

Project Summary

Directory Structure

```
measurement-free-quantum-classifier/  
  configs/  
    paths.yaml  
  src/  
    __init__.py  
    IQL/  
      __init__.py  
      models/  
        fixed_memory_iqc.py  
        static_isdo_model.py  
        adaptive_iqc.py  
        __init__.py  
      learning/  
        class_state.py  
        update.py  
        metrics.py  
        memory_bank.py  
        calculate_prototype.py  
        __init__.py  
      backends/  
        base.py  
        prime_b.py  
        hadamard.py  
        transition.py  
        exact.py  
        __init__.py  
      regimes/  
        regime3c_adaptive.py  
        regime3a_wta.py  
        regime2_online.py  
        __init__.py  
      inference/  
        weighted_vote_classifier.py  
        __init__.py  
      encoding/  
        embedding_to_state.py  
        __init__.py  
      baselines/  
        static_isdo_classifier.py  
  utils/  
    common_backup.py  
    common.py  
    paths.py  
    seed.py  
    label_utils.py  
    __init__.py
```

```
data/
  pcam_loader.py
  transforms.py
  __init__.py
quantum/
  compute_qsvm_kernel.py
  __init__.py
training/
  test_adaptive_iqc.py
  verify_consistency.py
  run_final_comparison.py
  compare_best_iqc_vs_classical.py
  test_fixed_memory_iqc.py
  validate_backends.py
  test_static_isdo_model.py
  compare_iqc_algorithms.py
  protocol_online/
    train_perceptron.py
  protocol_static/
    evaluate_isdo_k_sweep.py
    evaluate_static_isdo.py
  classical/
    make_embedding_split.py
    train_embedding_models.py
    extract_embeddings.py
    visualize_embeddings.py
    train_cnn.py
    verify_embeddings.py
    visualize_pcam.py
  protocol_adaptive/
    consolidate_memory.py
    train_adaptive_memory.py
classical/
  cnn.py
  __init__.py
```

File: configs/paths.yaml

```
base_root: "/home/tarakesh/Work/Repo/measurement-free-quantum-classifier"

paths:
  dataset: "dataset"
  checkpoints: "results/checkpoints"
  embeddings: "results/embeddings"
  figures: "results/figures"
  logs: "results/logs"
  class_prototypes: "results/embeddings/class_prototypes"
  artifacts: "results/artifacts"

class_count:
```

```
K: 3
K_values: [1, 2, 3, 5, 7, 11, 13, 17, 19, 23]
```

File: src/init.py

File: src/IQL/init.py

File: src/IQL/models/fixed_memory_iqc.py

```
# src/IQL/models/fixed_memory_iqc.py

import os
import numpy as np

from src.utils.paths import load_paths
from src.IQL.learning.class_state import ClassState
from src.IQL.learning.memory_bank import MemoryBank
from src.IQL.regimes.regime3a_wta import WinnerTakeAll
from src.IQL.inference.weighted_vote_classifier import
WeightedVoteClassifier
from src.IQL.backends.exact import ExactBackend
from src.IQL.learning.calculate_prototype import generate_prototypes
from src.utils.label_utils import ensure_binary

class FixedMemoryIQC:
    """
    Fixed-Memory Interference Quantum Classifier (IQC)

    Training pipeline:
    1. Generate K prototypes per class (if missing)
    2. Initialize Kx2 quantum memory states
    3. Train with Winner-Take-All (Regime-3A)
    4. Freeze memory
    """

    def __init__(self, K: int, eta: float = 0.1, backend=None):
        self.K = K
        self.eta = eta
        self.backend = backend or ExactBackend()

        self.memory_bank = None
```

```

        self.trainer = None
        self.classifier = None

    def _ensure_prototypes(self, X, y):
        """
        Generate prototypes if they do not already exist.
        """
        _, PATHS = load_paths()
        proto_base = PATHS["class_prototypes"]
        proto_dir = os.path.join(proto_base, f"K{self.K}")

        os.makedirs(proto_dir, exist_ok=True)
        y_binary = ensure_binary(y)
        generate_prototypes(
            X=X,
            y=y_binary,
            K=self.K,
            output_dir=proto_dir
        )

    def _load_prototypes(self):
        _, PATHS = load_paths()
        proto_dir = PATHS["class_prototypes"]

        vectors = []
        for cls in [0, 1]:
            for i in range(self.K):
                path = os.path.join(
                    proto_dir, f"K{self.K}", f"class{cls}_proto{i}.npy"
                )
                vectors.append(np.load(path))
        return vectors

    def fit(self, X, y):
        # -----
        # Step 1: ensure prototypes exist
        # -----
        self._ensure_prototypes(X, y)

        # -----
        # Step 2: initialize memory bank
        # -----
        proto_vectors = self._load_prototypes()
        class_states = [
            ClassState(v, backend=self.backend)
            for v in proto_vectors
        ]
        self.memory_bank = MemoryBank(class_states)

        # -----
        # Step 3: Regime-3A training
        # -----
        self.trainer = WinnerTakeAll(
            memory_bank=self.memory_bank,

```

```

        eta=self.eta,
        backend=self.backend
    )
    self.trainer.fit(X, y)

    # -----
    # Step 4: freeze → inference
    # -----
    self.classifier = WeightedVoteClassifier(self.memory_bank)
    return self

def predict(self, X):
    if self.classifier is None:
        raise RuntimeError("Model not trained. Call fit() first.")
    return [self.classifier.predict(x) for x in X]

```

File: src/IQL/models/static_isdo_model.py

```

# src/IQL/models/static_isdo_model.py

from src.IQL.baselines.static_isdo_classifier import StaticISDOClassifier
from src.utils.paths import load_paths
from src.IQL.learning.calculate_prototype import generate_prototypes
import os

class StaticISDOModel:
    """
    Static ISDO Model (Baseline)

    - K prototypes per class
    - No learning
    - Fixed interference reference state |chi>
    """

    def __init__(self, K: int):
        _, PATHS = load_paths()
        self.proto_dir = PATHS["class_prototypes"]
        self.K = K
        self.classifier = None

    def _ensure_prototypes(self, X, y):
        """
        Generate prototypes if they do not already exist.
        """
        _, PATHS = load_paths()
        proto_base = PATHS["class_prototypes"]
        proto_dir = os.path.join(proto_base, f"K{self.K}")
        os.makedirs(proto_dir, exist_ok=True)
        generate_prototypes(
            X=X,

```

```

        y=y,
        K=self.K,
        output_dir=proto_dir,
        seed = 42
    )

def fit(self,X,y):
    """
    Offline preparation only.
    Loads precomputed prototypes and builds classifier.
    """
    self._ensure_prototypes(X,y)
    self.classifier = StaticISDOClassifier(
        proto_dir=self.proto_dir,
        K=self.K
    )
    return self

def predict(self, X):
    if self.classifier is None:
        raise RuntimeError("Model not fitted. Call fit() first.")
    return self.classifier.predict(X)

```

File: src/IQL/models/adaptive_iqc.py

```

import numpy as np

from src.IQL.learning.class_state import ClassState
from src.IQL.learning.memory_bank import MemoryBank
from src.IQL.regimes.regime3c_adaptive import AdaptiveMemory
from src.IQL.regimes.regime3a_wta import WinnerTakeAll
from src.IQL.inference.weighted_vote_classifier import
WeightedVoteClassifier
from src.IQL.backends.exact import ExactBackend
from src.IQL.learning.calculate_prototype import generate_prototypes
from src.utils.paths import load_paths
from src.utils.label_utils import ensure_polar, ensure_binary

import os

class AdaptiveIQC:
    """
    Final Adaptive Interference Quantum Classifier (IQC)

    Pipeline:
    1. Prototype generation (offline, classical)
    2. Adaptive growth (Regime-3C)
    3. Consolidation (Regime-3A)
    4. Inference-only classifier
    """

```

```

"""

def __init__(
    self,
    K_init=3,
    eta=0.1,
    percentile=5,
    backend=None,
    consolidate=True,
):
    self.K_init = K_init
    self.eta = eta
    self.percentile = percentile
    self.backend = backend or ExactBackend()
    self.consolidate = consolidate

    self.memory_bank = None
    self.regime3c = None
    self.classifier = None

def _initialize_memory(self, X, y):
    _, PATHS = load_paths()
    proto_dir = os.path.join(PATHS["class_prototypes"],
                              f"K{self.K_init}")

    y_binary = ensure_binary(y)
    generate_prototypes(
        X=X,
        y=y_binary,
        K=self.K_init,
        output_dir=proto_dir,
        seed =42,
    )

    class_states = []
    for cls in [0, 1]:
        for i in range(self.K_init):
            vec = np.load(
                os.path.join(proto_dir, f"class{cls}_proto{i}.npy")
            )
            class_states.append(
                ClassState(vec, backend=self.backend)
            )

    self.memory_bank = MemoryBank(class_states)

def fit(self, X, y):
    # -----
    # Step 1 – initialize memory
    # -----
    y = ensure_polar(y)
    self._initialize_memory(X, y)

    # -----

```

```

# Step 2 – adaptive growth + pruning (Regime-3C)
# -----
self.regime3c = AdaptiveMemory(
    memory_bank=self.memory_bank,
    eta=self.eta,
    percentile=self.percentile,
    backend=self.backend,
)
self.regime3c.fit(X, y)

# -----
# Step 3 – optional consolidation (Regime-3A)
# -----
if self.consolidate:
    consolidator = WinnerTakeAll(
        memory_bank=self.memory_bank,
        eta=self.eta,
        backend=self.backend,
    )
    consolidator.fit(X, y)

# -----
# Step 4 – freeze & inference
# -----
self.classifier = WeightedVoteClassifier(self.memory_bank)
return self

def predict(self, X):
    if self.classifier is None:
        raise RuntimeError("Model not trained.")
    return [self.classifier.predict(x) for x in X]

```

File: src/IQL/models/**init**.py

File: src/IQL/learning/class_state.py

```

import numpy as np
from src.IQL.backends.base import InterferenceBackend

def normalize(v: np.ndarray) -> np.ndarray:
    norm = np.linalg.norm(v)
    if norm == 0:
        raise ValueError("Zero-norm vector cannot be normalized")
    return v / norm

```



```

class ClassState:
    """
    Represents the quantum class memory |chi>.
    Invariant: ||chi|| = 1 always.
    """

    def __init__(self, vector: np.ndarray, backend: InterferenceBackend):
        self.vector = normalize(vector.astype(np.complex128))
        self.backend = backend

    def score(self, psi: np.ndarray) -> float:
        """
        ISDO score: Re <chi | psi>
        """
        return self.backend.score(self.vector, psi)

    def update(self, delta: np.ndarray):
        """
        Update |chi> <- normalize(|chi> + delta)
        """
        self.vector = normalize(self.vector + delta)

```

File: src/IQL/learning/update.py

```

import numpy as np
from src.IQL.backends.base import InterferenceBackend

def update(
    chi: np.ndarray,
    psi: np.ndarray,
    y: int,
    eta: float,
    backend: InterferenceBackend,
):
    """
    Regime-2 update rule (quantum perceptron):

    If y * Re<chi|psi> >= 0:
        no update
    else:
        chi <- normalize(chi + eta * y * psi)
    """
    s = backend.score(chi, psi)

    if y * s >= 0:
        return chi, False # correct classification

    delta = eta * y * psi

```

```
chi_new = chi + delta
chi_new = chi_new / np.linalg.norm(chi_new)

return chi_new, True
```

File: src/IQL/learning/metrics.py

```
import numpy as np

def summarize_training(history: dict):
    margins = np.array(history["margins"])
    updates = np.array(history["updates"])

    return {
        "mean_margin": float(margins.mean()),
        "min_margin": float(margins.min()),
        "num_updates": int(updates.sum()),
        "update_rate": float(updates.mean()),
    }
```

File: src/IQL/learning/memory_bank.py

```
from src.IQL.learning.class_state import ClassState

class MemoryBank:
    def __init__(self, class_states):
        self.class_states = class_states

    def scores(self, psi):
        return [
            cs.score(psi)
            for cs in self.class_states
        ]

    def winner(self, psi):
        scores = self.scores(psi)
        idx = int(max(range(len(scores)), key=lambda i: abs(scores[i])))
        #idx = int(max(range(len(scores)), key=lambda i: scores[i])) ##
        causes lower score ??
        return idx, scores[idx]

    def add_memory(self, chi_vector, backend):
        self.class_states.append(ClassState(chi_vector, backend=backend))

    def remove(self, idx):
        """Remove memory at index idx."""
```

```

        if 0 <= idx < len(self.class_states):
            del self.class_states[idx]

```

File: src/IQL/learning/calculate_prototype.py

```

import os
import numpy as np
from sklearn.cluster import KMeans

from src.utils.seed import set_seed

# -----
# Helper: quantum-safe normalization
# -----
def to_quantum_state(x):
    x = np.asarray(x, dtype=np.float64).reshape(-1)
    x = x / np.sqrt(np.sum(x ** 2))
    assert np.isclose(np.sum(x ** 2), 1.0, atol=1e-12)
    return x

# -----
# Core function (IMPORTABLE)
# -----
def generate_prototypes(X, y, K, output_dir, seed=42):
    """
    Generate K prototypes per class using KMeans clustering.

    Args:
        X (np.ndarray): embeddings, shape (N, D)
        y (np.ndarray): labels in {0,1}
        K (int): number of prototypes per class
        output_dir (str): directory to save prototypes
        seed (int): random seed
    """
    set_seed(seed)
    os.makedirs(output_dir, exist_ok=True)

    for cls in [0, 1]:
        X_cls = X[y == cls].astype(np.float64)

        if len(X_cls) < K:
            raise ValueError(
                f"Not enough samples for class {cls}: "
                f"{len(X_cls)} < K={K}"
            )

        kmeans = KMeans(
            n_clusters=K,
            random_state=seed,

```

```

        n_init=10
    )
    kmeans.fit(X_cls)

    centers = kmeans.cluster_centers_

    for i in range(K):
        proto = to_quantum_state(centers[i])
        path = os.path.join(output_dir, f"class{cls}_proto{i}.npz")
        np.save(path, proto)

# -----
# Script mode (EXPERIMENTS ONLY)
# -----
if __name__ == "__main__":
    from src.utils.paths import load_paths

    # Reproducibility
    set_seed(42)

    # Load paths
    _, PATHS = load_paths()
    EMBED_DIR = PATHS["embeddings"]
    PROTO_BASE = PATHS["class_prototypes"]

    os.makedirs(EMBED_DIR, exist_ok=True)
    os.makedirs(PROTO_BASE, exist_ok=True)

    # Load embeddings (TRAIN ONLY)
    X = np.load(os.path.join(EMBED_DIR, "val_embeddings.npz"))
    y = np.load(os.path.join(EMBED_DIR, "val_labels.npz"))
    train_idx = np.load(os.path.join(EMBED_DIR, "split_train_idx.npz"))

    X_train = X[train_idx]
    y_train = y[train_idx]

    print("Loaded train embeddings:", X_train.shape)

    K_VALUES = PATHS["class_count"]["K_values"]

    for K in K_VALUES:
        print(f"\n=== Computing prototypes for K={K} ===")
        CLASS_DIR = os.path.join(PROTO_BASE, f"K{K}")
        generate_prototypes(
            X=X_train,
            y=y_train,
            K=K,
            output_dir=CLASS_DIR,
            seed=42
        )
        print(f"Saved prototypes to {CLASS_DIR}")

```

File: src/IQL/learning/init.py

File: src/IQL/backends/base.py

```
from abc import ABC, abstractmethod

class InterferenceBackend(ABC):
    """
    Abstract interface for computing interference scores.
    """

    @abstractmethod
    def score(self, chi, psi) -> float:
        """
        Return  $\text{Re}\langle \chi | \psi \rangle$  as a real scalar.
        """
        pass
```

File: src/IQL/backends/prime_b.py

```
import numpy as np
from qiskit import QuantumCircuit
from qiskit.quantum_info import Statevector, Pauli
from qiskit.circuit.library import StatePreparation

from .base import InterferenceBackend

class PrimeBBackend(InterferenceBackend):
    """
    PrimeB (ISDO-B') Backend
    -----

    Observable-engineered, decision-sufficient implementation of ISDO.

    Computes:
         $S(\psi; \chi) = \langle \psi | U_{\chi}^{\dagger} Z^{\{\otimes n\}} U_{\chi} | \psi \rangle$ 

    Properties:
    - No ancilla qubit
    - No controlled unitaries
    -  $\chi$  appears only as a basis rotation
    - Fixed, hardware-native observable
```

- Preserves sign + ordering (not exact inner product)

Intended role:

- Fast inference
- NISQ-friendly deployment backend

\"\"\"

@staticmethod

def _statevector_to_unitary(state: np.ndarray) -> np.ndarray:

\"\"\"

Construct a unitary U such that:

$$U |0\dots 0\rangle = |\text{state}\rangle$$

Uses Gram-Schmidt completion.

\"\"\"

state = np.asarray(state, dtype=np.complex128)

state = state / np.linalg.norm(state)

dim = len(state)

U = np.zeros((dim, dim), dtype=np.complex128)

U[:, 0] = state

for i in range(1, dim):

 v = np.zeros(dim, dtype=np.complex128)

 v[i] = 1.0

 for j in range(i):

 v -= np.vdot(U[:, j], v) * U[:, j]

 norm = np.linalg.norm(v)

 if norm < 1e-12:

 v = np.random.randn(dim) + 1j * np.random.randn(dim)

 for j in range(i):

 v -= np.vdot(U[:, j], v) * U[:, j]

 v /= np.linalg.norm(v)

 else:

 v /= norm

 U[:, i] = v

return U

def score(self, chi: np.ndarray, psi: np.ndarray) -> float:

\"\"\"

Compute PrimeB interference score.

Args:

 chi : np.ndarray

 Class memory state $|\chi\rangle$

 psi : np.ndarray

 Input state $|\psi\rangle$

Returns:

 float

```

        Decision-sufficient interference score
    """
    chi = np.asarray(chi, dtype=np.complex128)
    psi = np.asarray(psi, dtype=np.complex128)

    chi /= np.linalg.norm(chi)
    psi /= np.linalg.norm(psi)

    dim = len(psi)
    n = int(np.log2(dim))
    if 2 ** n != dim:
        raise ValueError("State dimension must be a power of 2")

    # Build circuit
    qc = QuantumCircuit(n)

    # Prepare  $|\psi\rangle$ 
    qc.append(StatePreparation(psi), range(n))

    # Apply  $U_\chi$ 
    U_chi = self._statevector_to_unitary(chi)
    qc.unitary(U_chi, range(n), label="U_chi")


    # Evaluate  $\langle Z^{\otimes n} \rangle$ 
    sv = Statevector.from_instruction(qc)
    observable = Pauli("Z"+"I" * (n-1))

    return float(sv.expectation_value(observable).real)
"""
pass

```

File: src/IQL/backends/hadamard.py

```

import numpy as np
from qiskit import QuantumCircuit
from qiskit.quantum_info import Statevector, Pauli
from qiskit.circuit.library import StatePreparation #  Correct import
from .base import InterferenceBackend

# If you also want the conceptual/oracle version:
class HadamardBackend(InterferenceBackend):
    """
    CONCEPTUAL Hadamard-test using oracle state preparation.

    WARNING: This uses non-unitary StatePreparation and is NOT
    physically realizable. Use only for conceptual understanding.
    For actual implementation, use TransitionInterferenceBackend.

    Computes  $\text{Re}\langle \chi | \psi \rangle$  in oracle model.
    """

```

```

def score(self, chi, psi) -> float:
    chi = np.asarray(chi, dtype=np.complex128)
    psi = np.asarray(psi, dtype=np.complex128)

    # Normalize
    chi = chi / np.linalg.norm(chi)
    psi = psi / np.linalg.norm(psi)

    assert chi.shape == psi.shape
    n = int(np.log2(len(psi)))
    assert 2**n == len(psi)

    qc = QuantumCircuit(1 + n)
    anc = 0
    data = list(range(1, 1 + n))

    # Hadamard on ancilla
    qc.h(anc)

    # Controlled state preparation (ORACLE ASSUMPTION)
    # When anc=0: prepare |psi>
    state_prep_psi = StatePreparation(psi)
    qc.append(state_prep_psi.control(1), [anc] + data)

    # Flip ancilla
    qc.x(anc)

    # When anc=1 (after flip, so anc=0): prepare |chi>
    state_prep_chi = StatePreparation(chi)
    qc.append(state_prep_chi.control(1), [anc] + data)

    # Flip back
    qc.x(anc)

    # Final Hadamard
    qc.h(anc)

    # Get statevector and measure Z on ancilla
    sv = Statevector.from_instruction(qc)
    z_exp = sv.expectation_value(Pauli('Z'), [anc]).real

    return float(z_exp)

```

File: src/IQL/backends/transition.py

```

import numpy as np
from qiskit import QuantumCircuit
from qiskit.quantum_info import Statevector, Pauli
from qiskit.circuit.library import UnitaryGate, StatePreparation # ✓
Correct import
from .base import InterferenceBackend

```



```

class TransitionBackend(InterferenceBackend):
    """
    CORRECT physical Hadamard-test using transition unitary.

    This is the physically realizable ISDO implementation.
    Computes  $\text{Re}\langle \chi | \psi \rangle$  using  $U_{\chi\psi} = U_{\chi} @ U_{\psi}^{\dagger}$ 

    This should be used for all hardware experiments and claims.
    """

    @staticmethod
    def _statevector_to_unitary(vec):
        """Build unitary that prepares vec from  $|0\dots 0\rangle$ """
        vec = np.asarray(vec, dtype=np.complex128)
        vec = vec / np.linalg.norm(vec)
        dim = len(vec)

        U = np.zeros((dim, dim), dtype=complex)
        U[:, 0] = vec

        # Gram-Schmidt to complete the unitary
        for i in range(1, dim):
            v = np.zeros(dim, dtype=complex)
            v[i] = 1.0

            for j in range(i):
                v -= np.vdot(U[:, j], v) * U[:, j]

            v_norm = np.linalg.norm(v)
            if v_norm > 1e-10:
                U[:, i] = v / v_norm
            else:
                v = np.random.randn(dim) + 1j * np.random.randn(dim)
                for j in range(i):
                    v -= np.vdot(U[:, j], v) * U[:, j]
                U[:, i] = v / np.linalg.norm(v)

        return U

    @staticmethod
    def _build_transition_unitary(psi, chi):
        """Build  $U_{\chi\psi} = U_{\chi} @ U_{\psi}^{\dagger}$ """
        U_psi = TransitionBackend._statevector_to_unitary(psi)
        U_chi = TransitionBackend._statevector_to_unitary(chi)

        # Transition unitary
        U_chi_psi = U_chi @ U_psi.conj().T

        return UnitaryGate(U_chi_psi)

    def score(self, chi, psi) -> float:
        chi = np.asarray(chi, dtype=np.complex128)

```

```

psi = np.asarray(psi, dtype=np.complex128)

# Normalize
chi = chi / np.linalg.norm(chi)
psi = psi / np.linalg.norm(psi)

assert chi.shape == psi.shape
n = int(np.log2(len(psi)))
assert 2**n == len(psi)

qc = QuantumCircuit(1 + n)
anc = 0
data = list(range(1, 1 + n))

# Prepare |psi> on data qubits
qc.append(StatePreparation(psi), data)

# Hadamard on ancilla
qc.h(anc)

# Controlled transition unitary
U_chi_psi = self._build_transition_unitary(psi, chi)
qc.append(U_chi_psi.control(1), [anc] + data)

# Final Hadamard
qc.h(anc)

# Get statevector and measure Z on ancilla
sv = Statevector.from_instruction(qc)
z_exp = sv.expectation_value(Pauli('Z'), [anc]).real

return float(z_exp)

```

File: src/IQL/backends/exact.py

```

import numpy as np
from .base import InterferenceBackend

class ExactBackend(InterferenceBackend):
    """
    Numpy-based interference backend.
    This reproduces existing behavior exactly.
    """

    def score(self, chi, psi) -> float:
        return float(np.real(np.vdot(chi, psi)))

```

File: src/IQL/backends/init.py

File: src/IQL/regimes/regime3c_adaptive.py

```
import numpy as np
from collections import deque
from src.IQL.learning.update import update
from src.IQL.backends.exact import ExactBackend
import pickle

class AdaptiveMemory:
    """
    Regime 3-C: Dynamic Memory Growth with Percentile-based  $\tau$ 
    """

    def __init__(
        self,
        memory_bank,
        eta=0.1,
        percentile=5,
        tau_abs = -0.4,
        margin_window=500,
        backend=ExactBackend()
    ):
        self.memory_bank = memory_bank
        self.eta = eta
        self.percentile = percentile
        self.tau_abs = tau_abs
        self.backend = backend

        # store recent margins
        self.margins = deque(maxlen=margin_window)

        self.num_updates = 0
        self.num_spawns = 0

        self.history = {
            "margin": [],
            "spawned": [],
            "num_memories": [],
        }

    def aggregated_score(self, psi):
        scores = self.memory_bank.scores(psi)
        return sum(scores) / len(scores)

    def step(self, psi, y):
        S = self.aggregated_score(psi)
        margin = y * S
```

```

# collect negative margins only
neg_margins = [m for m in self.margins if m < 0]

spawned = False

# compute percentile only if we have enough negative history
if len(neg_margins) >= 20:
    tau = np.percentile(neg_margins, self.percentile)

    if margin < tau:
        # 🔥 spawn new memory
        chi_new = y * psi
        chi_new = chi_new / np.linalg.norm(chi_new)
        self.memory_bank.add_memory(chi_new, self.backend)
        self.num_spawns += 1
        spawned = True

# otherwise, normal Regime-2 update on winner
if not spawned:
    idx, _ = self.memory_bank.winner(psi)
    cs = self.memory_bank.class_states[idx]

    chi_new, updated = update(
        cs.vector, psi, y, self.eta, self.backend
    )

    if updated:
        cs.vector = chi_new
        self.num_updates += 1

# logging
self.margins.append(margin)
self.history["margin"].append(margin)
self.history["spawned"].append(spawned)

self.history["num_memories"].append(len(self.memory_bank.class_states))

return margin, spawned

def memory_size(self):
    return len(self.memory_bank.class_states)

def fit(self, X, y):
    for psi, y in zip(X, y):
        self.step(psi, y)

def predict_one(self, X):
    _, score = self.memory_bank.winner(X)
    return 1 if score >= 0 else -1

def predict(self, X):
    return [self.predict_one(x) for x in X]

def save(self, path):

```

```

"""
Save trained memory + training history.
"""
payload = {
    "memory_bank": self.memory_bank,
    "eta": self.eta,
    "percentile": self.percentile,
    "tau_abs": self.tau_abs,
    "margins": list(self.margins),
    "num_updates": self.num_updates,
    "num_spawns": self.num_spawns,
    "history": self.history,
    "backend": self.backend,
}

with open(path, "wb") as f:
    pickle.dump(payload, f)

@classmethod
def load(cls, path):
    """
    Load a previously trained Regime-3C model.
    """
    with open(path, "rb") as f:
        payload = pickle.load(f)

    obj = cls(
        memory_bank=payload["memory_bank"],
        eta=payload["eta"],
        percentile=payload["percentile"],
        tau_abs=payload["tau_abs"],
        margin_window=len(payload["margins"]),
        backend=payload["backend"],
    )

    # restore training state
    from collections import deque
    obj.margins = deque(payload["margins"],
maxlen=len(payload["margins"]))
    obj.num_updates = payload["num_updates"]
    obj.num_spawns = payload["num_spawns"]
    obj.history = payload["history"]

    return obj

```

File: src/IQL/regimes/regime3a_wta.py

```

from src.IQL.learning.update import update
from src.IQL.backends.exact import ExactBackend
import pickle

```

```
class WinnerTakeAll:
    """
    Regime 3-A: Winner-Takes-All IQC
    Only the winning memory is updated.
    """

    def __init__(self, memory_bank, eta, backend = ExactBackend()):
        self.memory_bank = memory_bank
        self.eta = eta
        self.backend = backend
        self.num_updates = 0

        self.history = {
            "winner_idx": [],
            "scores": [],
            "updates": [],
        }

    def step(self, psi, y):
        idx, score = self.memory_bank.winner(psi)
        cs = self.memory_bank.class_states[idx]

        chi_new, updated = update(
            cs.vector, psi, y, self.eta, self.backend
        )

        if updated:
            cs.vector = chi_new
            self.num_updates += 1

        y_hat = 1 if score >= 0 else -1

        # logging
        self.history["winner_idx"].append(idx)
        self.history["scores"].append(score)
        self.history["updates"].append(updated)

        return y_hat, idx, updated

    def fit(self, X, y):
        correct = 0
        for x, y in zip(X, y):
            y_hat, _, _ = self.step(x, y)
            if y_hat == y:
                correct += 1
        return correct / len(X)

    def predict_one(self, X):
        _, score = self.memory_bank.winner(X)
        return 1 if score >= 0 else -1

    def predict(self, X):
        return [self.predict_one(x) for x in X]
```

```

def save(self, path):
    """
    Save trained memory bank and history.
    """
    payload = {
        "memory_bank": self.memory_bank,
        "eta": self.eta,
        "num_updates": self.num_updates,
        "history": self.history,
        "backend": self.backend,
    }

    with open(path, "wb") as f:
        pickle.dump(payload, f)

@classmethod
def load(cls, path):
    """
    Load a trained Winner-Take-All model.
    """
    with open(path, "rb") as f:
        payload = pickle.load(f)

    obj = cls(
        memory_bank=payload["memory_bank"],
        eta=payload["eta"],
        backend=payload["backend"],
    )

    # restore training statistics
    obj.num_updates = payload["num_updates"]
    obj.history = payload["history"]

    return obj

```

File: src/IQL/regimes/regime2_online.py

```

import numpy as np
from src.IQL.learning.update import update
import pickle

class OnlinePerceptron:
    """
    Online Interference Quantum Classifier (Regime 2)

    Fixed circuit.
    Trainable object: |chi>
    """

    def __init__(self, class_state, eta: float):

```

```
        self.class_state = class_state
        self.eta = eta
        # logs
        self.num_updates = 0
        self.history = {
            "scores": [],
            "margins": [],
            "updates": [],
        }

    def step(self, psi: np.ndarray, y: int):
        """
        Process a single training example.
        """
        s = self.class_state.score(psi)
        margin = y * s
        y_hat = 1 if s >= 0 else -1

        chi_new, updated = update(
            self.class_state.vector, psi, y, self.eta,
            self.class_state.backend
        )

        if updated:
            self.class_state.vector = chi_new
            self.num_updates += 1

        # logging
        self.history["scores"].append(s)
        self.history["margins"].append(margin)
        self.history["updates"].append(updated)

        return y_hat, s, updated

    def fit(self, X, y):
        """
        Single-pass online training.
        dataset: iterable of (psi, y)
        """
        correct = 0

        for i in range(len(X)):
            y_hat, _, _ = self.step(X[i], y[i])
            if y_hat == y[i]:
                correct += 1

        accuracy = correct / len(X)
        return accuracy

    def predict_one(self, X):
        s = self.class_state.score(X)
        return 1 if s >= 0 else -1

    def predict(self, X):
```



```

        return [self.predict_one(x) for x in X]

def save(self, path):
    """
    Save trained perceptron state and history.
    """
    payload = {
        "class_state": self.class_state,
        "eta": self.eta,
        "num_updates": self.num_updates,
        "history": self.history,
        "backend": self.class_state.backend,
    }

    with open(path, "wb") as f:
        pickle.dump(payload, f)

@classmethod
def load(cls, path):
    """
    Load a trained perceptron model.
    """
    with open(path, "rb") as f:
        payload = pickle.load(f)

    obj = cls(
        class_state=payload["class_state"],
        eta=payload["eta"],
    )

    # restore training statistics
    obj.num_updates = payload["num_updates"]
    obj.history = payload["history"]

    return obj

```

File: src/IQL/regimes/init.py

File: src/IQL/inference/weighted_vote_classifier.py

```

class WeightedVoteClassifier:
    def __init__(self, memory_bank, weights=None):
        self.memory_bank = memory_bank
        self.M = len(memory_bank.class_states)

        if weights is None:
            self.weights = [1.0 / self.M] * self.M

```

```

        else:
            s = sum(weights)
            self.weights = [w / s for w in weights]

    def score(self, psi):
        scores = self.memory_bank.scores(psi)
        return sum(w * s for w, s in zip(self.weights, scores))

    def predict(self, psi):
        return 1 if self.score(psi) >= 0 else -1

    def save(self, path):
        import pickle
        payload = {
            "memory_bank": self.memory_bank,
            "weights": self.weights,
        }
        with open(path, "wb") as f:
            pickle.dump(payload, f)

    @classmethod
    def load(cls, path):
        import pickle
        with open(path, "rb") as f:
            payload = pickle.load(f)
        obj = cls(payload["memory_bank"], payload["weights"])
        return obj

```

File: src/IQL/inference/init.py

File: src/IQL/encoding/embedding_to_state.py

```

import numpy as np

def embedding_to_state(x: np.ndarray) -> np.ndarray:
    """
    Maps a real embedding  $x \in \mathbb{R}^d$  to a quantum state  $|\psi\rangle$ .
    This is a purely geometric normalization.
    """
    x = x.astype(np.complex128)
    norm = np.linalg.norm(x)
    if norm == 0:
        raise ValueError("Zero embedding encountered")
    return x / norm

```

File: src/IQL/encoding/init.py

File: src/IQL/baselines/static_isdo_classifier.py

```
import os
import numpy as np
from tqdm import tqdm
from src.IQL.backends.exact import ExactBackend

class StaticISDOClassifier:
    def __init__(self, proto_dir, K):
        self.proto_dir = proto_dir
        self.K = K
        self.exact = ExactBackend()
        self.prototypes = {
            0: [np.load(os.path.join(proto_dir,
f"K{K}/class0_proto{i}.npy")) for i in range(K)],
            1: [np.load(os.path.join(proto_dir,
f"K{K}/class1_proto{i}.npy")) for i in range(K)],
        }

    def predict_one(self, psi):
        #A0 = sum(np.vdot(p, psi) for p in self.prototypes[0])
        #A1 = sum(np.vdot(p, psi) for p in self.prototypes[1])
        #return 1 if np.real(A0 - A1) < 0 else 0
        chi = sum(self.prototypes[0]) - sum(self.prototypes[1])
        chi /= np.linalg.norm(chi)
        return 1 if self.exact.score(chi, psi) < 0 else 0

    def predict(self, X):
        return np.array([self.predict_one(x) for x in tqdm(X, desc="ISDO
Prediction", leave=False)])
```

File: src/utis/common_backup.py

```
import numpy as np
from qiskit import QuantumCircuit
from qiskit.circuit.library import StatePreparation, UnitaryGate

def load_statevector(vec):
    """
    Create a Qiskit StatePreparation gate from a normalized vector.
```

```

NOTE: This is for CONCEPTUAL/ORACLE model only (Circuit A)
For physical implementation, use build_transition_unitary instead
"""
vec = np.asarray(vec, dtype=np.complex128)
norm = np.linalg.norm(vec)
if not np.isclose(norm, 1.0, atol=1e-12):
    raise ValueError("Statevector must be normalized")
return StatePreparation(vec)

def statevector_to_unitary(psi):
    """
    Convert a statevector to a unitary operator that creates it from
    |0...0>
    Uses Gram-Schmidt to complete the unitary matrix.

    This creates U_psi such that U_psi |0...0> = |psi>

    Used for building transition unitaries in Circuit B'.
    """
    psi = np.asarray(psi, dtype=np.complex128)
    dim = len(psi)

    # Normalize
    psi = psi / np.linalg.norm(psi)

    # Create unitary matrix where first column is psi
    U = np.zeros((dim, dim), dtype=complex)
    U[:, 0] = psi

    # Complete to full unitary using Gram-Schmidt orthogonalization
    for i in range(1, dim):
        # Start with standard basis vector
        v = np.zeros(dim, dtype=complex)
        v[i] = 1.0

        # Orthogonalize against all previous columns
        for j in range(i):
            v -= np.vdot(U[:, j], v) * U[:, j]

        # Normalize and store
        v_norm = np.linalg.norm(v)
        if v_norm > 1e-10:
            U[:, i] = v / v_norm
        else:
            # Use random vector if degenerate
            v = np.random.randn(dim) + 1j * np.random.randn(dim)
            for j in range(i):
                v -= np.vdot(U[:, j], v) * U[:, j]
            U[:, i] = v / np.linalg.norm(v)

    return U

```

```

def build_transition_unitary(psi, chi):
    """
    Build the transition unitary  $U_{\chi\psi} = U_{\chi} @ U_{\psi}^{\dagger}$ 

    This is the KEY OPERATION for physically realizable ISDO (Circuit B').

    This unitary satisfies:  $U_{\chi\psi} |\psi\rangle = |\chi\rangle$ 

    Args:
        psi: Source statevector
        chi: Target statevector

    Returns:
        UnitaryGate that implements the transition
    """
    # Build unitaries that prepare each state from  $|0\dots0\rangle$ 
    U_psi = statevector_to_unitary(psi)
    U_chi = statevector_to_unitary(chi)

    # Transition unitary:  $U_{\chi} @ U_{\psi}^{\dagger}$ 
    U_chi_psi = U_chi @ U_psi.conj().T

    # Verify it works
    psi_normalized = np.asarray(psi, dtype=np.complex128)
    psi_normalized = psi_normalized / np.linalg.norm(psi_normalized)
    chi_normalized = np.asarray(chi, dtype=np.complex128)
    chi_normalized = chi_normalized / np.linalg.norm(chi_normalized)

    result = U_chi_psi @ psi_normalized
    if not np.allclose(result, chi_normalized, atol=1e-10):
        raise ValueError("Transition unitary does not correctly map  $|\psi\rangle$  to  $|\chi\rangle$ ")

    return UnitaryGate(U_chi_psi)

def build_chi_state(class0_protos, class1_protos):
    """
    Build  $|\chi\rangle = \sum_k |\phi_k^0\rangle - \sum_k |\phi_k^1\rangle$ , normalized

    This constructs the reference state for ISDO classification.
    """
    chi = np.zeros_like(class0_protos[0], dtype=np.float64)

    for p in class0_protos:
        chi += p
    for p in class1_protos:
        chi -= p

    chi /= np.linalg.norm(chi)
    return chi

```

File: src/utis/common.py

```

import numpy as np
from qiskit import QuantumCircuit
from qiskit.circuit.library import StatePreparation, UnitaryGate

def load_statevector(vec):
    """
    Create a Qiskit StatePreparation gate from a normalized vector.

    NOTE: This is for CONCEPTUAL/ORACLE model only (Circuit A)
    For physical implementation, use build_transition_unitary instead
    """
    vec = np.asarray(vec, dtype=np.complex128)
    norm = np.linalg.norm(vec)
    if not np.isclose(norm, 1.0, atol=1e-12):
        raise ValueError("Statevector must be normalized")
    return StatePreparation(vec)

def statevector_to_unitary(psi):
    """
    Convert a statevector to a unitary operator using Householder
    efficiency.
    Construct a Householder reflection U such that  $U |e1\rangle = |\psi\rangle$ 
    where  $e1 = [1, 0, \dots, 0]^T$ .

    This is  $O(D^2)$  to build the matrix, compared to  $O(D^3)$  for Gram-
    Schmidt.
    """
    psi = np.asarray(psi, dtype=np.complex128)
    norm = np.linalg.norm(psi)
    if norm > 1e-15:
        psi = psi / norm

    dim = len(psi)
    e1 = np.zeros(dim, dtype=np.complex128)
    e1[0] = 1.0

    # Adjust phase to avoid numerical instability (choose phase to make w
    large)
    # We want to map phase * e1 to psi where phase has same angle as psi[0]
    # This ensures w = phase * e1 - psi is stable.
    angle = np.angle(psi[0]) if np.abs(psi[0]) > 1e-10 else 0.0
    phase = np.exp(1j * angle)

    target = phase * e1
    w = target - psi
    w_norm = np.linalg.norm(w)

    if w_norm < 1e-12:

```

```

    # psi is already phase * e1, so just return identity * phase
    return np.eye(dim, dtype=np.complex128) * phase

v = w / w_norm
# R = I - 2vv* maps target (phase * e1) to psi
# R * phase * e1 = psi => R * e1 = psi * phase*
# To get U * e1 = psi, we need U = R * phase
H = (np.eye(dim, dtype=np.complex128) - 2.0 * np.outer(v, v.conj())) *
phase
return H

def build_transition_unitary(psi, chi):
    """
    Build the transition unitary U_chi_psi = U_chi @ U_psi^dagger

    This is the KEY OPERATION for physically realizable ISDO (Circuit B').

    This unitary satisfies: U_chi_psi |psi> = |chi>

    Args:
        psi: Source statevector
        chi: Target statevector

    Returns:
        UnitaryGate that implements the transition
    """
    # Build unitaries that prepare each state from |0...0>
    U_psi = statevector_to_unitary(psi)
    U_chi = statevector_to_unitary(chi)

    # Transition unitary: U_chi @ U_psi^dagger
    U_chi_psi = U_chi @ U_psi.conj().T

    # Verify it works
    psi_normalized = np.asarray(psi, dtype=np.complex128)
    psi_normalized = psi_normalized / np.linalg.norm(psi_normalized)
    chi_normalized = np.asarray(chi, dtype=np.complex128)
    chi_normalized = chi_normalized / np.linalg.norm(chi_normalized)

    result = U_chi_psi @ psi_normalized
    if not np.allclose(result, chi_normalized, atol=1e-10):
        raise ValueError("Transition unitary does not correctly map |psi>
to |chi>")

    return UnitaryGate(U_chi_psi)

def build_chi_state(class0_protos, class1_protos):
    """
    Build |chi> = sum_k |phi_k^0> - sum_k |phi_k^1>, normalized

    This constructs the reference state for ISDO classification.

```

```
"""
chi = np.zeros_like(class0_protos[0], dtype=np.float64)

for p in class0_protos:
    chi += p
for p in class1_protos:
    chi -= p

chi /= np.linalg.norm(chi)
return chi
```

File: src/utlis/paths.py

```
import yaml
import os

def load_paths(config_path="configs/paths.yaml"):
    with open(config_path, "r") as f:
        cfg = yaml.safe_load(f)

    base_root = cfg["base_root"]
    paths = {
        k: os.path.join(base_root, v)
        for k, v in cfg["paths"].items()
    }
    paths["class_count"] = cfg["class_count"]
    return base_root, paths
```

File: src/utlis/seed.py

```
import random
import numpy as np
import torch
import os

def set_seed(seed: int = 42):
    # Python
    random.seed(seed)

    # NumPy
    np.random.seed(seed)

    # PyTorch
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

    # cuDNN (important)
```



```
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# Extra safety (hash-based ops)
os.environ["PYTHONHASHSEED"] = str(seed)

print(f"🌱 Global seed set to {seed}")
```

File: src/utils/label_utils.py

```
# src/utils/label_utils.py
"""
Unified label conversion utilities for quantum classifier.

Standard convention:
- Binary: {0, 1} for storage and classical models
- Polar: {-1, +1} for quantum interference calculations
"""
import numpy as np

def binary_to_polar(labels):
    """
    Convert binary labels {0, 1} to polar {-1, +1}.

    Args:
        labels: array-like with values in {0, 1}

    Returns:
        numpy array with values in {-1, +1}
    """
    labels = np.asarray(labels)
    return 2 * labels - 1

def polar_to_binary(labels):
    """
    Convert polar labels {-1, +1} to binary {0, 1}.

    Args:
        labels: array-like with values in {-1, +1}

    Returns:
        numpy array with values in {0, 1}
    """
    labels = np.asarray(labels)
    return (labels + 1) // 2

def ensure_polar(labels):
    """
    Ensure labels are in polar format {-1, +1}.
    Automatically detects format and converts if needed.
```

```

"""
labels = np.asarray(labels)
unique_vals = np.unique(labels)

if set(unique_vals).issubset({0, 1}):
    return binary_to_polar(labels)
elif set(unique_vals).issubset({-1, 1}):
    return labels
else:
    raise ValueError(f"Labels must be binary {{0,1}} or polar
{{-1,+1}}. Got: {unique_vals}")

def ensure_binary(labels):
    """
    Ensure labels are in binary format {0, 1}.
    Automatically detects format and converts if needed.
    """
    labels = np.asarray(labels)
    unique_vals = np.unique(labels)

    if set(unique_vals).issubset({0, 1}):
        return labels
    elif set(unique_vals).issubset({-1, 1}):
        return polar_to_binary(labels)
    else:
        raise ValueError(f"Labels must be binary {{0,1}} or polar
{{-1,+1}}. Got: {unique_vals}")

```

File: src/utils/init.py

File: src/data/pcam_loader.py

```

from torchvision import datasets, transforms
from torch.utils.data import DataLoader

def get_pcam_dataset(data_dir='/home/tarakesh/Work/Repo/measurement-free-
quantum-classifier/dataset', split='train', download=True, transform=None):
    """
    Wrapper for torchvision's built-in PCAM dataset.
    Automatically handles downloading and formatting.
    """
    if transform is None:
        # Default transformation for the hybrid model
        transform = transforms.Compose([
            transforms.ToTensor(), # Scales [0, 255] to [0.0, 1.0] and HWC
to CHW
        ])

```

```
dataset = datasets.PCAM(
    root=data_dir,
    split=split,
    download=download,
    transform=transform
)
return dataset

if __name__ == "__main__":
    print("PCAM Loader (using torchvision) initialized.")
```

File: src/data/transforms.py

```
from torchvision import transforms

def get_train_transforms():
    """
    Minimal, label-preserving augmentations for CNN training only.
    """
    return transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomVerticalFlip(),
        transforms.ColorJitter(
            brightness=0.1,
            contrast=0.1,
            saturation=0.05,
        ),
        transforms.ToTensor(),
        transforms.Normalize(
            mean=[0.5, 0.5, 0.5],
            std=[0.5, 0.5, 0.5],
        ),
    ])

def get_eval_transforms():
    """
    Deterministic transforms for validation, testing, and embedding
    extraction.
    """
    return transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(
            mean=[0.5, 0.5, 0.5],
            std=[0.5, 0.5, 0.5],
        ),
    ])
```

File: src/data/init.py

File: src/quantum/compute_qsvm_kernel.py

```
import os
import json
import numpy as np
from tqdm import tqdm

from qiskit_aer.primitives import SamplerV2
from qiskit.circuit.library import ZZFeatureMap
from qiskit_machine_learning.kernels import FidelityQuantumKernel
from qiskit_algorithms.state_fidelities import ComputeUncompute

from src.utils.paths import load_paths
from src.utils.seed import set_seed

# -----
# Reproducibility
# -----
set_seed(42)

# -----
# Load paths and data
# -----
BASE_ROOT, PATHS = load_paths()

EMBED_DIR = PATHS["embeddings"]
OUT_DIR = os.path.join(BASE_ROOT, "results", "qsvm_cache")
os.makedirs(OUT_DIR, exist_ok=True)

X = np.load(os.path.join(EMBED_DIR, "val_embeddings.npy"))
y = np.load(os.path.join(EMBED_DIR, "val_labels.npy"))

train_idx = np.load(os.path.join(EMBED_DIR, "split_train_idx.npy"))
test_idx = np.load(os.path.join(EMBED_DIR, "split_test_idx.npy"))

X_train = X[train_idx]
y_train = y[train_idx]

X_test = X[test_idx]
y_test = y[test_idx]

# -----
# SUBSAMPLING for Baseline Efficiency
# -----
```

```

# Limiting to 500 samples because  $O(N^2)$  kernel computation
# for 3500 samples would take ~17 hours on GPU.
MAX_TRAIN = 500000
MAX_TEST = 200000

if len(X_train) > MAX_TRAIN:
    print(f"Subsampling train set from {len(X_train)} to {MAX_TRAIN}...")
    rng = np.random.default_rng(42)
    indices = rng.choice(len(X_train), MAX_TRAIN, replace=False)
    X_train = X_train[indices]
    y_train = y_train[indices]

if len(X_test) > MAX_TEST:
    print(f"Subsampling test set from {len(X_test)} to {MAX_TEST}...")
    rng = np.random.default_rng(42)
    indices = rng.choice(len(X_test), MAX_TEST, replace=False)
    X_test = X_test[indices]
    y_test = y_test[indices]

# -----
# Normalize embeddings
# -----
X_train = X_train / np.linalg.norm(X_train, axis=1, keepdims=True)
X_test = X_test / np.linalg.norm(X_test, axis=1, keepdims=True)

# Infer number of qubits
dim = X_train.shape[1]
num_qubits = int(np.log2(dim))
assert 2 ** num_qubits == dim, "Embedding dimension must be  $2^n$ "

# -----
# Define FIXED quantum feature map
# -----
feature_map = ZZFeatureMap(
    feature_dimension=num_qubits,
    reps=1,
    entanglement="linear"
)

# -----
# GPU Accelerated Backend (Aer SamplerV2)
# -----
sampler = SamplerV2(
    options={"backend_options": {"method": "statevector", "device": "GPU"}}
)
fidelity = ComputeUncompute(sampler=sampler)

quantum_kernel = FidelityQuantumKernel(
    feature_map=feature_map,
    fidelity=fidelity
)

# -----
# Compute and save TRAIN kernel

```

```

# -----
print(f"Computing QSVM TRAIN kernel ({len(X_train)}x{len(X_train)}).")
K_train = quantum_kernel.evaluate(X_train, X_train)
np.save(os.path.join(OUT_DIR, "qsvm_kernel_train.npy"), K_train)

# -----
# Compute and save TEST kernel
# -----
print(f"Computing QSVM TEST kernel ({len(X_test)}x{len(X_train)}).")
K_test = quantum_kernel.evaluate(X_test, X_train)
np.save(os.path.join(OUT_DIR, "qsvm_kernel_test.npy"), K_test)

# -----
# Save Labels for verification
# -----
np.save(os.path.join(OUT_DIR, "y_train_sub.npy"), y_train)
np.save(os.path.join(OUT_DIR, "y_test_sub.npy"), y_test)

# -----
# Save metadata
# -----
meta = {
    "model": "QSVM",
    "num_qubits": num_qubits,
    "num_train": int(X_train.shape[0]),
    "num_test": int(X_test.shape[0]),
    "embedding_dimension": int(dim),
    "subsampling": True
}

with open(os.path.join(OUT_DIR, "qsvm_kernel_meta.json"), "w") as f:
    json.dump(meta, f, indent=2)

print("QSVM kernel computation complete.")

```

File: src/quantum/init.py

File: src/training/test_adaptive_iqc.py

```

import os
import numpy as np
from sklearn.metrics import accuracy_score

from src.IQL.models.adaptive_iqc import AdaptiveIQC
from src.utils.paths import load_paths

```

```

def main():
    # -----
    # Load data
    # -----
    _, PATHS = load_paths()
    EMBED_DIR = PATHS["embeddings"]

    X = np.load(os.path.join(EMBED_DIR, "val_embeddings.npy"))
    y = np.load(os.path.join(EMBED_DIR, "val_labels.npy")) # ±1

    train_idx = np.load(os.path.join(EMBED_DIR, "split_train_idx.npy"))
    test_idx = np.load(os.path.join(EMBED_DIR, "split_test_idx.npy"))

    X_train, y_train = X[train_idx], y[train_idx]
    X_test, y_test = X[test_idx], y[test_idx]

    # quantum-safe normalization
    X_train /= np.linalg.norm(X_train, axis=1, keepdims=True)
    X_test /= np.linalg.norm(X_test, axis=1, keepdims=True)

    # -----
    # Train AdaptiveIQC
    # -----
    model = AdaptiveIQC(
        K_init=3,
        eta=0.1,
        percentile=5,
        consolidate=True,
    )

    model.fit(X_train, y_train)

    # -----
    # Evaluate
    # -----
    y_pred = model.predict(X_test)
    acc = accuracy_score(y_test, y_pred)

    print("✅ AdaptiveIQC Test Accuracy:", round(acc, 4))

if __name__ == "__main__":
    main()

```

File: src/training/verify_consistency.py

```

import numpy as np
from src.IQL.learning.class_state import ClassState
from src.IQL.learning.memory_bank import MemoryBank

```

```

from src.IQL.backends.exact import ExactBackend
from src.IQL.regimes.regime2_online import OnlinePerceptron
from src.IQL.regimes.regime3a_wta import WinnerTakeAll
from src.IQL.regimes.regime3c_adaptive import AdaptiveMemory

def test_consistency():
    print("Running consistency tests...")

    # 1. Backend
    backend = ExactBackend()

    # 2. ClassState
    vec = np.array([1, 0, 0, 0], dtype=np.complex128)
    cs = ClassState(vec, backend)
    print("ClassState initialized.")

    psi = np.array([1, 0, 0, 0], dtype=np.complex128)
    score = cs.score(psi)
    print(f"ClassState score: {score}")
    assert np.isclose(score, 1.0)

    # 3. MemoryBank
    mb = MemoryBank([cs])
    print("MemoryBank initialized.")
    scores = mb.scores(psi)
    print(f"MemoryBank scores: {scores}")
    assert np.isclose(scores[0], 1.0)

    # 4. Models
    # OnlinePerceptron
    op = OnlinePerceptron(cs, eta=0.1)
    y_hat, s, updated = op.step(psi, 1)
    print(f"OnlinePerceptron step: y_hat={y_hat}, s={s}, updated={updated}")

    # WinnerTakeAll
    wta = WinnerTakeAll(mb, eta=0.1, backend=backend)
    y_hat, idx, updated = wta.step(psi, 1)
    print(f"WinnerTakeAll step: y_hat={y_hat}, idx={idx}, updated={updated}")

    # AdaptiveMemory
    am = AdaptiveMemory(mb, eta=0.1, backend=backend)
    margin, spawned = am.step(psi, 1)
    print(f"AdaptiveMemory step: margin={margin}, spawned={spawned}")

    print("All basic consistency tests passed!")

if __name__ == "__main__":
    test_consistency()

```


File: src/training/run_final_comparison.py

```

import os
import json
import numpy as np
from tqdm import tqdm
from sklearn.metrics import accuracy_score
from sklearn.svm import SVC

from src.utils.paths import load_paths
from src.IQL.interference.exact_backend import ExactBackend
from src.IQL.interference.transition_backend import TransitionBackend
from src.IQL.baselines.static_isdo_classifier import StaticISDOClassifier

# -----
# Config
# -----
INCLUDE_QSVM = False
K_ISDO = 3 # chosen from K-sweep (best)

# -----
# Load paths and data
# -----
_, PATHS = load_paths()
EMBED_DIR = PATHS["embeddings"]
PROTO_DIR = PATHS["class_prototypes"]
LOG_DIR = PATHS["logs"]
QSVM_DIR = os.path.join(PATHS["artifacts"], "qsvm_cache")

X = np.load(os.path.join(EMBED_DIR, "val_embeddings.npy"))
y = np.load(os.path.join(EMBED_DIR, "val_labels_polar.npy"))

test_idx = np.load(os.path.join(EMBED_DIR, "split_test_idx.npy"))
X_test = X[test_idx]
y_test = y[test_idx]

# quantum-safe normalization (already true, but explicit)
X_test = X_test / np.linalg.norm(X_test, axis=1, keepdims=True)

# Load base prototype once to avoid disk I/O in loops
chi_single = np.load(os.path.join(PROTO_DIR, "K1/class1_proto0.npy"))

results = {}

# =====
# IQC - Exact (measurement-free)
# =====
exact_backend = ExactBackend()

print("Evaluating IQC-Exact...")
y_pred_exact = []

```

```

for psi in tqdm(X_test, desc="IQC Exact"):
    s = exact_backend.score(chi=chi_single, psi=psi)
    y_pred_exact.append(1 if s >= 0 else -1)

results["IQC_Exact_Backend"] = accuracy_score(y_test, y_pred_exact)

# =====
# IQC - Transition (circuit B')
# =====
transition_backend = TransitionBackend()

print("Evaluating IQC-Transition (Circuit-B')...")
y_pred_transition = []
for psi in tqdm(X_test, desc="IQC Transition"):
    s = transition_backend.score(chi=chi_single, psi=psi)
    y_pred_transition.append(1 if s >= 0 else -1)

results["IQC_Transition_Backend"] = accuracy_score(y_test,
y_pred_transition)

# =====
# ISDO - K-prototype interference ( Exact )
# =====
isdo = StaticISDOClassifier(PROTO_DIR, K_ISDO)
print(f"Evaluating ISDO-K (K={K_ISDO})...")
y_pred_isdo = isdo.predict(X_test)
results["ISDO_K"] = accuracy_score((y_test + 1) // 2, y_pred_isdo)

# =====
# Fidelity (SWAP test) - load cached result
# =====
results["Fidelity_SWAP"] = 0.8784 # from evaluate_swap_test_batch.py

# =====
# Classical baselines - load from logs
# =====
with open(os.path.join(LOG_DIR, "embedding_baseline_results.json")) as f:
    classical = json.load(f)

for k, v in classical.items():
    results[k] = v["accuracy"]

# =====
# QSVM (optional)
# =====
if INCLUDE_QSVM:
    print("Evaluating QSVM baseline...")
    try:
        K_train = np.load(os.path.join(QSVM_DIR, "qsvm_kernel_train.npy"))
        K_test = np.load(os.path.join(QSVM_DIR, "qsvm_kernel_test.npy"))
        y_train = np.load(os.path.join(QSVM_DIR, "y_train_sub.npy"))

        # Note: SVC expects kernel values, labels should correspond to
        kernel indices

```

```

        qsvm = SVC(kernel="precomputed")
        qsvm.fit(K_train, y_train)

        y_test_sub = np.load(os.path.join(QSVM_DIR, "y_test_sub.npy"))
        y_pred_qsvm = qsvm.predict(K_test)
        results["QSVM"] = accuracy_score(y_test_sub, y_pred_qsvm)

    except Exception as e:
        print(f"QSVM evaluation skipped: {e}")
        results["QSVM"] = None

# -----
# Save
# -----
with open("final_comparison_results.json", "w") as f:
    json.dump(results, f, indent=2)

print("\n=== FINAL COMPARISON ===")
for k, v in results.items():
    if v is not None:
        print(f"{k:25s}: {v:.4f}")
    else:
        print(f"{k:25s}: N/A")

```

File: src/training/compare_best_iqc_vs_classical.py

```

import os
import json
import numpy as np
from sklearn.metrics import accuracy_score

from src.utils.paths import load_paths
from src.IQL.training.adaptive_memory_trainer import AdaptiveMemoryTrainer

# -----
# Load paths
# -----
_, PATHS = load_paths()
EMBED_DIR = PATHS["embeddings"]
LOG_DIR = PATHS["logs"]

X = np.load(os.path.join(EMBED_DIR, "val_embeddings.npy"))
y = np.load(os.path.join(EMBED_DIR, "val_labels_polar.npy"))

train_idx = np.load(os.path.join(EMBED_DIR, "split_train_idx.npy"))
test_idx = np.load(os.path.join(EMBED_DIR, "split_test_idx.npy"))

X_train, y_train = X[train_idx], y[train_idx]
X_test, y_test = X[test_idx], y[test_idx]

```

```

X_train /= np.linalg.norm(X_train, axis=1, keepdims=True)
X_test  /= np.linalg.norm(X_test, axis=1, keepdims=True)

results = {}

# -----
# Best IQC
# -----
adaptive = AdaptiveMemoryTrainer()
adaptive.fit(X_train, y_train)
results["IQC_Adaptive"] = accuracy_score(
    y_test, adaptive.predict(X_test)
)

# -----
# Classical baselines (from logs)
# -----
with open(os.path.join(LOG_DIR, "embedding_baseline_results.json")) as f:
    classical = json.load(f)

for k, v in classical.items():
    results[k] = v["accuracy"]

print("\n=== Best IQC vs Classical ===")
for k, v in results.items():
    print(f"{k:25s}: {v}")

```

File: src/training/test_fixed_memory_iqc.py

```

import os
import numpy as np
from sklearn.metrics import accuracy_score

from src.utils.paths import load_paths
from src.IQL.models.fixed_memory_iqc import FixedMemoryIQC

def main():
    # -----
    # Load paths
    # -----
    _, PATHS = load_paths()
    EMBED_DIR = PATHS["embeddings"]

    # -----
    # Load embeddings and labels (polar)
    # -----
    X = np.load(os.path.join(EMBED_DIR, "val_embeddings.npy"))
    y = np.load(os.path.join(EMBED_DIR, "val_labels_polar.npy")) # ±1

```

```

train_idx = np.load(os.path.join(EMBED_DIR, "split_train_idx.npy"))
test_idx = np.load(os.path.join(EMBED_DIR, "split_test_idx.npy"))

X_train, y_train = X[train_idx], y[train_idx]
X_test, y_test = X[test_idx], y[test_idx]

# -----
# Quantum-safe normalization (defensive)
# -----
X_train /= np.linalg.norm(X_train, axis=1, keepdims=True)
X_test /= np.linalg.norm(X_test, axis=1, keepdims=True)

# -----
# Train Fixed-Memory IQC
# -----
K = 1
model = FixedMemoryIQC(K=K, eta=0.1)
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
acc = accuracy_score(y_test, y_pred)

print(f"✅ FixedMemoryIQC | K={K} | Test Accuracy: {acc:.4f}")

if __name__ == "__main__":
    main()

```

File: src/training/validate_backends.py

```

import numpy as np

from src.IQL.backends.exact import ExactBackend
from src.IQL.backends.hadamard import HadamardBackend
from src.IQL.backends.transition import TransitionBackend
from src.IQL.backends.prime_b import PrimeBBackend

def random_state(n_qubits, seed=None):
    if seed is not None:
        np.random.seed(seed)
    dim = 2 ** n_qubits
    v = np.random.randn(dim) + 1j * np.random.randn(dim)
    return v / np.linalg.norm(v)

def run_backend_tests(n_qubits=3, n_tests=20):
    backends = {
        "Exact": ExactBackend(),
        "Hadamard": HadamardBackend(),

```

```

    "Transition": TransitionBackend(),
    "PrimeB": PrimeBBackend(),
}

print(f"\nRunning backend tests with {n_qubits} qubits\n")

# Fix  $\chi$ 
chi = random_state(n_qubits, seed=42)

scores = {name: [] for name in backends}

for i in range(n_tests):
    psi = random_state(n_qubits, seed=100 + i)

    print(f"Test {i + 1}")
    for name, backend in backends.items():
        s = backend.score(chi, psi)
        scores[name].append(s)
        print(f"    {name:10s}: {s:+.6f}")
    print()

# -----
# Analysis
# -----
print("\n=== Backend Agreement Analysis ===\n")

exact = np.array(scores["Exact"])

for name in ["Hadamard", "Transition"]:
    diff = np.max(np.abs(exact - np.array(scores[name])))
    print(f"Max |Exact - {name}| = {diff:.2e}")

# PrimeB: sign + ordering only
primeb = np.array(scores["PrimeB"])

sign_match = np.mean(np.sign(primeb) == np.sign(exact))
print(f"\nPrimeB sign agreement with Exact: {sign_match * 100:.1f}%")

# Rank correlation (ordering)
exact_rank = np.argsort(exact)
primeb_rank = np.argsort(primeb)
rank_corr = np.corrcoef(exact_rank, primeb_rank)[0, 1]
print(f"PrimeB rank correlation with Exact: {rank_corr:.3f}")

if __name__ == "__main__":
    run_backend_tests(n_qubits=3, n_tests=200)

"""
Test 194
Exact      : -0.224492
Hadamard   : -0.224492
Transition: -0.224492

```

```
PrimeB      : +0.095676

Test 195
Exact       : -0.028519
Hadamard    : -0.028519
Transition:  -0.028519
PrimeB      : -0.423231

Test 196
Exact       : +0.203938
Hadamard    : +0.203938
Transition:  +0.203938
PrimeB      : -0.201812

Test 197
Exact       : +0.143895
Hadamard    : +0.143895
Transition:  +0.143895
PrimeB      : +0.035991

Test 198
Exact       : -0.111603
Hadamard    : -0.111603
Transition:  -0.111603
PrimeB      : -0.143718

Test 199
Exact       : +0.164120
Hadamard    : +0.164120
Transition:  +0.164120
PrimeB      : +0.107708

Test 200
Exact       : +0.145881
Hadamard    : +0.145881
Transition:  +0.145881
PrimeB      : -0.250643
```

```
=== Backend Agreement Analysis ===
```

```
Max |Exact - Hadamard| = 3.22e-15
Max |Exact - Transition| = 4.97e-14
```

```
PrimeB sign agreement with Exact: 52.5%
PrimeB rank correlation with Exact: -0.004
"""
```

File: src/training/test_static_isdo_model.py

```

import os
import numpy as np
from sklearn.metrics import accuracy_score

from src.utils.paths import load_paths
from src.IQL.models.static_isdo_model import StaticISDOModel

def main():
    # -----
    # Load paths
    # -----
    _, PATHS = load_paths()
    EMBED_DIR = PATHS["embeddings"]

    # -----
    # Load embeddings and labels
    # -----
    X = np.load(os.path.join(EMBED_DIR, "val_embeddings.npy"))
    y = np.load(os.path.join(EMBED_DIR, "val_labels.npy")) # {0,1}

    train_idx = np.load(os.path.join(EMBED_DIR, "split_train_idx.npy"))
    test_idx = np.load(os.path.join(EMBED_DIR, "split_test_idx.npy"))

    X_train, y_train = X[train_idx], y[train_idx]
    X_test, y_test = X[test_idx], y[test_idx]

    # -----
    # Sanity: ensure quantum-safe normalization
    # -----
    X_train = X_train / np.linalg.norm(X_train, axis=1, keepdims=True)
    X_test = X_test / np.linalg.norm(X_test, axis=1, keepdims=True)

    # -----
    # Run Static ISDO Model
    # -----
    K = 4 # best K from sweep
    model = StaticISDOModel(K=K)
    model.fit(X_train, y_train)

    y_pred = model.predict(X_test)

    acc = accuracy_score(y_test, y_pred)
    print(f"✅ StaticISDOModel | K={K} | Test Accuracy: {acc:.4f}")

if __name__ == "__main__":
    main()

```

File: src/training/compare_iqc_algorithms.py


```

import os
import numpy as np
from sklearn.metrics import accuracy_score

from src.utils.paths import load_paths
from src.IQL.baselines.static_isdo_classifier import StaticISDOClassifier
from src.IQL.training.online_perceptron_trainer import
OnlinePerceptronTrainer
from src.IQL.training.adaptive_memory_trainer import AdaptiveMemoryTrainer
from src.IQL.states.class_state import ClassState
from src.IQL.memory.memory_bank import MemoryBank
import pickle

# -----
# Load data
# -----
_, PATHS = load_paths()
EMBED_DIR = PATHS["embeddings"]
PROTO_DIR = PATHS["class_prototypes"]

X = np.load(os.path.join(EMBED_DIR, "val_embeddings.npy"))
y = np.load(os.path.join(EMBED_DIR, "val_labels_polar.npy"))

train_idx = np.load(os.path.join(EMBED_DIR, "split_train_idx.npy"))
test_idx = np.load(os.path.join(EMBED_DIR, "split_test_idx.npy"))

X_train, y_train = X[train_idx], y[train_idx]
X_test, y_test = X[test_idx], y[test_idx]

X_train /= np.linalg.norm(X_train, axis=1, keepdims=True)
X_test /= np.linalg.norm(X_test, axis=1, keepdims=True)

results = {}

# -----
# Static ISDO
# -----
isdo = StaticISDOClassifier(PROTO_DIR, K=3)
results["Static_ISDO"] = accuracy_score((y_test + 1)//2,
isdo.predict(X_test))

# -----
# IQC-Online (Regime-2)
# -----

# bootstrap initialization (important!)
chi0 = np.zeros_like(X_train[0])
for psi, label in zip(X_train[:10], y_train[:10]):
    chi0 += label * psi
chi0 = chi0 / np.linalg.norm(chi0)

class_state = ClassState(chi0)
online = OnlinePerceptronTrainer(class_state, eta=0.1)

```

```

online.fit(X_train, y_train)
results["IQC_Online"] = accuracy_score(y_test, online.predict(X_test))

# -----
# IQC-Adaptive Memory (Regime-3C)
# -----

MEMORY_PATH = os.path.join(PATHS["artifacts"], "regime3c_memory.pkl")

with open(MEMORY_PATH, "rb") as f:
    memory_bank = pickle.load(f)

adaptive = AdaptiveMemoryTrainer(
    memory_bank=memory_bank,
    eta=0.1,
    percentile=5,          #  $\tau$  = 5th percentile of margins
    tau_abs = -0.121,
    margin_window=500
)
adaptive.fit(X_train, y_train)

results["IQC_Adaptive"] = accuracy_score(
    y_test, adaptive.predict(X_test)
)
results["Adaptive_Memory_Size"] = adaptive.memory_size()

print("\n=== IQC Algorithm Comparison ===")
for k, v in results.items():
    print(f"{k:25s}: {v}")

## output
"""
=== IQC Algorithm Comparison ===
Static_ISDO           : 0.8806666666666667
IQC_Online            : 0.904
IQC_Adaptive          : 0.56
Adaptive_Memory_Size  : 45
"""

```

File: src/training/protocol_online/train_perceptron.py

```

import numpy as np
import os

from src.IQL.learning.class_state import ClassState
from src.IQL.encoding.embedding_to_state import embedding_to_state
from src.IQL.regimes.regime2_online import OnlinePerceptron
from src.IQL.learning.metrics import summarize_training
from src.IQL.backends.exact import ExactBackend
from src.utils.paths import load_paths
from src.utils.seed import set_seed

```

```

# -----
# Reproducibility
# -----
set_seed(42)

# -----
# Load paths
# -----
_, PATHS = load_paths()
EMBED_DIR = PATHS["embeddings"]

os.makedirs(EMBED_DIR, exist_ok=True)

# -----
# Load embeddings (TRAIN ONLY)
# -----
X = np.load(os.path.join(EMBED_DIR, "val_embeddings.npy"))
y = np.load(os.path.join(EMBED_DIR, "val_labels_polar.npy"))
train_idx = np.load(os.path.join(EMBED_DIR, "split_train_idx.npy"))

X_train = X[train_idx]
y_train = y[train_idx]

print("Loaded train embeddings:", X_train.shape)

def main():

    chi0 = np.zeros_like(X_train[0])
    for psi, label in zip(X_train[:10], y_train[:10]):
        chi0 += label * psi
    chi0 = chi0 / np.linalg.norm(chi0)

    class_state = ClassState(chi0, backend=ExactBackend())
    trainer = OnlinePerceptron(class_state, eta=0.1)

    acc = trainer.fit(X_train, y_train)
    stats = summarize_training(trainer.history)

    print("Final accuracy:", acc)
    print("Training stats:", stats)

if __name__ == "__main__":
    main()

#### output
"""
🌱 Global seed set to 42
Loaded train embeddings: (3500, 32)
Final accuracy: 0.8562857142857143
Training stats: {'mean_margin': 0.14930659062683652, 'min_margin':

```

```
-0.7069261085786833, 'num_updates': 503, 'update_rate': 0.1437142857142857}
"""
```

File: src/training/protocol_static/evaluate_isdo_k_sweep.py

```
import os
import numpy as np
from sklearn.metrics import accuracy_score

from src.IQL.baselines.static_isdo_classifier import StaticISDOClassifier
from src.utils.paths import load_paths
import matplotlib.pyplot as plt

BASE_ROOT, PATHS = load_paths()

EMBED_DIR = PATHS["embeddings"]
PROTO_BASE = PATHS["class_prototypes"]

X = np.load(os.path.join(EMBED_DIR, "val_embeddings.npy"))
y = np.load(os.path.join(EMBED_DIR, "val_labels.npy"))

test_idx = np.load(os.path.join(EMBED_DIR, "split_test_idx.npy"))

X_test = X[test_idx]
y_test = y[test_idx]

accuracy = []
for K in PATHS["class_count"]["K_values"]:

    clf = StaticISDOClassifier(PROTO_BASE, K)

    y_pred = clf.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    accuracy.append(acc)
    print(f"ISDO | K={K:<2} | Accuracy: {acc:.4f}")

"""
ISDO | K=1 | Accuracy: 0.8827
ISDO | K=2 | Accuracy: 0.8800
ISDO | K=3 | Accuracy: 0.8960 ## best
ISDO | K=5 | Accuracy: 0.8840
ISDO | K=7 | Accuracy: 0.8840
ISDO | K=11 | Accuracy: 0.8820
ISDO | K=13 | Accuracy: 0.8800
ISDO | K=17 | Accuracy: 0.8740
ISDO | K=19 | Accuracy: 0.8780
ISDO | K=23 | Accuracy: 0.8747
"""

plt.plot(PATHS["class_count"]["K_values"], accuracy, marker="o")
```

```
plt.xlabel("Number of prototypes per class (K)")
plt.ylabel("Test Accuracy")
plt.title("ISDO Accuracy vs Interference Capacity")
plt.grid(True)
plt.savefig(os.path.join(PATHS["figures"], "isdo_k_sweep.png"))
```

File: src/training/protocol_static/evaluate_static_isdo.py

```
import os
import numpy as np
from sklearn.metrics import accuracy_score

from src.IQL.baselines.static_isdo_classifier import StaticISDOClassifier
from src.utils.paths import load_paths

BASE_ROOT, PATHS = load_paths()

EMBED_DIR = PATHS["embeddings"]
PROTO_DIR = PATHS["class_prototypes"]
K = int(PATHS["class_count"]["K"])

X = np.load(os.path.join(EMBED_DIR, "val_embeddings.npy"))
y = np.load(os.path.join(EMBED_DIR, "val_labels.npy"))

test_idx = np.load(os.path.join(EMBED_DIR, "split_test_idx.npy"))

X_test = X[test_idx]
y_test = y[test_idx]

clf = StaticISDOClassifier(PROTO_DIR, K)
y_pred = clf.predict(X_test)

acc = accuracy_score(y_test, y_pred)
print(f"ISDO Accuracy (test): {acc:.4f}")

"""
ISDO Accuracy (test): 0.8840
"""
```

File: src/training/classical/make_embedding_split.py

```
import os
import numpy as np
from sklearn.model_selection import train_test_split

from src.utils.paths import load_paths
from src.utils.seed import set_seed

set_seed(42)
```

```

BASE_ROOT, PATHS = load_paths()
EMBED_DIR = PATHS["embeddings"]

X = np.load(os.path.join(EMBED_DIR, "val_embeddings.npy"))
y = np.load(os.path.join(EMBED_DIR, "val_labels.npy"))

indices = np.arange(len(y))

train_idx, test_idx = train_test_split(
    indices,
    test_size=0.3,
    random_state=42,
    stratify=y
)

np.save(os.path.join(EMBED_DIR, "split_train_idx.npy"), train_idx)
np.save(os.path.join(EMBED_DIR, "split_test_idx.npy"), test_idx)

print("Saved split:")
print("Train:", len(train_idx))
print("Test :", len(test_idx))

```

File: src/training/classical/train_embedding_models.py

```

import os
import json
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, normalize
from sklearn.metrics import accuracy_score, roc_auc_score

from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.neighbors import KNeighborsClassifier

from src.utils.paths import load_paths
from src.utils.seed import set_seed
set_seed(42)

# -----
# Load paths
# -----
BASE_ROOT, PATHS = load_paths()

EMBED_DIR = PATHS["embeddings"]
LOG_DIR = PATHS["logs"]
os.makedirs(LOG_DIR, exist_ok=True)

# -----

```

```

# Load embeddings
# -----
X = np.load(os.path.join(EMBED_DIR, "val_embeddings.npy"))
y = np.load(os.path.join(EMBED_DIR, "val_labels.npy"))

train_idx = np.load(os.path.join(EMBED_DIR, "split_train_idx.npy"))
test_idx = np.load(os.path.join(EMBED_DIR, "split_test_idx.npy"))

print("Loaded embeddings:", X.shape)

# -----
# Preprocessing (DEPRECATED: Now handled in extract_embeddings.py)
# -----
# # 1) Standardize (important for linear models)
# scaler = StandardScaler()
# X_std = scaler.fit_transform(X)
#
# # 2) L2-normalize (important for similarity & quantum)
# X_l2 = normalize(X_std, norm="l2")

# -----
# Train / test split
# -----

# Using raw pre-normalized float64 embeddings for all models
Xtr = X[train_idx]
Xte = X[test_idx]
ytr = y[train_idx]
yte = y[test_idx]

results = {}

# =====
# ❶ Logistic Regression (Linear separability)
# =====
print("\nTraining Logistic Regression...")
logreg = LogisticRegression(
    max_iter=1000,
    n_jobs=-1
)
logreg.fit(Xtr, ytr)

pred_lr = logreg.predict(Xte)
proba_lr = logreg.predict_proba(Xte)[: , 1]

results["LogisticRegression"] = {
    "accuracy": accuracy_score(yte, pred_lr),
    "auc": roc_auc_score(yte, proba_lr)
}

# =====
# ❷ Linear SVM (Max-margin)
# =====
print("Training Linear SVM...")

```

```

svm = LinearSVC()
svm.fit(Xtr, ytr)

pred_svm = svm.predict(Xte)

results["LinearSVM"] = {
    "accuracy": accuracy_score(yte, pred_svm),
    "auc": None # LinearSVC has no probability estimates
}

# =====
# ❸ k-NN (Distance-based similarity)
# =====
print("Training k-NN...")
knn = KNeighborsClassifier(
    n_neighbors=5,
    metric="euclidean"
)
knn.fit(Xtr, ytr)
print("Knn neighbors:", knn.n_neighbors)
pred_knn = knn.predict(Xte)
proba_knn = knn.predict_proba(Xte)[:, 1]

results["kNN"] = {
    "accuracy": accuracy_score(yte, pred_knn),
    "auc": roc_auc_score(yte, proba_knn)
}

# -----
# Save results
# -----
with open(os.path.join(LOG_DIR, "embedding_baseline_results.json"), "w") as f:
    json.dump(results, f, indent=2)

# -----
# Print summary
# -----
print("\n=== Embedding Baseline Results ===")
for model, metrics in results.items():
    print(
        f"{model:>18} | "
        f"Acc: {metrics['accuracy']:.4f} | "
        f"AUC: {metrics['auc']}"
    )

## output
"""
🌱 Global seed set to 42
Loaded embeddings: (5000, 32)

Training Logistic Regression...
Training Linear SVM...
Training k-NN...

```



```

Knn neighbors: 5

=== Embedding Baseline Results ===
LogisticRegression | Acc: 0.9047 | AUC: 0.9664224751066857
      LinearSVM | Acc: 0.9053 | AUC: None
      kNN | Acc: 0.9260 | AUC: 0.9711219772403983
"""

```

File: src/training/classical/extract_embeddings.py

```

import os
import torch
import numpy as np
from torch.utils.data import DataLoader, Subset
from tqdm import tqdm

from src.classical.cnn import PCamCNN
from src.data.pcam_loader import get_pcam_dataset
from src.data.transforms import get_eval_transforms
from src.utils.paths import load_paths
from src.utils.seed import set_seed

set_seed(42)

BASE_ROOT, PATHS = load_paths()
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"

CHECKPOINT = os.path.join(PATHS["checkpoints"], "pcam_cnn_best.pt")
os.makedirs(PATHS["embeddings"], exist_ok=True)

model = PCamCNN(embedding_dim=32).to(DEVICE)
model.load_state_dict(torch.load(CHECKPOINT, map_location=DEVICE))
model.eval()

dataset = get_pcam_dataset(PATHS["dataset"], "val", get_eval_transforms())
subset = Subset(dataset, range(5000))
loader = DataLoader(subset, batch_size=128, num_workers=6, pin_memory=True)

embeds, labels, lable_polar = [], [], []

with torch.no_grad():
    for x, y in tqdm(loader):
        z = model(x.to(DEVICE), return_embedding=True)
        # Convert to float64 FIRST, then normalize for maximum precision
        z = z.to(torch.float64)
        z = torch.nn.functional.normalize(z, p=2, dim=1)

        embeds.append(z.cpu().numpy())
        labels.append(y.numpy().astype(np.float64))
        lable_polar.append(((y.numpy()*2 - 1).astype(np.float64))

```

```
np.save(os.path.join(PATHS["embeddings"], "val_embeddings.npy"),
np.vstack(embeds).astype(np.float64))
np.save(os.path.join(PATHS["embeddings"], "val_labels.npy"),
np.concatenate(labels).astype(np.float64))
np.save(os.path.join(PATHS["embeddings"], "val_labels_polar.npy"),
np.concatenate(lable_polar).astype(np.float64))
```

File: src/training/classical/visualize_embeddings.py

```
import os
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
from src.utils.paths import load_paths
from src.utils.seed import set_seed

set_seed(42)

_, PATHS = load_paths()

X = np.load(os.path.join(PATHS["embeddings"], "val_embeddings.npy"))
y = np.load(os.path.join(PATHS["embeddings"], "val_labels.npy"))

tsne = TSNE(n_components=2, perplexity=30, max_iter=1000, random_state=42)
X2 = tsne.fit_transform(X)

plt.figure(figsize=(7, 6))
plt.scatter(X2[y == 0, 0], X2[y == 0, 1], s=8, label="Benign")
plt.scatter(X2[y == 1, 0], X2[y == 1, 1], s=8, label="Malignant")
plt.legend()
plt.savefig(os.path.join(PATHS["figures"], "embedding_tsne.png"), dpi=300)
plt.show()
```

File: src/training/classical/train_cnn.py

```
import os
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from tqdm import tqdm
import json
import matplotlib.pyplot as plt

from src.classical.cnn import PCamCNN
from src.data.pcam_loader import get_pcam_dataset
from src.data.transforms import get_train_transforms, get_eval_transforms
from src.utils.paths import load_paths
from src.utils.seed import set_seed
```

```

set_seed(42)
#torch.backends.cudnn.benchmark = True

# -----
# Load paths
# -----
BASE_ROOT, PATHS = load_paths()
DATA_ROOT = PATHS["dataset"]

# -----
# Config
# -----
BATCH_SIZE = 64
EPOCHS = 30
LR = 1e-3
EMBEDDING_DIM = 32
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"

os.makedirs(PATHS["checkpoints"], exist_ok=True)
os.makedirs(PATHS["logs"], exist_ok=True)
os.makedirs(PATHS["figures"], exist_ok=True)

# -----
# Training / Evaluation loops
# -----
def train_one_epoch(model, loader, criterion, optimizer):
    model.train()
    running_loss, correct, total = 0.0, 0, 0

    for images, labels in tqdm(loader, desc="Training", leave=False):
        images, labels = images.to(DEVICE), labels.to(DEVICE)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * images.size(0)
        correct += outputs.argmax(1).eq(labels).sum().item()
        total += labels.size(0)

    return running_loss / total, correct / total

@torch.no_grad()
def evaluate(model, loader, criterion):
    model.eval()
    running_loss, correct, total = 0.0, 0, 0

    for images, labels in tqdm(loader, desc="Validation", leave=False):
        images, labels = images.to(DEVICE), labels.to(DEVICE)
        outputs = model(images)
        loss = criterion(outputs, labels)

```

```

        running_loss += loss.item() * images.size(0)
        correct += outputs.argmax(1).eq(labels).sum().item()
        total += labels.size(0)

    return running_loss / total, correct / total

def main():
    print(f"🚀 Training on device: {DEVICE}")

    train_set = get_pcam_dataset(DATA_ROOT, "train",
get_train_transforms())
    val_set = get_pcam_dataset(DATA_ROOT, "val", get_eval_transforms())

    train_loader = DataLoader(train_set, BATCH_SIZE, shuffle=True,
num_workers=6, pin_memory=True)
    val_loader = DataLoader(val_set, BATCH_SIZE, shuffle=False,
num_workers=6, pin_memory=True)

    model = PCamCNN(embedding_dim=EMBEDDING_DIM).to(DEVICE)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=LR,
weight_decay=1e-4)
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
        optimizer, mode="max", factor=0.5, patience=2
    )

    best_val_acc, patience, wait = 0.0, 10, 0
    history = {k: [] for k in ["train_loss", "train_acc", "val_loss",
"val_acc"]}

    for epoch in range(1, EPOCHS + 1):
        print(f"\n📅 Epoch {epoch}/{EPOCHS}")

        tr_loss, tr_acc = train_one_epoch(model, train_loader, criterion,
optimizer)
        val_loss, val_acc = evaluate(model, val_loader, criterion)
        scheduler.step(val_acc)

        history["train_loss"].append(tr_loss)
        history["train_acc"].append(tr_acc)
        history["val_loss"].append(val_loss)
        history["val_acc"].append(val_acc)

        print(f"Train Acc {tr_acc:.4f} | Val Acc {val_acc:.4f}")

        if val_acc > best_val_acc:
            best_val_acc = val_acc
            torch.save(model.state_dict(),
os.path.join(PATHS["checkpoints"], "pcam_cnn_best.pt"))
            print("✅ Best validation accuracy reached : Saved checkpoint")
            wait = 0
        else:

```

```

        wait += 1

    if wait >= patience:
        print("■ Early stopping")
        break

    torch.save(model.state_dict(), os.path.join(PATHS["checkpoints"],
"pcam_cnn_final.pt"))
    print("✅ Final checkpoint saved")
    # Save logs
    with open(os.path.join(PATHS["logs"], "train_history.json"), "w") as f:
        json.dump(history, f, indent=2)

    # Plots
    epochs = range(1, len(history["train_loss"]) + 1)
    plt.figure()
    plt.plot(epochs, history["train_acc"], label="Train")
    plt.plot(epochs, history["val_acc"], label="Val")
    plt.legend()
    plt.savefig(os.path.join(PATHS["figures"], "cnn_accuracy.png"))
    plt.close()

    plt.figure()
    plt.plot(epochs, history["train_loss"], label="Train")
    plt.plot(epochs, history["val_loss"], label="Val")
    plt.legend()
    plt.savefig(os.path.join(PATHS["figures"], "cnn_loss.png"))
    plt.close()

if __name__ == "__main__":
    main()

```

File: src/training/classical/verify_embbeings.py

```

import os
import numpy as np
from src.utils.paths import load_paths

def verify_embeddings():
    BASE_ROOT, PATHS = load_paths()
    EMBED_DIR = PATHS["embeddings"]

    file_path = os.path.join(EMBED_DIR, "val_embeddings.npy")
    if not os.path.exists(file_path):
        print(f"File not found: {file_path}")
        return

    print(f"Verifying: {file_path}")
    X = np.load(file_path)

```

```
print(f"Shape: {X.shape}, Dtype: {X.dtype}")

# Calculate norm-squared for each sample
norms_sq = np.sum(X**2, axis=1)

max_val = np.max(norms_sq)
min_val = np.min(norms_sq)
mean_val = np.mean(norms_sq)

print(f"Max norm squared: {max_val:.15f}")
print(f"Min norm squared: {min_val:.15f}")
print(f"Mean norm squared: {mean_val:.15f}")

# Qiskit usually has a tolerance around 1e-8 or 1e-10
tolerance = 1e-8
violations = np.sum(np.abs(norms_sq - 1.0) > tolerance)

print(f"Violations (> {tolerance} absolute diff from 1.0):
{violations}")

if violations > 0:
    idx = np.argmax(np.abs(norms_sq - 1.0))
    print(f"Worst violation at index {idx}: {norms_sq[idx]:.15f}")

if __name__ == "__main__":
    verify_embeddings()
```

File: src/training/classical/visualize_pcam.py

```
import matplotlib.pyplot as plt
from src.data.pcam_loader import get_pcam_dataset
from src.utils.paths import load_paths
from src.utils.seed import set_seed

set_seed(42)

_, PATHS = load_paths()

dataset = get_pcam_dataset(PATHS["dataset"], "test")

plt.figure(figsize=(10, 5))
for i in range(2):
    img, label = dataset[i]
    plt.subplot(1, 2, i + 1)
    plt.imshow(img.permute(1, 2, 0))
    plt.title("Malignant" if label else "Benign")
    plt.axis("off")

plt.show()
```

File: src/training/protocol_adaptive/consolidate_memory.py

```

import os
import numpy as np

from src.utils.paths import load_paths
from src.utils.seed import set_seed

from src.IQL.encoding.embedding_to_state import embedding_to_state
from src.IQL.regimes.regime3a_wta import WinnerTakeAll
from src.IQL.inference.weighted_vote_classifier import
WeightedVoteClassifier
from src.IQL.backends.exact import ExactBackend
from src.IQL.learning.memory_bank import MemoryBank
import pickle

# -----
# Reproducibility
# -----
set_seed(42)

# -----
# Load paths
# -----
_, PATHS = load_paths()
EMBED_DIR = PATHS["embeddings"]
os.makedirs(EMBED_DIR, exist_ok=True)

# -----
# Load embeddings (TRAIN SPLIT)
# -----
X = np.load(os.path.join(EMBED_DIR, "val_embeddings.npy"))
y = np.load(os.path.join(EMBED_DIR, "val_labels_polar.npy"))
train_idx = np.load(os.path.join(EMBED_DIR, "split_train_idx.npy"))

X_train = X[train_idx]
y_train = y[train_idx]

print("Loaded train embeddings:", X_train.shape)

# -----
# ⑧ LOAD MEMORY BANK FROM REGIME 3-C
# -----
# IMPORTANT:
# This must be the SAME memory_bank produced by Regime 3-C

MEMORY_PATH = os.path.join(PATHS["artifacts"], "regime3c_memory.pkl")

with open(MEMORY_PATH, "rb") as f:

```

```

memory_bank = pickle.load(f)

print("Loaded memory bank with",
      len(memory_bank.class_states),
      "memories")

# -----
# 🔄 CONSOLIDATION PHASE (NO GROWTH)
# -----
# Use Regime 3-A trainer:
# - updates memories
# - NO spawning logic
trainer = WinnerTakeAll(
    memory_bank=memory_bank,
    eta=0.05,      # slightly smaller eta for stabilization
    backend=ExactBackend()
)

acc_train = trainer.fit(X_train, y_train)
print("Consolidation pass accuracy:", acc_train)
print("Updates during consolidation:", trainer.num_updates)

# -----
# 📊 FINAL EVALUATION (Regime 3-B inference)
# -----
classifier = WeightedVoteClassifier(memory_bank)

correct = 0
for x, y in zip(X_train, y_train):
    if classifier.predict(x) == y:
        correct += 1

final_acc = correct / len(X_train)
print("FINAL Regime 3-C accuracy:", final_acc)

#### output
"""
🌱 Global seed set to 42
Loaded train embeddings: (3500, 32)
Loaded memory bank with 22 memories
Consolidation pass accuracy: 0.8048571428571428
Updates during consolidation: 683
FINAL Regime 3-C accuracy: 0.884
"""

```

File: src/training/protocol_adaptive/train_adaptive_memory.py


```

import os
import numpy as np
from collections import Counter

from src.utils.paths import load_paths
from src.utils.seed import set_seed

from src.IQL.learning.class_state import ClassState
from src.IQL.encoding.embedding_to_state import embedding_to_state
from src.IQL.learning.memory_bank import MemoryBank
from src.IQL.backends.exact import ExactBackend

from src.IQL.regimes.regime3c_adaptive import AdaptiveMemory
from src.IQL.inference.weighted_vote_classifier import
WeightedVoteClassifier
import pickle

# -----
# Reproducibility
# -----
set_seed(42)

# -----
# Load paths
# -----
_, PATHS = load_paths()
EMBED_DIR = PATHS["embeddings"]
MEMORY_PATH = os.path.join(PATHS["artifacts"], "regime3c_memory.pkl")

os.makedirs(EMBED_DIR, exist_ok=True)
os.makedirs(PATHS["artifacts"], exist_ok=True)

# -----
# Load embeddings (TRAIN SPLIT)
# -----
X = np.load(os.path.join(EMBED_DIR, "val_embeddings.npy"))
y = np.load(os.path.join(EMBED_DIR, "val_labels_polar.npy"))
train_idx = np.load(os.path.join(EMBED_DIR, "split_train_idx.npy"))

X_train = X[train_idx]
y_train = y[train_idx]

print("Loaded train embeddings:", X_train.shape)

# -----
# Initialize memory bank (M = 3)
# -----
d = X_train[0].shape[0]

backend = ExactBackend()

```

```

class_states = []
for _ in range(3):
    v = np.random.randn(d)
    v /= np.linalg.norm(v)
    class_states.append(ClassState(v, backend=backend))

memory_bank = MemoryBank(
    class_states=class_states
)

print("Initial number of memories:", len(memory_bank.class_states))

# -----
# Train Regime 3-C (percentile-based  $\tau$ )
# -----
trainer = AdaptiveMemory(
    memory_bank=memory_bank,
    eta=0.1,
    percentile=5,          #  $\tau$  = 5th percentile of margins
    tau_abs = -0.121,
    margin_window=500,    # sliding window for stability
    backend=backend,
)

trainer.fit(X_train, y_train)

print("Training finished.")
print("Number of memories after training:", len(memory_bank.class_states))
print("Number of spawned memories:", trainer.num_spawns)
print("Number of updates:", trainer.num_updates)

# -----
# Evaluate using Regime 3-B inference
# -----
classifier = WeightedVoteClassifier(memory_bank)

correct = 0
for psi, y in zip(X_train, y_train):
    if classifier.predict(psi) == y:
        correct += 1

acc_3c = correct / len(X_train)
print("Regime 3-C accuracy (3-B inference):", acc_3c)

# -----
# Optional diagnostics
# -----
print("Final memory count:", len(memory_bank.class_states))

```

```

with open(MEMORY_PATH, "wb") as f:
    pickle.dump(memory_bank, f)

print("Saved Regime 3-C memory bank.")

### output
"""
🌱 Global seed set to 42
Loaded train embeddings: (3500, 32)
Initial number of memories: 3
Training finished.
Number of memories after training: 22
Number of spawned memories: 19
Number of updates: 429
Regime 3-C accuracy (3-B inference): 0.788
Final memory count: 22
Saved Regime 3-C memory bank.
"""

```

File: src/classical/cnn.py

```

import torch
import torch.nn as nn
import torch.nn.functional as F

class PCamCNN(nn.Module):
    """
    Lightweight CNN for PCam feature extraction.
    Produces low-dimensional embeddings suitable for quantum encoding.
    """

    def __init__(self, embedding_dim: int = 32, num_classes: int = 2):
        super().__init__()

        # ----- Convolutional backbone -----
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),  # 48x48

            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),  # 24x24

            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),

```

```
        nn.AdaptiveAvgPool2d((1, 1)) # 128 x 1 x 1
    )

    # ----- Embedding head -----
    self.embedding = nn.Linear(128, embedding_dim)

    # ----- Temporary classifier (used ONLY for CNN training) -----
    --
    self.classifier = nn.Linear(embedding_dim, num_classes)

def forward(self, x, return_embedding: bool = False):
    x = self.features(x)
    x = x.view(x.size(0), -1) # flatten

    embedding = self.embedding(x)
    embedding = F.relu(embedding)

    if return_embedding:
        return embedding

    logits = self.classifier(embedding)
    return logits
```

File: src/classical/**init**.py