, b_output)
									TensorFlow	API-Specific	non-static

		GPU Released Memory Failure (GRMF)	This smell refers to when a developer allocates tensors or models on the GPU without explicitly freeing or resetting GPU memory, relying instead on garbage collection, which primarily manages CPU memory	1. Memory leaks → inefficient memory utilization
				2. Accumulation of unused memory → out-of-memory (OOM) errors, performance degradation, or even training halts
				3. May delay the release of unused resources
				4. Slower computations
					Explicit GPU Memory Cleanup	Explicitly delete unused objects, invoke "gc.collect()" to prompt memory cleanup, and call "tf.keras.backend.clear_session()" between runs to fully release resources.	for run_id in range(25):
							  model = build_model(**hparams)) # each time: new CNN on GPU
							  model.fit(train_ds, epochs=2)
								import tensorflow as tf, gc
								
								for run_id in range(25):
								  model = build_model(**hparams))
								  model.fit(train_ds, epochs=2)
								
								  del model
								  tf.keras.backend.clear_session()
								  gc.collect()
									TensorFlow	Generic/also appears in Pytorch
										
										One of the listed Pytorch smell: Unreleased GPU Memory
											non-static

	Augmentation Resource Leaks: Smells in this subcategory arise when data-augmentation pipelines retain intermediate buffers or cached augmented samples without ever freeing them.	Image Buffer Accumulation (IBA)	This smell refers to when a developer loads or processes images without using context managers or explicitly releasing temporary buffers.	Can cause resource leaks e.g. CPU memory for augmented images isn't freed and GPU buffers aren't cleared, leading to gradual memory growth, potential out-of-memory errors, and degraded performance.	Context Manager Resource Control	Proactively manage resources by employing context managers or ensuring explicit disposal.	def augment_images_tf(paths):
							  images = []
							  for p in paths:
							  f = tf.io.gfile.GFile(p, 'rb') # file opened but never closed
							  data = f.read()
							  img = tf.image.decode_jpeg(data, channels=3)
							  img = tf.image.random_flip_left_right(img)
							  images.append(img)
							  return images
								def augment_images_tf(paths):
								  images = []
								  for p in paths:
								  # context manager ensures file handle is closed promptly
								  with tf.io.gfile.GFile(p, 'rb') as f:
								  data = f.read()
								  img = tf.image.decode_jpeg(data, channels=3)
								  img = tf.image.random_flip_left_right(img)
								  images.append(img)
								  return images
									TensorFlow	Generic/also appears in Pytorch	non-static

	Tensor and Variable Management: Smells in this subcategory arise when tensors, gradient buffers, or model variables are mismanaged, through unnecessary allocations, retained references, or missing required transformations.	Unreleased Tensor Reference (UTR)	This smell refers to when a developer reassigns or stores new tensor objects without releasing references to the previous ones.	1. Memory occupied by the tensors remains allocated unnecessarily
				2. Prevents the proper deallocation of memory → tensors to persist in memory even after they are no longer required
					Manual Tensor Disposal	Manually dispose tensors before reassignment.	let ys = model.predict(xs);
							
							for (let i = 0; i < 1_000; ++i) {
							  ys = tf.add(ys, tf.scalar(1));
							}
							
								let ys = model.predict(xs);
								for (let i = 0; i < 1_000; ++i) {
								  const next = ys.add(1);
								  ys.dispose();
								  ys = next;
								}
								
									TensorFlow	Generic/also appears in Pytorch
										
										One of the listed Pytorch smell: 1. Lingering References
										2. Unreleased Tensor/Model Reference
											non-static

		Shape Mismatch Leak (SML)	This smell refers to when a developer omits necessary tensor reshaping before operations, triggering implicit broadcasting or oversized buffers.	1. Increased memory consumption 2. Memory leaks	Preemptive Tensor Reshaping	Reshape tensors before operations to avoid broadcasting overhead.	# `batch_labels` has shape (N, 1), `predictions` has shape (N,) predictions = model(batch_features, training=False) errors = tf.abs(tf.subtract(batch_labels, predictions)) 	predictions = model(batch_features, training=False) batch_labels = tf.reshape(batch_labels, tf.shape(predictions)) errors = tf.abs(batch_labels - predictions) 	TensorFlow	Generic/also appears in Pytorch	non-static

Training Pipeline Management: Smells in this category arise when any phase of the training loop, including model lifecycle management, hyperparameter configuration, data loading and preprocessing, checkpointing, learning-rate scheduling, stateful layer modes (e.g., batch normalization), callback handling, data shuffling, or early stopping is misconfigured or handled improperly.	**Data Loading and Preprocessing:** Smells in this subcategory arise when data-loading or preprocessing pipelines are designed or configured inefficiently, whether by repeatedly recreating datasets or iterators, failing to cache or prefetch, applying expensive transformations before filtering or batching, loading entire datasets into memory at once, or causing unnecessary host–device copies.	Dataset-Iterator Retention (DIR)	This smell refers to when a developer instantiates new "tf.data.Dataset" operations or iterators on each call to "model.predict()" (or inside an inference loop).	Continual creation of iterators → ressource leak	Efficient Iterator Management	Passing "next_iterator()" to "predict()" instead of the full dataset.	import tensorflow as tf
							import numpy as np
							
							SIZE = 5000
							
							inp = tf.keras.layers.Input(shape=(SIZE,), dtype='float32')
							x = tf.keras.layers.Dense(units=SIZE)(inp)
							
							model = tf.keras.Model(inputs=inp, outputs=x)
							
							np_data = np.random.rand(1, SIZE)
							ds = tf.data.Dataset.from_tensor_slices(np_data).batch(1).repeat()
							
							while True:
							  model.predict(x=ds, steps=1)
							  preds = model.predict(dataset)
							
								import tensorflow as tf
								import numpy as np
								
								SIZE = 5000
								
								inp = tf.keras.layers.Input(shape=(SIZE,), dtype='float32')
								x = tf.keras.layers.Dense(units=SIZE)(inp)
								
								model = tf.keras.Model(inputs=inp, outputs=x)
								
								np_data = np.random.rand(1, SIZE)
								ds = tf.data.Dataset.from_tensor_slices(np_data).batch(1).repeat()
								it = tf.data.make_one_shot_iterator(ds)
								tensor = it.get_next()
								
								while True:
								  model.predict(x=tensor, steps=1)
								
									TensorFlow	Generic/also appears in Pytorch
										
										One of the listed Pytorch smell: 1. Redundant DataLoader Instantiation
										
										2. Zipped or Cycled DataLoader
											non-static

	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.	Session Pile-Up (SPU)	This smell refers to when a developer reuses Keras/TensorFlow model instances or sessions without invoking "clear_session()" or resetting the computation graph.	1. Accumulation of outdated ressources
				
				2. Can result in memory leaks
				3. Reduced system performance
					Avoid Model Recreation in Loops	Avoid Recreating Models in a loop.	for i in range(10):
							  model = create_model() # builds a new graph each iteration
							  model.compile(optimizer='adam',
							  loss='sparse_categorical_crossentropy',
							  metrics=['accuracy'])
							  model.fit(x_train, y_train, epochs=1)
							
								import tensorflow as tf
								
								for i in range(10):
								  tf.keras.backend.clear_session() # clear old graphs/sessions
								  model = create_model()
								  model.compile(optimizer='adam',
								  loss='sparse_categorical_crossentropy',
								  metrics=['accuracy'])
								  model.fit(x_train, y_train, epochs=1)
								
									TensorFlow	API-Specific	non-static

		Improper Model Reuse (IMR)	This smell refers to a developer reusing model instances across training or evaluation loops without clearing or resetting model objects, weights, or computation graphs between iterations.	1. Accumulation of redundant computation graphs, layer weights, and tensors → increased memory use and degraded throughput.
				2. Memory leaks and wasted compute cycles → compromised stability and scalability of the workflow.
					Resource Reset and Cleanup	Clear or reset resources.	cont=0
							while cont<20:
							  cont+=1
							
							  img_to_train_discr=image_generator(8)
							  #it returns a tuple(image, 0/1)
							  discr.train_on_batch(img_to_train_discr[0], img_to_train_discr[1])
							  img_to_train_gan=image_generator_for_gan(8)
							  gan.train_on_batch(img_to_train_gan[0],img_to_train_gan[1])
							
								cont=0
								LEAK_FLUSH_EVERY = 3
								discr, gan = build_models() # helper function that returns compiled models
								
								while cont<20:
								  cont+=1
								
								  img_to_train_discr=image_generator(8)
								  #it returns a tuple(image, 0/1)
								  discr.train_on_batch(img_to_train_discr[0], img_to_train_discr[1])
								  img_to_train_gan=image_generator_for_gan(8)
								  gan.train_on_batch(img_to_train_gan[0],img_to_train_gan[1])
								  if cont % LEAK_FLUSH_EVERY == 0 :
								  tf.keras.backend.clear_session()
								  gc.collect()
								  discr, gan = build_models()
								
									Keras	Generic/API-Specific for Computation Graph/also appears in Pytorch
										
										One of the listed Pytorch smell: 1. Encoder-Decoder Inside Training Loop
										2. Tracing Inside Loop Without Cleanup
											non-static

	Hyperparameter Configuration: Smells in this subcategory arise when hyperparameter settings (e.g., learning rate, batch size, optimizer parameters, regularization factors) are chosen without regard for hardware capabilities, data characteristics, or convergence properties, such as using a learning rate that causes instability, misconfiguring regularization or optimizer parameters, or adopting dropout rates that slow down learning.	Minibatch Mismatch (MBM)	This smell refers to when a developer chooses a minibatch size that does not align with the hardware's capabilities, either excessively large or unnecessarily small, leading to suboptimal resource usage during training.	1. Possible increased memory consumption → compromise efficiency and stability of ML models
				2. Can lead to Out-of-memory (OOM) errors
				3. Higher computational overhead → limited scalability
					Adjusting Batch Size	Lower batch size.	# Assume `dataset` is a tf.data.Dataset of (features, labels)
							
							model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
							
							model.fit(
							  dataset,
							  epochs=10,
							  batch_size=1024,
							)
							
								model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
								model.fit(
								  dataset,
								  epochs=10,
								  batch_size=32,
								)
								
									Keras	Generic/also appears in Pytorch
										
										One of the listed Pytorch smell: Oversized Batch Handling
											non-static

Graph Management: Smells in this category arise when	**Misuse of Graph Constants:** Smells in this subcategory arise when developers embed large data arrays, lookup tables, configuration objects, or other unnecessary constant values directly into the computational graph.	Graph-Constant Bottleneck (GCB)	This smell refers to when a developer embeds large data arrays as graph constants.	1. Accumulation of large, unused memory objects
				2. Increased memory consumption and exacerbating memory leaks
					Use Placeholders for Large Data	Use placeholders when working with large datasets.	for _ in range(1000):
							  large_const = tf.constant(large_np_array)
							  sess.run(large_const)
							
								ph = tf.placeholder(dtype=large_np_array.dtype,
								  shape=large_np_array.shape)
								for _ in range(1000):
								  sess.run(fetch_op, feed_dict={ph: large_np_array})
								
									TensorFlow	API-Specific	Static

the computational graph is mismanaged, leading to an unnecessarily large graph, slower performance, excessive memory use, or difficulty exporting and reusing the model, whether through improper handling of constants, mixing training and inference nodes, continuously adding operations inside loops, inserting non-serializable constructs, or failing to remove outdated fragments.	**Improper Graph Reuse:** Smells in this category arise when the computational graph is mismanaged, leading to an unnecessarily large graph, slower performance, excessive memory use, or difficulty exporting and reusing the model, whether through improper handling of constants, mixing training and inference nodes, continuously adding operations inside loops, including pieces that can't be saved or exported, or failing to remove outdated fragments.	Unbounded Graph Expansion (UGE)	This smell refers to when a developer adds new operations to TensorFlow's default graph inside a loop without resetting or isolating the graph between iterations.	1. Retention of unnecessary operations, consuming memory and computational resources
				2. Inefficient memory utilization and potential memory leaks → system performance degradation and increase memory consumption
					Graph Isolation and Optimization	Use separate graphs and optimize graph construction in TensorFlow.	`def read_tensor_from_image_file(file_name, input_height=299, input_width=299, input_mean=0, input_std=255):`
							`  input_name = "file_reader"`
							`  output_name = "normalized"`
							`  file_reader = tf.read_file(file_name, input_name)`
							
							`  image_reader = tf.image.decode_jpeg(file_reader, channels = 3,`
							`  name='jpeg_reader')`
							`  float_caster = tf.cast(image_reader, tf.float32)`
							`  dims_expander = tf.expand_dims(float_caster, 0)`
							`  resized = tf.image.resize_bilinear(dims_expander, [input_height, input_width])`
							`  normalized = tf.divide(tf.subtract(resized, [input_mean]), [input_std])`
							`  sess = tf.Session()`
							`  result = sess.run(normalized)`
							
							`  return result`
								`def read_tensors_from_image_files(file_names, input_height=299, input_width=299, input_mean=0, input_std=255):`
								`  with tf.Graph().as_default():`
								`  input_name = "file_reader"`
								`  output_name = "normalized"`
								`  file_name_placeholder = tf.placeholder(tf.string, shape=[])`
								`  file_reader = tf.read_file(file_name_placeholder, input_name)`
								`  image_reader = tf.image.decode_jpeg(file_reader, channels = 3,`
								`  name='jpeg_reader')`
								`  float_caster = tf.cast(image_reader, tf.float32)`
								`  dims_expander = tf.expand_dims(float_caster, 0)`
								`  resized = tf.image.resize_bilinear(dims_expander, [input_height, input_width])`
								`  normalized = tf.divide(tf.subtract(resized, [input_mean]), [input_std])`
								
								`  with tf.Session() as sess:`
								`  for file_name in file_names:`
								`  yield sess.run(normalized, {file_name_placeholder: file_name})`
									TensorFlow	API-Specific	non-static

Framework/Abstraction Usage: Smells in this category arise whenever code steps outside the framework's built-in, high-level interfaces, choosing manual loops, primitive ops, or direct device/resource calls instead of the intended helpers. This broad bucket covers any misuse of layers, optimizers, data pipelines, or training APIs that bypasses the safety, optimizations, and readability of the framework's abstractions.	**Abstraction Violation:** Smells in this subcategory arise when developers bypass a given high-level construct by using lower-level primitives or manual manipulations in its place. These violations erode the abstraction layers that the framework provides, making code harder to understand, test, and maintain.	Using Lambda layers to perform complex or repeated operations	This smell refers to when a developer uses Lambda layers to perform complex or repeated operations instead of simple, one-off tensor transformations for which they're intended.	Unnecessary computational overhead → degraded model efficiency and hindered scalability of ML systems	Subclass Custom Operations	Encapsulate custom operations in subclasses.	`# apply IN and BN on the input tensor independently`
							`x_in = InstanceNormalization(axis=3)(x)`
							`x_bn = BatchNormalization(axis=3)(x)`
							
							`# addition of the feature maps outputed by IN and BN`
							`x = Add()([x_in, x_bn])`
								`class Crop(keras.layers.Layer):`
								`  def __init__(self, dim, start, end, **kwargs):`
								`  """`
								`  Slice the tensor on the last dimension, keeping what is between start`
								`  and end.`
								`  Args`
								`  dim  (int)  : dimension of the tensor (including the batch dim)`
								`  start (int)  : index of where to start the cropping`
								`  end  (int)  : index of where to stop the cropping`
								`  """`
								`  super(Crop, self).__init__(**kwargs)`
								`  self.dimension = dim`
								`  self.start = start`
								`  self.end = end`
								
								`  def call(self, inputs):`
								`  if self.dimension == 0:`
								`  return inputs[self.start:self.end]`
								`  if self.dimension == 1:`
								`  return inputs[:, self.start:self.end]`
								`  if self.dimension == 2:`
								`  return inputs[:, :, self.start:self.end]`
								`  if self.dimension == 3:`
								`  return inputs[:, :, :, self.start:self.end]`
								`  if self.dimension == 4:`
								`  return inputs[:, :, :, :, self.start:self.end]`
								
								`  def compute_output_shape(self, input_shape):`
								`  return (input_shape[:-1] + (self.end - self.start,))`
								
								`  def get_config(self):`
								`  config = {`
								`  'dim': self.dimension,`
								`  'start': self.start,`
								`  'end': self.end,`
								`  }`
								`  base_config = super(Crop, self).get_config()`
								`  return dict(list(base_config.items()) + list(config.items()))`
									Keras	API-Specific	Static

				Retention of intermediate tensors and the computation graph → Can cause an accumulation of memory → prevents effective garbage collection → memory leaks	Replace Lambda with Custom Layers	Replace Lambda with Custom Layers			TensorFlow	API-Specific	Static

		Primitive API Leakage (PAL)	This smell refers to when a developer invokes low-level TensorFlow operations directly inside a Keras model definition instead of using the corresponding Keras layers.	1. May create challenges in managing the model's structure and hinder the effective use of Keras' higher-level functionalities
				2. May limit the portability and flexibility of the model
				3. Reduces code modularity → complicates debugging and maintenance
					TensorFlow Operation Encapsulation	Encapsulate TensorFlow operations.	`# Bad Code Example (inside Model.call or Sequential pipeline)`
							
							`x = tf.multiply(tf.reshape(x, (-1, 32, 2)), 0.5)`
								`# Recommended Best Practice`
								
								`x = Reshape((32, 2))(x)`
								`x = Multiply()([x, 0.5])`
									TensorFlow	API-Specific	Static

Environment and Configuration Management: Smells in this category arise from misconfiguring the execution environment or its settings, whether through version or dependency conflicts, missing or conflicting environment variables, malformed configuration files, container/VM or filesystem/permission errors, incorrect locale/timezone, or unconstrained resource allocation.	**Environment Setup Issues:** Smells in this subcategory are caused by misconfiguring any aspect of the runtime environment, such as version or dependency conflicts, incorrect or missing environment variables, malformed configuration files, container/VM setup errors, file-system or permission problems, or locale/timezone mismatches.	Library Path Mismatch (LPM)	This smell refers to when a developer configures CUDA-related environment variables (such as CUDA_HOME, PATH, or LD_LIBRARY_PATH) to different or mismatched CUDA installations instead of a single, consistent location.	1. Disrupts the proper loading and initialization of CUDA libraries and drivers → possible resources leaks
				2. Improper GPU memory allocation/deallocation → memory fragmentation
				3. Can lead to memory leaks and inefficient GPU utilization
				4. Ultimately, may compromise system stability and performance
					Consistent Environment Configuration	Ensure consistent environment configuration for CUDA paths.	`export CUDA_HOME=/usr/local/cuda-7.5`
							`export PATH=/usr/local/cuda-7.0/bin:$PATH`
							`export LD_LIBRARY_PATH=/usr/local/cuda-7.5/lib64:$LD_LIBRARY_PATH`
								`export CUDA_HOME=/usr/local/cuda-7.5`
								`export PATH=$CUDA_HOME/bin:$PATH`
								`export LD_LIBRARY_PATH=$CUDA_HOME/lib64:$LD_LIBRARY_PATH`
									Keras	Generic	non-static

	Resource Configuration Issue: Smells in this subcategory are caused by improperly tuning or allocating finite system resources, such as unconstrained parallelism, excessive memory or disk I/O limits, or misconfigured GPU/CPU usage.	Unecessary Parallelism (UP)	This smell refers to when a developer parallelizes model training across all available cores (e.g. using n_jobs=-1 or similar) without limiting the number of worker processes or dispatch control.	1. Excessive parallel jobs → resource exhaustion and memory leaks
				2. accumulation of unused or improperly managed processes and threads → to memory leaks
				3. Ultimately, may compromise system stability and performance
					GridSearchCV Parallelism Control	Controll parallelism in GridSearchCV.	`from sklearn.ensemble import RandomForestClassifier`
							`from sklearn.model_selection import GridSearchCV`
							
							`estimator  = RandomForestClassifier()`
							`param_grid = {'n_estimators': [50, 100], 'max_depth': [10, 20]}`
							
							`# Uses all cores (n_jobs=-1), can exhaust memory and leak resources`
							`grid = GridSearchCV(estimator, param_grid, n_jobs=-1)`
							`grid.fit(X_train, y_train)`
								`from sklearn.ensemble import RandomForestClassifier`
								`from sklearn.model_selection import GridSearchCV`
								
								`estimator  = RandomForestClassifier()`
								`param_grid = {'n_estimators': [50, 100], 'max_depth': [10, 20]}`
								
								`# Limit to 4 cores and dispatch only 2× as many jobs at once`
								`grid = GridSearchCV(`
								`  estimator,`
								`  param_grid,`
								`  n_jobs=4,`
								`  pre_dispatch='2*n_jobs'`
								`)`
								`grid.fit(X_train, y_train)`
									Keras	Generic/also appears in Pytorch	non-static