

# Project Summary

---

## Directory Structure

```
measurement-free-quantum-classifier/  
  configs/  
    paths.yaml  
  src/  
    __init__.py  
    ISDO/  
      __init__.py  
      observables/  
        isdo.py  
        __init__.py  
      circuits/  
        transition_isdo.py  
        __init__.py  
      baselines/  
        static_isdo_classifier.py  
        __init__.py  
    utils/  
      common_backup.py  
      common.py  
      paths.py  
      seed.py  
      __init__.py  
    data/  
      pcam_loader.py  
      transforms.py  
      __init__.py  
    quantum/  
      compute_qsvm_kernel.py  
      __init__.py  
    classical/  
      cnn.py  
      __init__.py  
    experiments/  
      run_final_comparison.py  
      __init__.py  
    iqc/  
      consolidate_memory.py  
      train_perceptron.py  
      train_adaptive_memory.py  
      verify_transition_backend.py  
      __init__.py  
    isdo/  
      evaluate_isdo_k_sweep.py  
      evaluate_transition_isdo.py  
      evaluate_static_isdo.py  
      __init__.py
```

```
        prototype/  
            calculate_prototype.py  
            __init__.py  
    classical/  
        make_embedding_split.py  
        train_embedding_models.py  
        extract_embeddings.py  
        visualize_embeddings.py  
        train_cnn.py  
        verify_embeddings.py  
        __init__.py  
        visualize_pcm.py  
    IQC/  
        __init__.py  
    learning/  
        perceptron_update.py  
        __init__.py  
    states/  
        class_state.py  
        __init__.py  
    training/  
        winner_take_all_trainer.py  
        adaptive_memory_trainer.py  
        online_perceptron_trainer.py  
        metrics.py  
        __init__.py  
    memory/  
        memory_bank.py  
        __init__.py  
    interference/  
        base.py  
        transition_backend.py  
        transition_backend_backup.py  
        exact_backend.py  
        oracle_backend.py  
        __init__.py  
    inference/  
        weighted_vote_classifier.py  
        __init__.py  
    encoding/  
        embedding_to_state.py  
        __init__.py  
    Archive_src/  
        __init__.py  
    swap_test/  
        swap_test_classifier.py  
        evaluate_swap_test_batch.py  
        __init__.py  
    quantum/  
        __init__.py  
    isdo/  
        __init__.py  
        isdo_K_sweep/  
            old_evaluate_interference_k4.py
```

```
    __init__.py
    isdo_circuit_test/
        test_isdo_circuits_v1.py
    __init__.py
    circuit/
        circuit_a_controlled_state.py
    __init__.py
statevector_similarity/
    compute_class_states.py
    evaluate_statevector_similarity.py
    __init__.py
rfc/
    reflection_classifier.py
    __init__.py
experiments/
    __init__.py
    iqc/
        run_regime3c_v1.py
        run_regime3b.py
        verify_isdo_bprime_backend.py
        verify_hadamard_backend.py
        run_regime3a.py
        __init__.py
IQC/
    __init__.py
    training/
        regime3c_trainer_v1.py
        __init__.py
    interference/
        __init__.py
    inference/
        regime3a_classifier.py
        __init__.py
research_docs/
    comparison_report.md
    implementation_plan.md

interference_quantum_classifier_iqc_paper_draft_non_claim_leaking.md
Fidelity_and_Measurement_Free_Methods_Comparison.md
research_answers.md
project_blueprint.md
results/
    artifacts/
        regime3c_memory.pkl
    checkpoints/
        pcam_cnn_final.pt
        pcam_cnn_best.pt
    qsvm_cache/
    embeddings/
        val_labels.npy
        val_labels_polar.npy
        val_embeddings.npy
        split_test_idx.npy
        split_train_idx.npy
```

```
class_states_meta.json
class_prototypes/
  K7/
    class0_proto5.npy
    class1_proto6.npy
    class1_proto0.npy
    class1_proto1.npy
    class1_proto2.npy
    class0_proto0.npy
    class0_proto3.npy
    class0_proto1.npy
    class0_proto4.npy
    class1_proto4.npy
    class1_proto3.npy
    class0_proto2.npy
    class0_proto6.npy
    class1_proto5.npy
  K17/
    class0_proto12.npy
    class0_proto5.npy
    class1_proto6.npy
    class1_proto10.npy
    class1_proto0.npy
    class1_proto7.npy
    class1_proto1.npy
    class1_proto9.npy
    class1_proto2.npy
    class1_proto14.npy
    class1_proto8.npy
    class1_proto11.npy
    class1_proto13.npy
    class0_proto0.npy
    class0_proto7.npy
    class0_proto3.npy
    class0_proto14.npy
    class0_proto1.npy
    class0_proto4.npy
    class0_proto9.npy
    class0_proto11.npy
    class0_proto16.npy
    class0_proto8.npy
    class0_proto15.npy
    class1_proto12.npy
    class0_proto10.npy
    class1_proto15.npy
    class1_proto16.npy
    class1_proto4.npy
    class1_proto3.npy
    class0_proto2.npy
    class0_proto6.npy
    class0_proto13.npy
    class1_proto5.npy
  K1/
    class1_proto0.npy
```

```
class0_proto0.npy
K13/
class0_proto12.npy
class0_proto5.npy
class1_proto6.npy
class1_proto10.npy
class1_proto0.npy
class1_proto7.npy
class1_proto1.npy
class1_proto9.npy
class1_proto2.npy
class1_proto8.npy
class1_proto11.npy
class0_proto0.npy
class0_proto7.npy
class0_proto3.npy
class0_proto1.npy
class0_proto4.npy
class0_proto9.npy
class0_proto11.npy
class0_proto8.npy
class1_proto12.npy
class0_proto10.npy
class1_proto4.npy
class1_proto3.npy
class0_proto2.npy
class0_proto6.npy
class1_proto5.npy
K23/
class1_proto18.npy
class0_proto12.npy
class0_proto5.npy
class1_proto6.npy
class0_proto18.npy
class1_proto10.npy
class1_proto0.npy
class0_proto19.npy
class0_proto21.npy
class1_proto7.npy
class1_proto1.npy
class1_proto9.npy
class1_proto2.npy
class1_proto14.npy
class1_proto8.npy
class1_proto11.npy
class1_proto13.npy
class0_proto0.npy
class0_proto7.npy
class0_proto17.npy
class0_proto20.npy
class1_proto22.npy
class0_proto3.npy
class0_proto14.npy
class0_proto1.npy
```

class0\_proto4.npy  
class0\_proto9.npy  
class0\_proto11.npy  
class0\_proto16.npy  
class0\_proto8.npy  
class0\_proto15.npy  
class1\_proto12.npy  
class1\_proto21.npy  
class0\_proto10.npy  
class0\_proto22.npy  
class1\_proto19.npy  
class1\_proto15.npy  
class1\_proto16.npy  
class1\_proto4.npy  
class1\_proto20.npy  
class1\_proto3.npy  
class0\_proto2.npy  
class0\_proto6.npy  
class0\_proto13.npy  
class1\_proto17.npy  
class1\_proto5.npy

K5/

class1\_proto0.npy  
class1\_proto1.npy  
class1\_proto2.npy  
class0\_proto0.npy  
class0\_proto3.npy  
class0\_proto1.npy  
class0\_proto4.npy  
class1\_proto4.npy  
class1\_proto3.npy  
class0\_proto2.npy

K11/

class0\_proto5.npy  
class1\_proto6.npy  
class1\_proto10.npy  
class1\_proto0.npy  
class1\_proto7.npy  
class1\_proto1.npy  
class1\_proto9.npy  
class1\_proto2.npy  
class1\_proto8.npy  
class0\_proto0.npy  
class0\_proto7.npy  
class0\_proto3.npy  
class0\_proto1.npy  
class0\_proto4.npy  
class0\_proto9.npy  
class0\_proto8.npy  
class0\_proto10.npy  
class1\_proto4.npy  
class1\_proto3.npy  
class0\_proto2.npy  
class0\_proto6.npy

```
class1_proto5.npy
K19/
class1_proto18.npy
class0_proto12.npy
class0_proto5.npy
class1_proto6.npy
class0_proto18.npy
class1_proto10.npy
class1_proto0.npy
class1_proto7.npy
class1_proto1.npy
class1_proto9.npy
class1_proto2.npy
class1_proto14.npy
class1_proto8.npy
class1_proto11.npy
class1_proto13.npy
class0_proto0.npy
class0_proto7.npy
class0_proto17.npy
class0_proto3.npy
class0_proto14.npy
class0_proto1.npy
class0_proto4.npy
class0_proto9.npy
class0_proto11.npy
class0_proto16.npy
class0_proto8.npy
class0_proto15.npy
class1_proto12.npy
class0_proto10.npy
class1_proto15.npy
class1_proto16.npy
class1_proto4.npy
class1_proto3.npy
class0_proto2.npy
class0_proto6.npy
class0_proto13.npy
class1_proto17.npy
class1_proto5.npy
K2/
class1_proto0.npy
class1_proto1.npy
class0_proto0.npy
class0_proto1.npy
K3/
class1_proto0.npy
class1_proto1.npy
class1_proto2.npy
class0_proto0.npy
class0_proto1.npy
class0_proto2.npy
logs/
train_history.json
```

```
embedding_baseline_results.json
figures/
```

## 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/ISDO/init.py

## File: src/ISDO/observables/isdo.py

```
# src/ISDO/observables/isdo.py
import numpy as np
from src.ISDO.circuits.transition_isdo import run as run_isdo_circuit

def isdo_observable(chi, psi, real=True) -> float:
    """
    ISDO observable:
    Linear interference score  $\text{Re}\langle\chi|\psi\rangle$ 
    """
    if real:
        return float(np.real(np.vdot(chi, psi)))
    else:
        # Use the quantum circuit to compute the observable
        return run_isdo_circuit(psi, chi)
```



File: src/ISDO/observables/init.py

File: src/ISDO/circuits/transition\_isdo.py

```
import numpy as np
from qiskit import QuantumCircuit
from qiskit.quantum_info import Statevector, Pauli
from src.utils.common import build_transition_unitary

def build(psi, chi):
    """
    ISDO Circuit B': Transition-based interference (CORRECT PHYSICAL
    IMPLEMENTATION)

    This circuit measures  $\text{Re}\langle\chi|\psi\rangle$  using a controlled transition unitary.

    Circuit structure:
      Ancilla:  $|0\rangle$  —H—●—H—M
                |
      Data:     $|\psi\rangle$  —U $_{\chi\psi}$ —

    Where U $_{\chi\psi}$  is the transition unitary: U $_{\chi\psi}$   $|\psi\rangle = |\chi\rangle$ 

    This produces LINEAR interference, not quadratic!
    """
    # Ensure complex128 for Qiskit compatibility
    psi = np.asarray(psi, dtype=np.complex128)
    chi = np.asarray(chi, dtype=np.complex128)

    n = int(np.log2(len(psi)))
    qc = QuantumCircuit(1 + n, 1)

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

    # Prepare  $|\psi\rangle$  on data qubits
    from qiskit.circuit.library import StatePreparation
    qc.append(StatePreparation(psi), data)

    # Hadamard on ancilla
    qc.h(anc)

    # Controlled transition unitary
    U_chi_psi = build_transition_unitary(psi, chi)
```

```

qc.append(U_chi_psi.control(1), [anc] + data)

# Final Hadamard
qc.h(anc)

# Measure ancilla
#qc.measure(anc, 0)

return qc

def run(psi, chi):
    """
    Exact (statevector) evaluation of  $\langle Z \rangle$  which gives  $\text{Re}\langle \chi | \psi \rangle$ 

    This is the CORRECT physical implementation of ISDO.
    """
    qc = build(psi, chi)
    #qc_no_meas = qc.remove_final_measurements(inplace=False)
    sv = Statevector.from_instruction(qc)
    z_exp = sv.expectation_value(Pauli('Z'), [0]).real
    return z_exp

def verify(psi, chi):
    """
    Verify that the circuit correctly computes  $\text{Re}\langle \chi | \psi \rangle$ 
    """
    # Normalize inputs
    psi = np.asarray(psi, dtype=np.complex128)
    chi = np.asarray(chi, dtype=np.complex128)
    psi = psi / np.linalg.norm(psi)
    chi = chi / np.linalg.norm(chi)

    # Expected value
    expected = np.real(np.vdot(chi, psi))

    # Circuit result
    measured = run(psi, chi)

    # Check
    is_correct = np.allclose(measured, expected, atol=1e-10)

    print(f"Expected: {expected}")
    print(f"Measured: {measured}")
    print(f"Correct: {is_correct}")

    return is_correct

```

File: src/ISDO/circuits/init.py

## File: src/ISDO/baselines/static\_isdo\_classifier.py

```
import os
import numpy as np
from tqdm import tqdm
from src.ISDO.observables.isdo import isdo_observable

class StaticISDOClassifier:
    def __init__(self, proto_dir, K):
        self.proto_dir = proto_dir
        self.K = K
        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 isdo_observable(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/ISDO/baselines/init.py

## 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/utls/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/utils/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/utils/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/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
            transforms.ToCHW()
        ])

    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/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

File: src/experiments/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.IQC.interference.exact_backend import ExactBackend
from src.IQC.interference.transition_backend import TransitionBackend
from src.ISDO.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"] = 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"] = 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/experiments/init.py

File: src/experiments/iqc/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.IQC.encoding.embedding_to_state import embedding_to_state
from src.IQC.training.winner_take_all_trainer import WinnerTakeAllTrainer
from src.IQC.inference.weighted_vote_classifier import
WeightedVoteClassifier
from src.IQC.inference.exact_backend import ExactBackend

# -----
# 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)

# -----
# Prepare dataset
# -----
dataset = [
    (embedding_to_state(x), int(label))
    for x, label in zip(X_train, y_train)
]

# shuffle (important for consolidation)
rng = np.random.default_rng(42)
perm = rng.permutation(len(dataset))
dataset = [dataset[i] for i in perm]

# -----
# 🔄 LOAD MEMORY BANK FROM REGIME 3-C
# -----
# IMPORTANT:
# This must be the SAME memory_bank produced by Regime 3-C
from src.IQC.memory.memory_bank import MemoryBank
import pickle

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 = WinnerTakeAllTrainer(
    memory_bank=memory_bank,
    eta=0.05      # slightly smaller eta for stabilization
)

acc_train = trainer.train(dataset)
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 psi, y in dataset:
    if classifier.predict(psi) == y:
        correct += 1

final_acc = correct / len(dataset)
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/experiments/iqc/train\_perceptron.py

```

import numpy as np
import os

from src.IQC.states.class_state import ClassState
from src.IQC.encoding.embedding_to_state import embedding_to_state
from src.IQC.training.online_perceptron_trainer import
OnlinePerceptronTrainer
from src.IQC.training.metrics import summarize_training

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():

    dataset = [
        (embedding_to_state(x), int(label))
        for x, label in zip(X_train, y_train)
    ]

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

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

    acc = trainer.train(dataset)
    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/experiments/iqc/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.IQC.states.class_state import ClassState
from src.IQC.encoding.embedding_to_state import embedding_to_state
from src.IQC.memory.memory_bank import MemoryBank
from src.IQC.interference.exact_backend import ExactBackend
from src.IQC.interference.oracle_backend import OracleBackend

from src.IQC.training.adaptive_memory_trainer import AdaptiveMemoryTrainer
from src.IQC.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)

# -----
# Prepare dataset (same as Regime 2 / 3-A / 3-B)
# -----
dataset = [
    (embedding_to_state(x), int(label))
    for x, label in zip(X_train, y_train)
]

# shuffle (important for online + growth)

```

```

rng = np.random.default_rng(42)
perm = rng.permutation(len(dataset))
dataset = [dataset[i] for i in perm]

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

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

backend = ExactBackend()
backend_hadamard = OracleBackend()

memory_bank = MemoryBank(
    class_states=class_states,
    backend=backend
)

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

# -----
# Train Regime 3-C (percentile-based  $\tau$ )
# -----
trainer = AdaptiveMemoryTrainer(
    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
)

trainer.train(dataset)

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 dataset:
    if classifier.predict(psi) == y:

```

```

        correct += 1

acc_3c = correct / len(dataset)
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/experiments/iqc/verify\_transition\_backend.py

```

import numpy as np
from src.IQC.interference.exact_backend import ExactBackend
from src.IQC.interference.transition_backend import TransitionBackend

def random_state(n):
    v = np.random.randn(2**n) + 1j * np.random.randn(2**n)
    v /= np.linalg.norm(v)
    return v

def sign(x):
    return 1 if x >= 0 else -1

np.random.seed(0)

math_backend = ExactBackend()
TransitionBackend = TransitionBackend()

```

```

n = 3 # small, exact verification
num_tests = 50

sign_agree = 0
vals = []

for _ in range(num_tests):
    chi = random_state(n)
    psi = random_state(n)

    s_math = math_backend.score(chi, psi)
    s_transition = TransitionBackend.score(chi, psi)

    vals.append((s_math, s_transition))

    if sign(s_math) == sign(s_transition):
        sign_agree += 1

print("Sign agreement:", sign_agree, "/", num_tests)
print("Mean abs error:", np.mean([abs(a - b) for a, b in vals]))

## output
"""
Sign agreement: 50 / 50
Mean abs error: 1.3332529524845427e-14
"""

```

File: src/experiments/iqc/init.py

File: src/experiments/isdo/evaluate\_isdo\_k\_sweep.py

```

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

from src.ISDO.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"]:
    #proto_dir = os.path.join(PROTO_BASE, f"K{K}")
    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/experiments/isdo/evaluate\_transition\_isdo.py

```

"""
Comparison of ISDO Circuit Implementations

This script demonstrates three approaches:
1. Circuit A: Conceptual (Oracle model) - for pedagogy only
2. Circuit B: Reflection-based - gives quadratic fidelity
3. Circuit B': Transition-based - CORRECT linear ISDO

Only Circuit B' gives the true ISDO observable:  $\text{Re}\langle\chi|\psi\rangle$ 
"""

import numpy as np

```

```

from src.ISDO.circuits.transition_isdo import run, verify

def test_all_circuits():
    """
    Test all three circuit implementations and compare results
    """
    # Create two test states
    psi = np.array([0.6, 0.8, 0.0, 0.0], dtype=np.complex128)
    chi = np.array([0.8, 0.6, 0.0, 0.0], dtype=np.complex128)

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

    # Expected ISDO value:  $\text{Re}\langle\chi|\psi\rangle$ 
    expected_isdo = np.real(np.vdot(chi, psi))

    # Expected RFC (quadratic):  $1 - 2|\langle\chi|\psi\rangle|^2$ 
    inner_product_magnitude_sq = np.abs(np.vdot(chi, psi))**2
    expected_rfc = 1 - 2 * inner_product_magnitude_sq

    print("=" * 70)
    print("ISDO CIRCUIT COMPARISON")
    print("=" * 70)
    print(f"\n $|\psi\rangle = \{\text{psi}\}$ ")
    print(f" $|\chi\rangle = \{\text{chi}\}$ ")
    print(f" $\langle\chi|\psi\rangle = \{\text{np.vdot(chi, psi)}\}$ ")
    print(f" $|\langle\chi|\psi\rangle|^2 = \{\text{inner\_product\_magnitude\_sq}\}$ ")
    print()

    # Circuit B': Transition-based (CORRECT)
    print("-" * 70)
    print("Circuit B': Transition-Based Interference (CORRECT)")
    print("-" * 70)
    print("Purpose: CORRECT physical ISDO implementation")
    print("Observable:  $\text{Re}\langle\chi|\psi\rangle$  (linear, signed, phase-sensitive)")
    print("Status: Use this for all hardware and claims")
    try:
        result_b_prime = run(psi, chi)
        print(f"Result:      {result_b_prime:.6f}")
        print(f"Expected: {expected_isdo:.6f}")
        print(f"Match:      {np.allclose(result_b_prime, expected_isdo, atol=1e-6)}")

        print("\nRunning full verification...")
        verify(psi, chi)
    except Exception as e:
        print(f"Error: {e}")
    print()

    # Summary
    print("=" * 70)
    print("SUMMARY")

```

```

print("=" * 70)
print(f"True ISDO ( $\text{Re}\langle\chi|\psi\rangle$ ): {expected_isdo:.6f}")
print(f"RFC alternative ( $1-2|\langle\chi|\psi\rangle|^2$ ): {expected_rfc:.6f}")
print()
print("✓ Circuit A: Conceptual/oracle model only")
print("✗ Circuit B: Gives RFC (quadratic), not ISDO")
print("✓ Circuit B': CORRECT implementation - USE THIS")
print()

def test_different_states():
    """
    Test with multiple state pairs to show the difference
    """
    print("\n" + "=" * 70)
    print("TESTING MULTIPLE STATE PAIRS")
    print("=" * 70)

    test_cases = [
        # Same states
        (np.array([1.0, 0, 0, 0]), np.array([1.0, 0, 0, 0])),
        # Orthogonal states
        (np.array([1.0, 0, 0, 0]), np.array([0, 1.0, 0, 0])),
        # Opposite states
        (np.array([1.0, 0, 0, 0]), np.array([-1.0, 0, 0, 0])),
        # General case
        (np.array([0.6, 0.8, 0, 0]), np.array([0.8, -0.6, 0, 0])),
    ]

    for i, (psi, chi) in enumerate(test_cases, 1):
        psi = psi / np.linalg.norm(psi)
        chi = chi / np.linalg.norm(chi)

        true_isdo = np.real(np.vdot(chi, psi))
        rfc = 1 - 2 * np.abs(np.vdot(chi, psi))**2

        try:
            measured_b_prime = run(psi, chi)

            print(f"\nTest {i}:")
            print(f"  True ISDO ( $\text{Re}\langle\chi|\psi\rangle$ ): {true_isdo:+.4f}")
            print(f"  Circuit B' (transition): {measured_b_prime:+.4f} ✓")
        except Exception as e:
            print(f"\nTest {i}: Error - {e}")

if __name__ == "__main__":
    test_all_circuits()
    test_different_states()

```

File: src/experiments/isdo/evaluate\_static\_isdo.py

```

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

from src.ISDO.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/experiments/isdo/**init**.py

File: src/experiments/isdo/prototype/calculate\_prototype.py

```

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

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"]
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.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"))

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

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

K_VALUES = PATHS["class_count"]["K_values"]
# -----
# Helper: quantum-safe normalize
# -----
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

# -----
# K-sweep prototype generation
# -----

for K in K_VALUES:
    print(f"\n=== Computing prototypes for K={K} ===")

    CLASS_DIR = os.path.join(PROTO_BASE, f"K{K}")
    os.makedirs(CLASS_DIR, exist_ok=True)

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

        print(f"Clustering class {cls} with {len(X_cls)} samples")

        kmeans = KMeans(
            n_clusters=K,
            random_state=42,
            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(CLASS_DIR, f"class{cls}_proto{i}.npz")  
    np.save(path, proto)  
    print(f"Saved {path}")
```

File: src/experiments/isdo/prototype/init.py

File: src/experiments/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.npz"))  
y = np.load(os.path.join(EMBED_DIR, "val_labels.npz"))  
  
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.npz"), train_idx)  
np.save(os.path.join(EMBED_DIR, "split_test_idx.npz"), test_idx)  
  
print("Saved split:")  
print("Train:", len(train_idx))  
print("Test :", len(test_idx))
```

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

File: src/experiments/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/IQC/**init.py**

File: src/IQC/learning/perceptron\_update.py

```
import numpy as np  
from src.ISDO.observables.isdo import isdo_observable  
  
def perceptron_update(  

```

```

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

    If  $y * \text{Re}\langle \chi | \psi \rangle \geq 0$ :
        no update
    else:
         $\chi \leftarrow \text{normalize}(\chi + \eta * y * \psi)$ 
    """
    s = isdo_observable(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/IQC/learning/init.py

File: src/IQC/states/class\_state.py

```

import numpy as np
from src.ISDO.observables.isdo import isdo_observable

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  $|\chi\rangle$ .
    Invariant:  $||\chi|| = 1$  always.
    """

    def __init__(self, vector: np.ndarray):

```



```

        self.vector = normalize(vector.astype(np.complex128))

    def score(self, psi: np.ndarray) -> float:
        """
        ISDO score:  $\text{Re} \langle \chi | \psi \rangle$ 
        """
        return isdo_observable(self.vector, psi)

    def update(self, delta: np.ndarray):
        """
        Update  $|\chi\rangle \leftarrow \text{normalize}(|\chi\rangle + \text{delta})$ 
        """
        self.vector = normalize(self.vector + delta)

```

File: src/IQC/states/init.py

File: src/IQC/training/winner\_take\_all\_trainer.py

```

from ..learning.perceptron_update import perceptron_update

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

    def __init__(self, memory_bank, eta):
        self.memory_bank = memory_bank
        self.eta = eta
        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 = perceptron_update(
            cs.vector, psi, y, self.eta
        )

        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 train(self, dataset):
    correct = 0
    for psi, y in dataset:
        y_hat, _, _ = self.step(psi, y)
        if y_hat == y:
            correct += 1
    return correct / len(dataset)

```

File: src/IQC/training/adaptive\_memory\_trainer.py

```

import numpy as np
from collections import deque
from ..learning.perceptron_update import perceptron_update

class AdaptiveMemoryTrainer:
    """
    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,
    ):
        self.memory_bank = memory_bank
        self.eta = eta
        self.percentile = percentile
        self.tau_abs = tau_abs

        # 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.num_spawns += 1
                spawned = True

        # otherwise, normal Regime-2 update on winner
        if not spawned and margin < 0:
            idx, _ = self.memory_bank.winner(psi)
            cs = self.memory_bank.class_states[idx]

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

            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 train(self, dataset):
    for psi, y in dataset:
        self.step(psi, y)
```

File: src/IQC/training/online\_perceptron\_trainer.py

```
import numpy as np
from ..learning.perceptron_update import perceptron_update
from src.ISDO.observables.isdo import isdo_observable

class OnlinePerceptronTrainer:
    """
    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.
        """
        chi_vec = self.class_state.vector
        s = isdo_observable(chi_vec, psi)
        margin = y * s
        y_hat = 1 if s >= 0 else -1

        chi_new, updated = perceptron_update(
            chi_vec, psi, y, self.eta
        )

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

        for psi, y in dataset:
            y_hat, _, _ = self.step(psi, y)
            if y_hat == y:
                correct += 1

        accuracy = correct / len(dataset)
        return accuracy
```

## File: src/IQC/training/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/IQC/training/init.py

## File: src/IQC/memory/memory\_bank.py

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

```

        self.backend = backend

    def scores(self, psi):
        return [
            self.backend.score(cs.vector, 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):
        from ..states.class_state import ClassState
        self.class_states.append(ClassState(chi_vector))

```

File: src/IQC/memory/init.py

File: src/IQC/interference/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/IQC/interference/transition\_backend.py

```

from src.ISDO.circuits.transition_isdo import run as run_isdo_circuit
from .base import InterferenceBackend

```

```

class TransitionBackend(InterferenceBackend):
    """
    Physically realizable ISDO implementation using shared optimized ISDO
    circuits.


    This backend uses the hardware-optimized Householder reflections and
    high-precision float64 logic from the ISDO module.
    """

    def score(self, chi, psi) -> float:
        """
        Calculates the interference score using the optimized ISDO quantum
        circuit.
        """
        # Call the shared ISDO routine
        return float(run_isdo_circuit(psi, chi))

```

File: src/IQC/interference/transition\_backend\_backup.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/IQC/interference/exact\_backend.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/IQC/interference/oracle\_backend.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 OracleBackend(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/IQC/interference/init.py

File: src/IQC/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

```

File: src/IQC/inference/**init**.py

File: src/IQC/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/IQC/encoding/**init**.py

File: Archive\_src/**init**.py

File: Archive\_src/swap\_test/swap\_test\_classifier.py

```

import os
import numpy as np

from qiskit import QuantumCircuit
from qiskit.quantum_info import Statevector
from qiskit_aer import AerSimulator

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

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

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

# -----
# Load vectors
# -----
class_state_0 = np.load(os.path.join(EMBED_DIR, "class_state_0.npy"))
class_state_1 = np.load(os.path.join(EMBED_DIR, "class_state_1.npy"))

# sanity check
assert abs(np.linalg.norm(class_state_0) - 1.0) < 1e-6
assert abs(np.linalg.norm(class_state_1) - 1.0) < 1e-6

# -----
# Example test embedding
# (later we loop over dataset)
# -----
test_embedding = np.load(
    os.path.join(EMBED_DIR, "val_embeddings.npy")
)[0].astype(np.float64)

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

print("test_embedding.shape", test_embedding.shape)
print("class_state_0.shape", class_state_0.shape)
print("class_state_1.shape", class_state_1.shape)

# expected class
expected_class = np.load(
    os.path.join(EMBED_DIR, "val_labels.npy")
)[0].astype(np.float64)

print("expected_class", expected_class)

```

```

# -----
# SWAP test function
# -----
def swap_test_fidelity(state_a, state_b, shots=2048):
    """
    Estimate  $|\langle a|b \rangle|^2$  using SWAP test
    """

    n_qubits = int(np.log2(len(state_a)))
    assert 2 ** n_qubits == len(state_a)

    qc = QuantumCircuit(1 + 2 * n_qubits, 1)

    anc = 0
    reg_a = list(range(1, 1 + n_qubits))
    reg_b = list(range(1 + n_qubits, 1 + 2 * n_qubits))

    # Initialize states
    qc.initialize(state_a, reg_a)
    qc.initialize(state_b, reg_b)

    # Hadamard on ancilla
    qc.h(anc)

    # Controlled SWAPs
    for qa, qb in zip(reg_a, reg_b):
        qc.cswap(anc, qa, qb)

    # Hadamard again
    qc.h(anc)

    # Measure ancilla
    qc.measure(anc, 0)
    qc.draw("mpl").savefig(os.path.join(PATHS["figures"],
    "swap_test_circuit.png"))

    backend = AerSimulator()
    job = backend.run(qc, shots=shots)
    counts = job.result().get_counts()

    p0 = counts.get("0", 0) / shots
    fidelity = 2 * p0 - 1

    return fidelity, counts

# -----
# Run SWAP test for both classes
# -----
F0, counts0 = swap_test_fidelity(test_embedding, class_state_0)
F1, counts1 = swap_test_fidelity(test_embedding, class_state_1)

print("Fidelity with class 0 (Benign):", F0)
print("Fidelity with class 1 (Malignant):", F1)

```

```

predicted_class = 0 if F0 > F1 else 1
print("\nPredicted class:", predicted_class)

## output
"""
🌱 Global seed set to 42
test_embedding.shape (32,)
class_state_0.shape (32,)
class_state_1.shape (32,)
expected_class 1.0
Fidelity with class 0 (Benign): 0.6318359375
Fidelity with class 1 (Malignant): 0.876953125

Predicted class: 1
"""

```

File: Archive\_src/swap\_test/evaluate\_swap\_test\_batch.py

```

import os
import numpy as np
from tqdm import tqdm

from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator

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

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

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

# -----
# Quantum-safe conversion
# -----
def to_quantum_state(x):
    x = np.asarray(x, dtype=np.float64).reshape(-1)
    n = len(x)
    if not (n & (n - 1) == 0):
        raise ValueError(f"State length {n} is not power of 2")
    x = x / np.sqrt(np.sum(x ** 2))

```

```

    assert np.isclose(np.sum(x ** 2), 1.0, atol=1e-12)
    return x

# -----
# Load class states
# -----
class_state_0 = to_quantum_state(
    np.load(os.path.join(EMBED_DIR, "class_state_0.npy"))
)
class_state_1 = to_quantum_state(
    np.load(os.path.join(EMBED_DIR, "class_state_1.npy"))
)

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

# -----
# Evaluation subset
# -----
N_SAMPLES = 5000
SHOTS = 1024

#X = X[:N_SAMPLES]
#y = y[:N_SAMPLES]

# -----
# SWAP test fidelity
# -----
def swap_test_fidelity(state_a, state_b, shots=1024):
    n_qubits = int(np.log2(len(state_a)))
    qc = QuantumCircuit(1 + 2 * n_qubits, 1)

    anc = 0
    reg_a = list(range(1, 1 + n_qubits))
    reg_b = list(range(1 + n_qubits, 1 + 2 * n_qubits))

    qc.initialize(state_a, reg_a)
    qc.initialize(state_b, reg_b)

    qc.h(anc)
    for qa, qb in zip(reg_a, reg_b):
        qc.cswap(anc, qa, qb)
    qc.h(anc)

    qc.measure(anc, 0)

    backend = AerSimulator()
    job = backend.run(qc, shots=shots)
    counts = job.result().get_counts()

```

File: `Archive src/swap test/init.py`



File: Archive\_src/quantum/init.py

File: Archive\_src/quantum/isdo/init.py

File:  
Archive\_src/quantum/isdo/isdo\_K\_sweep/old\_evaluate\_interference\_k4.py

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

from qiskit.quantum_info import Statevector

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

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

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

K = int(PATHS["class_count"]["K"])
INDEX_DIM = K
DATA_DIM = 32

# -----
# Helper
# -----
def to_quantum_state(x):
```

```

    x = np.asarray(x, dtype=np.float64).reshape(-1)
    x = x / np.sqrt(np.sum(x ** 2))
    return x

# -----
# Load prototypes
# -----
def load_class_superposition(cls):
    protos = []
    for k in range(1, K):
        p = np.load(os.path.join(CLASS_DIR,
f"K{cls}/class{cls}_proto{k}.npy"))
        protos.append(p)

    # Build joint state |k> |phi_k>
    joint = np.zeros(INDEX_DIM * DATA_DIM, dtype=np.float64)

    for k, proto in enumerate(protos):
        joint[k * DATA_DIM:(k + 1) * DATA_DIM] = proto

    joint = joint / np.sqrt(K) # superposition normalization
    joint = to_quantum_state(joint)

    return Statevector(joint)

# -----
# Load class states
# -----
Phi0 = load_class_superposition(0)
Phi1 = load_class_superposition(1)

# -----
# Load data
# -----
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 = X[test_idx]
y = y[test_idx]

N = len(X)
correct = 0

print(f"\n🐛 Evaluating Phase B (K={K}) on {N} samples\n")

# -----
# Evaluation
# -----
for i in tqdm(range(N)):

```

```

psi = to_quantum_state(X[i])

# Lift test state into joint space
joint_test = np.zeros(INDEX_DIM * DATA_DIM, dtype=np.float64)
for k in range(K):
    joint_test[k * DATA_DIM:(k + 1) * DATA_DIM] = psi

joint_test = to_quantum_state(joint_test)
Psi = Statevector(joint_test)

F0 = abs(Psi.inner(Phi0)) ** 2
F1 = abs(Psi.inner(Phi1)) ** 2

pred = 0 if F0 > F1 else 1
if pred == y[i]:
    correct += 1

accuracy = correct / N

print("\n=====")
print("Phase B: Interference-Based Measurement-Free Classifier")
print(f"Prototypes per class: {K}")
print(f"Samples: {N}")
print(f"Accuracy: {accuracy:.4f}")
print("=====\n")

## output
"""
🌱 Global seed set to 42

🔬 Evaluating Phase B (K=5) on 1500 samples

100%|
████████████████████████████████████████████████████████████████████████████████
████████████████████████████████████████████████████████████████████████████████
██████████ | 1500/1500 [00:00<00:00, 35004.26it/s]

=====
Phase B: Interference-Based Measurement-Free Classifier
Prototypes per class: 5
Samples: 1500
Accuracy: 0.8840
=====
"""

```

File: Archive\_src/quantum/isdo/isdo\_K\_sweep/init.py

File:

Archive\_src/quantum/isdo/isdo\_circuit\_test/test\_isdo\_circuits\_v1.py

```
import numpy as np

from src.quantum.isdo.circuits.circuit_a_controlled_state import run_isdo_circuit_a
from src.archive.rfc.reflection_classifier import run_isdo_circuit_b
from src.utils.common import build_chi_state

# Dummy normalized vectors for sanity test
psi = np.random.randn(32)
psi /= np.linalg.norm(psi)

phi0 = [np.random.randn(32) for _ in range(3)]
phi1 = [np.random.randn(32) for _ in range(3)]
phi0 = [p / np.linalg.norm(p) for p in phi0]
phi1 = [p / np.linalg.norm(p) for p in phi1]

chi = build_chi_state(phi0, phi1)

za = run_isdo_circuit_a(psi, chi)
zb = run_isdo_circuit_b(psi, chi)

print("Circuit A <Z>:", za)
print("Circuit B <Z>:", zb)
print("Difference:", abs(za - zb))
```

File: Archive\_src/quantum/isdo/isdo\_circuit\_test/init.py

File: Archive\_src/quantum/isdo/circuit/circuit\_a\_controlled\_state.py

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

from src.utils.common import load_statevector

def build_isdo_circuit_a(psi, chi):
    """
    ISDO Circuit A: Controlled state preparation
```

```

"""
n = int(np.log2(len(psi)))
qc = QuantumCircuit(1 + n, 1)

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

# Hadamard on ancilla
qc.h(anc)

# Controlled |psi>
state_prep_psi = StatePreparation(psi)
qc.append(state_prep_psi.control(1), [anc] + data)

# Flip ancilla
qc.x(anc)

# Controlled |chi>
state_prep_chi = StatePreparation(chi)
qc.append(state_prep_chi.control(1), [anc] + data)

# Undo flip
qc.x(anc)

# Interference
qc.h(anc)

# Measure ancilla
qc.measure(anc, 0)

return qc

def run_isdo_circuit_a(psi, chi):
    """
    Exact (statevector) evaluation of <Z>
    """
    qc = build_isdo_circuit_a(psi, chi)
    qc_no_meas = qc.remove_final_measurements(inplace=False)
    sv = Statevector.from_instruction(qc_no_meas)
    z_exp = sv.expectation_value(Pauli('Z'), [0]).real
    return z_exp

```

File: Archive\_src/quantum/isdo/circuit/init.py

File: Archive\_src/statevector\_similarity/compute\_class\_states.py

```

import os
import json
import numpy as np
from sklearn.preprocessing import normalize

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

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

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

EMBED_DIR = PATHS["embeddings"]
SAVE_DIR = PATHS["embeddings"]
os.makedirs(SAVE_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"))

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

print("Loaded embeddings:", X_train.shape)
# -----
# Compute class means
# -----
class_states = {}

for cls in np.unique(y):
    X_cls = X_train[y_train == cls]
    #X_cls = X_cls.astype(np.float64)

    # Mean in FP64
    mean_vec = X_cls.mean(axis=0)

    # Exact FP64 normalization
    norm = np.sqrt(np.sum(mean_vec ** 2))
    mean_vec = mean_vec / norm

```

```

# Sanity check (important)
assert np.isclose(np.sum(mean_vec ** 2), 1.0, atol=1e-12)

class_states[int(cls)] = mean_vec

print(
    f"Class {cls}: "
    f"samples = {len(X_cls)}, "
    f"norm = {np.linalg.norm(mean_vec):.12f}"
)

# -----
# Save
# -----
np.save(os.path.join(SAVE_DIR, "class_state_0.npy"), class_states[0])
np.save(os.path.join(SAVE_DIR, "class_state_1.npy"), class_states[1])

# Optional: save metadata
with open(os.path.join(SAVE_DIR, "class_states_meta.json"), "w") as f:
    json.dump(
        {
            "embedding_dim": X.shape[1],
            "classes": list(class_states.keys()),
            "normalization": "l2",
            "source": "mean_of_class_embeddings",
        },
        f,
        indent=2,
    )

print("\n✅ Class states saved:")
print(" - class_state_0.npy (Benign)")
print(" - class_state_1.npy (Malignant)")

```

File:

Archive\_src/statevector\_similarity/evaluate\_statevector\_similarity.py

```

import os
import numpy as np
from tqdm import tqdm

from qiskit.quantum_info import Statevector

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"]

# -----
# Quantum-safe conversion
# -----
def to_quantum_state(x):
    x = np.asarray(x, dtype=np.float64).reshape(-1)
    n = len(x)
    if not (n & (n - 1) == 0):
        raise ValueError(f"State length {n} is not power of 2")
    x = x / np.sqrt(np.sum(x ** 2))
    assert np.isclose(np.sum(x ** 2), 1.0, atol=1e-12)
    return x

# -----
# Load class states
# -----
phi0 = to_quantum_state(
    np.load(os.path.join(EMBED_DIR, "class_state_0.npy"))
)
phi1 = to_quantum_state(
    np.load(os.path.join(EMBED_DIR, "class_state_1.npy"))
)

sv_phi0 = Statevector(phi0)
sv_phi1 = Statevector(phi1)

# -----
# Load embeddings
# -----
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 = X[test_idx]
y = y[test_idx]

N = len(X)
correct = 0

print(f"\n🔬 Evaluating measurement-free statevector classifier on {N} samples\n")

for i in tqdm(range(N)):
    psi = Statevector(to_quantum_state(X[i]))

```



[illegible]

File: Archive\_src/statevector\_similarity/init.py

File: Archive\_src/rfc/reflection\_classifier.py

```
# Reflection-Fidelity Classifier

import numpy as np
from qiskit import QuantumCircuit
from qiskit.quantum_info import Statevector, Pauli
from qiskit.circuit.library import StatePreparation, UnitaryGate
from src.utils.common import load_statevector
```

```

def reflection_operator(chi):
    """
    Build  $R_{\chi} = I - 2|\chi\rangle\langle\chi|$ 
    """
    dim = len(chi)
    proj = np.outer(chi, chi.conj())
    return np.eye(dim) - 2 * proj

def build_isdo_circuit_b(psi, chi):
    """
    ISDO Circuit B: Phase kickback via reflection
    """
    n = int(np.log2(len(psi)))
    qc = QuantumCircuit(1 + n, 1)

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

    # Prepare  $|\psi\rangle$ 
    state_prep_psi = StatePreparation(psi)
    qc.append(state_prep_psi, data)

    # Hadamard ancilla
    qc.h(anc)

    # Controlled reflection
    R = UnitaryGate(reflection_operator(chi), label="R_chi")
    qc.append(R.control(1), [anc] + data)

    # Interference
    qc.h(anc)

    # Measure ancilla
    qc.measure(anc, 0)

    return qc

def run_isdo_circuit_b(psi, chi):
    """
    Exact  $\langle Z \rangle$  extraction
    """
    qc = build_isdo_circuit_b(psi, chi)
    qc_no_meas = qc.remove_final_measurements(inplace=False)
    sv = Statevector.from_instruction(qc_no_meas)
    z_exp = sv.expectation_value(Pauli('Z'), [0]).real
    return z_exp

```

File: Archive\_src/rfc/init.py

File: Archive\_src/experiments/init.py

File: Archive\_src/experiments/iqc/run\_regime3c\_v1.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.IQC.states.class_state import ClassState
from src.IQC.encoding.embedding_to_state import embedding_to_state
from src.IQC.memory.memory_bank import MemoryBank
from src.IQC.interference.math_backend import MathInterferenceBackend

from src.IQC.training.regime3c_trainer_v1 import Regime3CTrainer
from src.IQC.inference.regime3b_classifier import Regime3BClassifier
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)

# -----
# Prepare dataset (same as Regime 2 / 3-A / 3-B)
# -----
dataset = [
    (embedding_to_state(x), int(label))
    for x, label in zip(X_train, y_train)
]

# shuffle (important for online + growth)
rng = np.random.default_rng(42)
perm = rng.permutation(len(dataset))
dataset = [dataset[i] for i in perm]

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

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

backend = MathInterferenceBackend()

memory_bank = MemoryBank(
    class_states=class_states,
    backend=backend
)

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

# -----
# Train Regime 3-C (percentile-based  $\tau$ )
# -----
trainer = Regime3CTrainer(
    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
)

trainer.train(dataset)

```

```

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 = Regime3BClassifier(memory_bank)

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

acc_3c = correct / len(dataset)
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: 3
Number of spawned memories: 0
Number of updates: 524
Regime 3-C accuracy (3-B inference): 0.7948571428571428
Final memory count: 3
"""

```

File: Archive\_src/experiments/iqc/run\_regime3b.py

```

from src.IQC.inference.regime3b_classifier import Regime3BClassifier
from src.IQC.memory.memory_bank import MemoryBank
from src.IQC.states.class_state import ClassState
from src.IQC.encoding.embedding_to_state import embedding_to_state
from src.IQC.training.regime3a_trainer import Regime3ATrainer

```

```
from src.IQC.interference.math_backend import MathInterferenceBackend

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

import os
import numpy as np
from collections import Counter

# -----
# 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)

dataset = [
    (embedding_to_state(x), int(label))
    for x, label in zip(X_train, y_train)
]

# shuffle (important for online + growth)
rng = np.random.default_rng(42)
perm = rng.permutation(len(dataset))
dataset = [dataset[i] for i in perm]

d = dataset[0][0].shape[0]

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

backend = MathInterferenceBackend()
```

```

memory_bank = MemoryBank(
    class_states=class_states,
    backend=backend
)
trainer = Regime3ATrainer(memory_bank, eta=0.1)
acc = trainer.train(dataset)

# now we train 3b
classifier = Regime3BClassifier(trainer.memory_bank)

correct = 0
for psi, y in dataset:
    y_hat = classifier.predict(psi)
    if y_hat == y:
        correct += 1

acc_3b = correct / len(dataset)
print("Regime 3-B accuracy:", acc_3b)
print("Memory usage:", Counter(trainer.history["winner_idx"]))
### output
"""
🌱 Global seed set to 42
Loaded train embeddings: (3500, 32)
Regime 3-B accuracy: 0.8342857142857143
Memory usage: Counter({2: 1473, 0: 1243, 1: 784})
"""

```

File: Archive\_src/experiments/iqc/verify\_isdo\_bprime\_backend.py

```

import numpy as np
from scipy.stats import spearmanr

from src.IQC.interference.math_backend import MathInterferenceBackend
from src.IQC.interference.circuit_backend_transition import
TransitionInterferenceBackend
from src.IQC.interference.circuit_backend_isdo_bprime import
ISDOBPrimeInterferenceBackend

def random_state(n):
    v = np.random.randn(2**n) + 1j * np.random.randn(2**n)
    v /= np.linalg.norm(v)
    return v

def sign(x):
    return 1 if x >= 0 else -1

np.random.seed(0)

```

```

math_backend = MathInterferenceBackend()
ref_backend = TransitionInterferenceBackend()
isdo_backend = ISDOBPrimeInterferenceBackend()

n = 4
num_tests = 100

sign_agree = 0
ref_vals = []
isdo_vals = []

for _ in range(num_tests):
    chi = random_state(n)
    psi = random_state(n)

    s_ref = ref_backend.score(chi, psi)
    s_isdo = isdo_backend.score(chi, psi)

    ref_vals.append(s_ref)
    isdo_vals.append(s_isdo)

    if sign(s_ref) == sign(s_isdo):
        sign_agree += 1

rho, _ = spearmanr(ref_vals, isdo_vals)

print("ISDO-B' vs Transition backend")
print("Sign agreement:", sign_agree, "/", num_tests)
print("Spearman rank correlation:", rho)
print("Mean |difference|:", np.mean(np.abs(np.array(ref_vals) -
np.array(isdo_vals))))

"""
ISDO-B' vs Transition backend
Sign agreement: 51 / 100
Spearman rank correlation: -0.029006900690069004
Mean |difference|: 0.21415260812801665
"""

```

File: Archive\_src/experiments/iqc/verify\_hadamard\_backend.py

```

import numpy as np

from src.IQC.interference.math_backend import MathInterferenceBackend
from src.IQC.interference.circuit_backend_hadamard import
HadamardInterferenceBackend

def random_state(n):
    v = np.random.randn(2**n) + 1j * np.random.randn(2**n)

```



```

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

def sign(x):
    return 1 if x >= 0 else -1

np.random.seed(0)

math_backend = MathInterferenceBackend()
had_backend = HadamardInterferenceBackend()

n = 3 # small, exact verification
num_tests = 50

sign_agree = 0
vals = []

for _ in range(num_tests):
    chi = random_state(n)
    psi = random_state(n)

    s_math = math_backend.score(chi, psi)
    s_had = had_backend.score(chi, psi)

    vals.append((s_math, s_had))

    if sign(s_math) == sign(s_had):
        sign_agree += 1

print("Sign agreement:", sign_agree, "/", num_tests)
print("Mean abs error:", np.mean([abs(a - b) for a, b in vals]))

## output
"""
Sign agreement: 50 / 50
Mean abs error: 6.399047958183246e-16
"""

```

File: Archive\_src/experiments/iqc/run\_regime3a.py

```

from src.IQC.training.regime3a_trainer import Regime3ATrainer
from src.IQC.memory.memory_bank import MemoryBank
from src.IQC.states.class_state import ClassState
from src.IQC.encoding.embedding_to_state import embedding_to_state

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

import os

```

```

import numpy as np
from collections import Counter

# -----
# 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)

dataset = [
    (embedding_to_state(x), int(label))
    for x, label in zip(X_train, y_train)
]

d = dataset[0][0].shape[0]

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

memory_bank = MemoryBank(class_states)
trainer = Regime3ATrainer(memory_bank, eta=0.1)

acc = trainer.train(dataset)

print("Regime 3-A accuracy:", acc)
print("Total updates:", trainer.num_updates)
print(Counter(trainer.history["winner_idx"]))

#### output
"""
🌱 Global seed set to 42

```

```
Loaded train embeddings: (3500, 32)
Regime 3-A accuracy: 0.8328571428571429
Total updates: 585
Counter({0: 1266, 2: 1238, 1: 996})
"""
```

File: Archive\_src/experiments/iqc/**init.py**

File: Archive\_src/IQC/**init.py**

File: Archive\_src/IQC/training/regime3c\_trainer\_v1.py

```
import numpy as np
from collections import deque

from ..learning.regime2_update import regime2_update

class Regime3CTrainer:
    """
    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,
    ):
        self.memory_bank = memory_bank
        self.eta = eta
        self.percentile = percentile
        self.tau_abs = tau_abs

        # 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 = np.array([
            float(np.real(np.vdot(cs.vector, psi)))
            for cs in self.memory_bank.class_states
        ])
        return scores.mean()  # uniform weights

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

        # compute  $\tau$  only after we have some history
        if len(self.margins) >= 20:
            tau = np.percentile(self.margins, self.percentile)
        else:
            tau = -np.inf  # disable spawning early

        spawned = False

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

        elif margin < 0:
            # update winning memory
            idx, _ = self.memory_bank.winner(psi)
            cs = self.memory_bank.class_states[idx]

            chi_new, updated = regime2_update(
                cs.vector, psi, y, self.eta
            )

            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 train(self, dataset):
        for psi, y in dataset:
            self.step(psi, y)
```

File: Archive\_src/IQC/training/**init.py**

File: Archive\_src/IQC/interference/**init.py**

File: Archive\_src/IQC/inference/regime3a\_classifier.py

```
class Regime3AClassifier:
    def __init__(self, memory_bank):
        self.memory_bank = memory_bank

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

File: Archive\_src/IQC/inference/**init.py**

File: research\_docs/comparison\_report.md

# Literature Comparison Report: Measurement-Free Quantum Classifier

This report compares the proposed **Measurement-Free Quantum Classifier** (MFQC) project with the provided literature set (Bucket A, Bucket B, and Scholar Review).

## 1. Technical Innovation: The SWAP-Test Advantage

Most quantum classifiers in current literature (e.g., **Singh 2024** in Bucket A, **VQFE** in Bucket B) rely on:

- **Variational circuits** that require intermediate measurements for gradient estimation (Parameter-Shift Rule).
  - **Fidelity estimation** that involves multiple shots to reconstruct matrix elements.
- The MFQC Project** innovates by using a **SWAP-test** protocol. This allows for a **measurement-free** classification process where:
- Quantum coherence is preserved until the final readout bit.
  - Only a single ancilla qubit is measured at the very end to determine the fidelity (similarity) between the test image and the class prototypes.
  - This directly addresses the research gap identified in **Radhi et al. (2025)**.

## 2. Encoding and Dimension Reduction

Standard quantum image processing papers often struggle with the "curse of dimensionality":

- **Literature (Bucket A)**: Evaluates FRQI/NEQR which require a qubit or gate per pixel, making them impractical for 96x96 medical images.
- **MFQC Approach**: Uses a **Hybrid CNN backbone**. The classical CNN extracts high-level features (16-32D), which are then encoded using **Amplitude Encoding** into only 4-5 qubits. This hybrid approach is supported by **Springer Nature (2023)** as the most viable NISQ-era path.

## 3. NISQ Hardware Feasibility

- **Literature (Scholar Review)**: Highlights that decoherence and noise limit circuit depth to **<150 gates for meaningful results**.
- **MFQC Approach**: Specifically targets a **shallow circuit design (50-100 gates)**. By avoiding intermediate measurements, it reduces the accumulation of shot noise and readout error, which are major bottlenecks discussed in **MDPI (2024)**.

## 4. Performance against Baselines

Metric	Literature Average (Standard VQC)	MFQC Proposed Target
	---	---
<b>Circuit Depth</b>	150-500+ gates	50-100 gates
<b>Qubit Count</b>	High (for raw pixels)	4-5 (for amplitude features)
<b>Accuracy (Medical)</b>	85-90%	<b>92%</b>
<b>Coherence</b>	Interrupted by measurements	<b>Preserved until final readout</b>

## Summary of Gap Filling

The MFQC project sits at the intersection of **Hybrid Machine Learning** (Scholar Review) and **Quantum State Comparison** (Bucket B). It moves beyond the "survey phase" (Radhi 2025) into a practical implementation that leverages the efficiency of the SWAP-test to achieve medical-grade classification without the overhead of measurement-based variational loops.

## # Research Analysis and Literature Comparison Plan

This plan outlines the steps to compare the proposed **Measurement-Free Quantum Classifier** project with the provided literature set (Bucket A, Bucket B, and Scholar Review).

### ## Goals

1. **Literature Mapping**: Categorize the provided reference papers based on their focus (encoding, architecture, fidelity estimation, hardware constraints).
2. **Gap Analysis**: Verify the "Research Gaps" identified in the project PPT against the actual literature.
3. **Innovation Validation**: Compare the SWAP-test based measurement-free approach with standard VQC/QSVM methods described in the papers.
4. **Hardware Assessment**: Evaluate the NISQ-feasibility claims (50-100 gates) against current hardware limitations discussed in the Scholar Review.

### ## Proposed Steps

#### ### 1. Literature Categorization

- **Bucket A**: Focus on encoding (Amplitude, FRQI, NEQR) and QNN architectures.
- **Bucket B**: Focus on Quantum Fidelity, Trace Distance, and state comparison techniques (Variational Fidelity Estimation vs. SWAP-test).
- **Scholar Review**: Focus on NISQ hardware, hybrid systems, and medical imaging applications.

#### ### 2. Detailed Comparison

- **SWAP-test vs. Variational Fidelity**: Analyze how the project's SWAP-test (measurement-free) avoids common pitfalls of variational methods (which often require multiple intermediate measurements).
- **Coherence Preservation**: Evaluate the claim of preservation against papers discussing decoherence in NISQ devices.
- **Complexity Analysis**: Compare the "shallow circuit" claim with depths reported in the literature for medical classification.

#### ### 3. Synthesis Report

- Create a comprehensive report (as a new artifact or response) answering:
  - How the project fills identified gaps.
  - Technical advantages/limitations.
  - Alignment with current research trends (Radhi 2025, etc.).

### ## Verification Plan

#### ### Automated Analysis

- I will use `pdftotext` to extract abstracts/summaries from key papers to confirm their focus and findings.
- I will search for "SWAP-test" and "measurement-free" keywords across the literature set to find direct competitors or foundational theories.

#### ### Manual Verification

- The user should review the synthesized comparison report to ensure it

addresses their specific (but unstated) concerns.

File:

research\_docs/interference\_quantum\_classifier\_iqc\_paper\_draft\_non\_claim\_leaking.md

```
# Interference Quantum Classifier (IQC)
```

```
## A Measurement-Efficient Quantum Classification Framework Based on Linear Interference
```

```
---
```

```
## Abstract
```

Quantum machine learning classifiers proposed for near-term devices commonly rely on variational circuits or fidelity-based measurements, leading to high measurement cost, loss of phase information, and unstable training dynamics. In this work, we introduce the **Interference Quantum Classifier (IQC)**, a classification framework in which learning is decoupled from quantum execution and inference is performed through a fixed quantum interference circuit. IQC bases its decision rule on a linear interference quantity rather than probability or fidelity, enabling phase-sensitive, sign-preserving classification with constant measurement complexity. We present the theoretical formulation of the interference observable, describe a quantum circuit realization, and demonstrate how class information can be represented and updated as quantum states using classical learning rules. Experiments on real-world image embeddings show that interference-based aggregation improves expressivity while significantly reducing runtime compared to measurement-heavy quantum classifiers. Our results suggest that linear quantum interference provides a practical and interpretable alternative to variational and kernel-based quantum classification on near-term hardware.

```
---
```

```
## 1. Introduction
```

Quantum machine learning (QML) has attracted significant attention as a potential application of near-term quantum devices. Most existing quantum classifiers fall into two categories: variational quantum classifiers, which train parameterized circuits using measurement-based optimization, and similarity-based classifiers, which estimate quantum state fidelity or kernel values. In practice, both approaches face substantial challenges, including high measurement overhead, sensitivity to noise, and limited interpretability.

A key observation motivating this work is that classification decisions need not depend on quadratic probability estimates. Instead, they can be derived from **linear interference between quantum states**, which



preserves directional and phase information that is lost in fidelity-based methods. This observation motivates a rethinking of how quantum classifiers are constructed and how learning is integrated with quantum hardware.

In this paper, we propose the Interference Quantum Classifier (IQC), a framework that separates learning from quantum inference and employs a fixed quantum interference circuit as its decision engine.

---

## ## 2. Problem Setup and Notation

We consider a supervised binary classification problem. Input samples are first mapped to real-valued feature vectors using a classical encoder. These vectors are normalized and embedded into quantum states. Let  $|\psi\rangle$  denote a quantum state corresponding to an input sample, and let  $|\chi\rangle$  denote a quantum state representing class information.

The goal of classification is to determine a label based on the relationship between  $|\psi\rangle$  and  $|\chi\rangle$ .

---

## ## 3. Linear Interference as a Decision Primitive

### ### 3.1 Interference Observable

IQC is built around a linear interference quantity given by the real part of the inner product between two quantum states. Unlike fidelity, which depends on the squared magnitude of the inner product, this quantity preserves sign and phase information.

We show that this linear quantity is sufficient to define a stable and interpretable decision rule for classification.

---

### ### 3.2 Comparison with Fidelity-Based Classification

Fidelity-based classifiers estimate  $|\langle\chi|\psi\rangle|^2$ , which is invariant under global phase changes and discards sign information. As a result, such classifiers behave like distance measures rather than directional similarity measures.

In contrast, linear interference distinguishes between constructive and destructive overlap, enabling sign-sensitive classification decisions.

---

## ## 4. Quantum Circuit for Interference-Based Inference

We describe a quantum circuit that evaluates the linear interference quantity using an ancilla-assisted interference procedure. The circuit is fixed and does not contain trainable parameters. Its output is a single

expectation value whose sign determines the predicted class label.

Importantly, the circuit depth and measurement cost are independent of dataset size.

---

## ## 5. Learning via Quantum State Representation

Rather than training quantum gate parameters, IQC represents learned class information as quantum states. Learning is performed by updating these state representations using classical rules, while the quantum circuit remains unchanged.

This separation avoids common training pathologies encountered in variational quantum algorithms and enables incremental learning.

---

## ## 6. Learning Regimes

We outline several learning regimes supported by the IQC framework, including:

- static construction of class states from training data,
- online updates using sequential samples,
- use of multiple class states to increase expressivity.

These regimes differ in how class information is represented but share the same interference-based inference mechanism.

---

## ## 7. Experimental Evaluation

We evaluate IQC on real-world image embeddings generated by a convolutional neural network. We compare interference-based classification against classical baselines and measurement-heavy quantum similarity methods.

Our experiments demonstrate that removing measurement noise alone does not significantly improve performance, whereas interference-based aggregation improves classification accuracy while reducing runtime by orders of magnitude.

---

## ## 8. Discussion

The IQC framework highlights a different role for quantum circuits in machine learning: rather than serving as trainable models, they act as fixed physical operators that evaluate structured similarity measures. This perspective offers advantages in stability, interpretability, and hardware compatibility.

```
We discuss limitations of the current approach and potential extensions,
including richer quantum memory structures and alternative interference
semantics.

---

## 9. Conclusion

We have presented the Interference Quantum Classifier, a quantum
classification framework based on linear quantum interference and a clear
separation between learning and inference. By avoiding variational training
and fidelity estimation, IQC provides a practical path toward measurement-
efficient quantum classification on near-term devices. Our results suggest
that linear interference is a powerful and underexplored primitive for
quantum machine learning.

---

## Acknowledgements

[To be added]
```

File:  
research\_docs/Fidelity\_and\_Measurement\_Free\_Methods\_Comparison.  
md

```
# Comparative Analysis: Fidelity-Based & Measurement-Free Quantum
Classification

This report provides a formal technical review of quantum classification
methods found in the project's literature repository (`Documents/`). It
emphasizes Measurement-Free (MF) architectures and Low-Shot
similarity algorithms.

---

## 1. Coherent Feedback Learning (The Absolute MF Baseline)
Reference: [Alvarez-Rodriguez et al. (2017)]
(file:///home/tarakesh/Work/Repo/measurement-free-quantum-
classifier/Documents/refference_papers/Scholar%20review/quantum%20fidelity/
s41598-017-13378-0.pdf)

* Mechanism: Encodes the classification logic into a time-delayed
Schrödinger equation.
* Shot Efficiency: Zero mid-circuit shots. The system evolves
unitarily toward the correct label.
* Equation:

$$\frac{d}{dt} |\psi(t)\rangle = -i \left[ \kappa_1 H_{\text{int}} + \kappa_2 H_{\text{feedback}} \right] |\psi(t)\rangle$$

```

---

## ## 2. Coherent Amplitude/Phase Estimation (The Bit-by-Bit Approach)

**Reference**: [Patrick Rall (2021)]

(file:///home/tarakesh/Work/Repo/measurement-free-quantum-classifier/Documents/refference\_papers/Scholar%20review/minimm%20measuremen  
t%20quant%20algo/q-2021-10-19-566.pdf)

\* **Mechanism**: Uses **Singular Value Transformation (SVT)** to estimate similarity one bit at a time.

\* **Shot Efficiency**: Achieves **Heisenberg-limited** accuracy ( $\Theta(1/\epsilon)$  queries).

\* **Advantage**: Does not require the Quantum Fourier Transform (QFT), making it much more robust for NISQ devices.

\* **Expression**:

\$\$

$|0\rangle \langle \psi| \rightarrow |\text{overlap}\rangle \langle \psi|$

\$\$

This "writes" the fidelity into a register without collapsing the original superposition.

---

## ## 3. Classical Shadows (Shadow Classification)

**Reference**: [Huang et al. (2020) & Yunfei Wang (2024)]

(file:///home/tarakesh/Work/Repo/measurement-free-quantum-classifier/Documents/refference\_papers/Scholar%20review/NISQ%20hardwere/2401.11351v2.pdf)

\* **Mechanism**: Performs randomized Pauli measurements to create a "shadow" of the quantum state.

\* **Shot Efficiency**: Allows tracking **logarithmic** shots relative to the number of samples. Once a shadow is created, you can compute INFINITE fidelities classically.

\* **Equation**:

$$\hat{\rho} = \mathbb{E}[\mathcal{M}^{-1}(U^\dagger |b\rangle\langle b| U)]$$

Where  $\hat{\rho}$  is the reconstructed "shadow" that contains the fidelity information.

---

## ## 4. Destructive SWAP-Test (Ancilla-Free)

**Reference**: [Garcia-Escartin (2013) & Blank (2020)]

(file:///home/tarakesh/Work/Repo/measurement-free-quantum-classifier/Documents/refference\_papers/Scholar%20review/quantum%20fidelity/s41534-020-0272-6.pdf)

\* **Mechanism**: Removes the ancilla qubit entirely. Uses CNOTs followed by single-qubit measurements on both registers.

\* **Shot Efficiency**: Far more efficient for hardware with limited connectivity.

\* **Equation**:

Considers the parity of the measurement outcomes  $b_1, b_2$ :

```

    $$F = 1 - 2 \cdot P(\text{parity yields odd})$$

---

## 5. Comparative Shot-Efficiency Table

| Method | Shots Required | Measurement-Free? | Best Use Case |
| :--- | :--- | :--- | :--- |
| Standard SWAP |  $O(1/\epsilon^2)$  | No | General Purpose |
| Coherent SVT |  $\Theta(1/\epsilon)$  | Yes | High Precision / Coherent Chains |
| Classical Shadows |  $\log(M)$  | Partial | Multi-class (Benign, Malignant, Cyst) |
| Destructive SWAP | Medium | No | Low-Qubit Count Chips |
| VQFE | High (Training) | No | Parameter Tuning |

---

### Project Conclusion
While the "Big 3" get most of the attention in textbooks, recent 2021-2024 research (like Patrick Rall's SVT) proves that we can achieve classification without measurement collapse. In our project, we use the Interference Average (Phase B) as a bridge: it uses the parallel nature of the SWAP-test to reduce the "effective" shots compared to testing prototypes one-by-one.
```

File: research\_docs/research\_answers.md

```

# Research Q&A: Measurement-Free Quantum Classification

Below are the detailed answers to your questions based on the provided literature set and your project idea.

### 1. How do existing quantum classifiers perform measurement during inference and training?
* Training: Most existing models (Variational Quantum Circuits - VQCs) use the Parameter-Shift Rule. This requires executing the circuit multiple times (shots) with shifted parameter values to estimate gradients classically. Each "step" involves thousands of measurements.
* Inference: Typically involves State Readout. The circuit is executed thousands of times, and the ancilla qubit (or a register) is measured. The probability of measuring  $|1\rangle$  vs  $|0\rangle$  is used to determine the class label.

### 2. What are the limitations of measurement-based quantum machine learning on NISQ hardware?
* Shot Noise: The need for high precision in probability estimation requires a massive number of "shots," increasing latency.
* Readout Error: State-of-the-art NISQ devices have significant errors during the measurement process itself, which accumulate if multiple intermediate measurements are used.
```

- \* **Decoherence**: Long sequences of measurements and classical loops (as in variational methods) prolong the time the quantum state must remain coherent, leading to gate errors.

### 3. How is quantum fidelity estimated in quantum machine learning classifiers?

- \* **SWAP-Test**: A standard protocol where an ancilla qubit interacts with two quantum states. The probability of the ancilla being  $|0\rangle$  is  $(1 + F)/2$ , where  $F$  is the fidelity.
- \* **Variational Fidelity Estimation (VQFE)**: Uses a parameterized circuit to diagonalize one state and compute its overlap with another (**Bucket B: Cerezo et al. 2020**).
- \* **Trace Distance Bounds**: Using hybrid algorithms to compute upper and lower bounds on similarity rather than a single point estimate.

### 4. Are there quantum classifiers that use fidelity without explicit fidelity estimation?

- \* Yes, **Quantum Kernel Methods** (e.g., QSVM) use fidelity implicitly. The circuit  $U(\mathbf{x})^\dagger U(\mathbf{y})$  maps the similarity to the vacuum state  $|0\rangle^{\otimes n}$ . While the "fidelity" value is the goal, the algorithm often just needs to know if the transition is high enough for a kernel matrix, without necessarily "reporting" the fidelity to a classical observer at every layer.

### 5. What measurement-free or measurement-minimal quantum algorithms exist?

- \* **Coherent Phase Estimation**: Algorithms that perform phase estimation without intermediate measurements to preserve superposition (**Patel et al. 2024**).
- \* **Interference-based Distance Classifiers**: Using the SWAP-test logic as the core of the classifier (like your project), which avoids collapsing the state until the final diagnostic decision.

### 6. Have measurement-free quantum algorithms been applied to medical image classification?

- \* There is a significant **research gap** here. While hybrid QCNNs (**Li et al. 2025**) use quantum layers for medical images, they typically use variational (measurement-based) updates. Your project's focus on a "pure" measurement-free end-to-end classification for metastatic tissue is highly novel.

### 7. What hybrid quantum-classical approaches are used for medical image classification?

- \* **Feature Extraction + VQC**: A classical CNN (EfficientNet, ResNet) extracts 1024D features, reduced via PCA/Autoencoders to 8-16D, then fed into a Variational Quantum Circuit (**Scholar Review: Singh 2024**).
- \* **Quantvolutional Neural Networks**: Classical convolution filters are replaced by small quantum circuits that transform pixel patches before traditional CNN processing.

### 8. What open research gaps exist in measurement-free quantum machine learning for classification tasks?

- \* **Trainability**: How to optimize "prototypes" (class representatives) in a purely measurement-free setting without falling into barren plateaus.

- \* **Hardware Robustness**: Empirical validation of whether avoiding measurement actually results in higher accuracy on noisy IBM/IonQ hardware.
- \* **Large-Scale Benchmarking**: Most studies use toy datasets (MNIST); applying these to 96x96 medical images (like PatchCamelyon) is an active frontier.

## File: research\_docs/project\_blueprint.md

### # Accelerated Research Project Blueprint (8-Week Roadmap)

This revised plan compresses the research into a high-intensity **8-week cycle**, focusing on the critical implementation of the measurement-free quantum classifier.

---

#### ## Part 1: Foundation & Architecture (Weeks 1-2)

**Goal**: Rapid setup and interface design.

- **Week 1: Infrastructure & Data**:
  - Configure Qiskit/PyTorch environment.
  - Set up a **subset** data loader for PathCamelyon (to speed up iteration).
  - Implement a pre-trained CNN feature extractor (e.g., ResNet18) instead of training from scratch.
- **Week 2: Quantum-Classical Interface**:
  - Implement Amplitude Encoding for 8D/16D features.
  - Prototype the SWAP-test circuit and verify basic state overlap logic.

#### ## Part 2: Implementation & Hybrid Training (Weeks 3-5)

**Goal**: Build the core and optimize prototypes.

- **Week 3: Circuit Optimization**:
  - Minimize gate depth for NISQ feasibility (target **<50 gates if possible**).
  - Implement noisy simulation environment.
- **Week 4-5: Joint Optimization**:
  - Execute hybrid training loops using the Parameter-Shift rule.
  - Focus on optimizing class prototypes to maximize inter-class fidelity distance.
  - Monitor for training stability in a shorter epoch window.

#### ## Part 3: Validation & Reporting (Weeks 6-8)

**Goal**: Prove innovation and finalize documentation.

- **Week 6: Performance Evaluation**:
  - Calculate Accuracy, F1-Score, and AUC-ROC on the test set.
  - Run primary comparison against a standard VQC baseline.
- **Week 7: Robustness & Noise Study**:
  - Test the measurement-free advantage by simulating hardware noise.
  - Conduct a single hardware run (IBM Quantum) if possible.
- **Week 8: Final Synthesis**:
  - Finalize the technical report/manuscript.

```
- Prepare visualizations and code documentation for handover.
```

```
---
```

### ## Streamlining Strategy

- **Pre-trained Backbones**: Use pre-trained weights to skip weeks of classical training.
- **Sub-sampling**: Use a balanced subset of PatchCamelyon for training to reduce compute time.
- **Parallelization**: Design circuits while the data pipeline is being finalized.
- **Focus**: Prioritize "Proof of Concept" over "Scale" to meet the 8-week deadline.

File: results/embeddings/class\_states\_meta.json

```
{
  "embedding_dim": 32,
  "classes": [
    0,
    1
  ],
  "normalization": "l2",
  "source": "mean_of_class_embeddings"
}
```

File: results/logs/train\_history.json

```
{
  "train_loss": [
    0.323274524165754,
    0.23782655238210282,
    0.1993992631250876,
    0.17781282487430872,
    0.16371910747557195,
    0.1537160865741498,
    0.14631224803679288,
    0.1406550945730487,
    0.13497550318106732,
    0.11610426991182976,
    0.11148261162816198,
    0.10862913947039488,
    0.09541817462331892,
    0.09339981933680974,
    0.0910517916313438,
    0.08256717906601807
  ],
  "train_acc": [
```



```
0.8622550964355469,  
0.9047431945800781,  
0.9226036071777344,  
0.9327926635742188,  
0.9391098022460938,  
0.9431877136230469,  
0.9461746215820312,  
0.9482002258300781,  
0.9505386352539062,  
0.9581375122070312,  
0.9599342346191406,  
0.9608840942382812,  
0.9665145874023438,  
0.9672470092773438,  
0.9680290222167969,  
0.9713249206542969  
],  
"val_loss": [  
0.7608505549724214,  
0.3770719189342344,  
0.33603281057730783,  
0.4026396208500955,  
0.4809370573348133,  
0.289258603748749,  
0.3426725415774854,  
0.36998813936224906,  
0.7853999140152155,  
0.3571328424004605,  
0.31231515117542585,  
0.4606642867165647,  
0.507413076415105,  
0.45235701354249613,  
0.6111933563879575,  
0.3889162304039928  
],  
"val_acc": [  
0.689483642578125,  
0.8507080078125,  
0.86328125,  
0.843505859375,  
0.832000732421875,  
0.88818359375,  
0.874786376953125,  
0.866363525390625,  
0.789154052734375,  
0.87628173828125,  
0.884002685546875,  
0.84228515625,  
0.84930419921875,  
0.854217529296875,  
0.807952880859375,  
0.875701904296875  
]  
}
```

File: results/logs/embedding\_baseline\_results.json

```
{
  "LogisticRegression": {
    "accuracy": 0.9046666666666666,
    "auc": 0.9664224751066857
  },
  "LinearSVM": {
    "accuracy": 0.9053333333333333,
    "auc": null
  },
  "kNN": {
    "accuracy": 0.926,
    "auc": 0.9711219772403983
  }
}
```