

Project Summary

Directory Structure

```
measurement-free-quantum-classifier/  
  scripts_Classical/  
    make_embedding_split.py  
    train_embedding_models.py  
    extract_embeddings.py  
    visualize_embeddings.py  
    train_cnn.py  
    visualize_pcam.py  
  configs/  
    paths.yaml  
  src/  
    __init__.py  
    utils/  
      common.py  
      paths.py  
      seed.py  
      __init__.py  
    data/  
      pcam_loader.py  
      transforms.py  
      __init__.py  
    quantum/  
      __init__.py  
      isdo/  
        evaluate_isdo.py  
        isdo_classifier.py  
        __init__.py  
        isdo_K_sweep/  
          evaluate_isdo_k_sweep.py  
          old_evaluate_interference_k4.py  
          calculate_isdo_k_sweep.py  
          __init__.py  
        isdo_circuit_test/  
          test_isdo_circuits_v2.py  
          test_isdo_circuits_v1.py  
          __init__.py  
        circuits/  
          circuit_a_controlled_state.py  
          circuit_b_prime_transition.py  
          __init__.py  
    archive/  
      __init__.py  
      swap_test/  
        swap_test_classifier.py  
        evaluate_swap_test_batch.py  
        __init__.py
```

```
statevector_similarity/  
    compute_class_states.py  
    evaluate_statevector_similarity.py  
    __init__.py  
rfc/  
    reflection_classifier.py  
    __init__.py  
classical/  
    cnn.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  
    run_regime2.py  
    verify_transition_backend.py  
    run_regime3c_consolidation.py  
    run_regime3c_v2.py  
    verify_isdo_bprime_v2_backend.py  
    __init__.py  
IQC/  
    __init__.py  
learning/  
    regime2_update.py  
    __init__.py  
states/  
    class_state.py  
    state_init.py  
    __init__.py  
training/  
    regime2_trainer.py  
    regime3c_trainer_v2.py  
    regime3a_trainer.py  
    metrics.py  
    regime3c_trainer_v1.py  
    __init__.py  
observable/  
    isdo_score.py  
    __init__.py  
memory/  
    memory_bank.py  
    __init__.py  
interference/  
    circuit_backend_hadamard.py  
    base.py  
    circuit_backend_isdo_bprime.py  
    circuit_backend_isdo_bprime_v2.py  
    circuit_backend.py  
    circuit_backend_transition.py  
    math_backend.py
```

```
    __init__.py
inference/
    regime3b_classifier.py
    regime3a_classifier.py
    __init__.py
encoding/
    embedding_to_state.py
    __init__.py
results/
    artifacts/
        regime3c_memory.pkl
    checkpoints/
        pcam_cnn_final.pt
        pcam_cnn_best.pt
    embeddings/
        val_labels.npy
        val_labels_polar.npy
        class_state_1.npy
        val_embeddings.npy
        split_test_idx.npy
        class_state_0.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: scripts_Classical/make_embedding_split.py

```

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

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

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

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

indices = np.arange(len(y))

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

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

```

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

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

File: scripts_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
# -----
# 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
# -----

# Standardized features (LR, SVM)
Xtr_s = X_std[train_idx]
Xte_s = X_std[test_idx]
ytr    = y[train_idx]
yte    = y[test_idx]

# L2-normalized features (kNN)
Xtr_l2 = X_l2[train_idx]
Xte_l2 = X_l2[test_idx]

results = {}

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

pred_lr = logreg.predict(Xte_s)
proba_lr = logreg.predict_proba(Xte_s)[:, 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_s, ytr)

pred_svm = svm.predict(Xte_s)

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_l2, ytr)
print("Knn neighbors:", knn.n_neighbors)
pred_knn = knn.predict(Xte_l2)
proba_knn = knn.predict_proba(Xte_l2)[:, 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: 0.9087 | AUC: 0.9706703413940256
      LinearSVM    | Acc: 0.9120 | AUC: None
              kNN  | Acc: 0.9260 | AUC: 0.9690398293029872
"""
```

File: scripts_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, label_polar = [], [], []

with torch.no_grad():
    for x, y in tqdm(loader):
        z = model(x.to(DEVICE), return_embedding=True)
        embeds.append(z.cpu().numpy())
        labels.append(y.numpy())
        label_polar.append((y.numpy()*2 - 1))

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(label_polar).astype(np.float64))

```

File: scripts_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: scripts_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 = 20

```

```

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, 5, 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: scripts_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: 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/utils/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 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  $U_{\text{chi\_psi}} = U_{\text{chi}} @ U_{\text{psi}}^{\dagger}$ 

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

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

    Args:
        psi: Source statevector
        chi: Target statevector

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

    # Transition unitary:  $U_{\text{chi}} @ U_{\text{psi}}^{\dagger}$ 
    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/utis/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/utis/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
            to CHW
        ])

```

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

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

File: src/data/transforms.py

```
from torchvision import transforms

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

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

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

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

File: src/quantum/isdo/evaluate_isdo.py

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

from src.quantum.isdo.isdo_classifier import ISDOClassifier
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 = ISDOClassifier(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/quantum/isdo/isdo_classifier.py

```

import os
import numpy as np

class ISDOClassifier:
    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"class0_proto{i}.npy"))
for i in range(K)],
            1: [np.load(os.path.join(proto_dir, f"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

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

```

File: src/quantum/isdo/init.py

File: src/quantum/isdo/isdo_K_sweep/evaluate_isdo_k_sweep.py

```

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

from src.quantum.isdo.isdo_classifier import ISDOClassifier
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 = ISDOClassifier(proto_dir, 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/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}.np"))
        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"))

```


/

Phase B: Interference-Based Measurement-Free Classifier

Prototypes per class: 5

Samples: 1500

Accuracy: 0.8840

=====
"""

File: src/quantum/isdo/isdo_K_sweep/calculate_isdo_k_sweep.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}.npy")
            np.save(path, proto)
            print(f"Saved {path}")

```

File: src/quantum/isdo/isdo_K_sweep/init.py

File: src/quantum/isdo/isdo_circuit_test/test_isdo_circuits_v2.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.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.quantum.isdo.circuits.circuit_b_prime_transition import
run_isdo_circuit_b_prime, verify_isdo_b_prime

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 A: Oracle model (conceptual)
    print("-" * 70)
    print("Circuit A: Oracle Model (Conceptual)")
    print("-" * 70)
    print("Purpose: Pedagogical/motivational")
    print("Observable: Attempts  $\text{Re}\langle\chi|\psi\rangle$  but with oracle assumption")
    print("Status: Conceptual only, not for hardware")
    try:
        result_a = run_isdo_circuit_a(psi, chi)
        print(f"Result: {result_a:.6f}")
        print(f"Expected: {expected_isdo:.6f}")
        print(f"Match: {np.allclose(result_a, expected_isdo, atol=1e-6)}")
    except Exception as e:
        print(f"Error: {e}")
    print()

```

```

# Circuit B: Reflection-based
print("-" * 70)
print("Circuit B: Reflection-Based Phase Kickback")
print("-" * 70)
print("Purpose: Physical implementation attempt")
print("Observable:  $1 - 2|\langle \chi | \psi \rangle|^2$  (quadratic, NOT linear!)")
print("Status: Works but gives wrong observable for ISDO")
try:
    result_b = run_isdo_circuit_b(psi, chi)
    print(f"Result: {result_b:.6f}")
    print(f"Expected (RFC): {expected_rfc:.6f}")
    print(f"Expected (ISDO): {expected_isdo:.6f}")
    print(f"Matches RFC: {np.allclose(result_b, expected_rfc, atol=1e-6)}")
    print(f"Matches ISDO: {np.allclose(result_b, expected_isdo, atol=1e-6)}")
except Exception as e:
    print(f"Error: {e}")
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_isdo_circuit_b_prime(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_isdo_b_prime(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 = run_isdo_circuit_b(psi, chi)
        measured_b_prime = run_isdo_circuit_b_prime(psi, chi)

        print(f"\nTest {i}:")
        print(f"  True ISDO ( $\langle \chi | \psi \rangle$ ): {true_isdo:+.4f}")
        print(f"  Circuit B (reflection): {measured_b:+.4f} (should be {rfc:+.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/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: src/quantum/isdo/isdo_circuit_test/init.py

File: src/quantum/isdo/circuits/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: src/quantum/isdo/circuits/circuit_b_prime_transition.py

```

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

def build_isdo_circuit_b_prime(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!

"""

```
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 (in practice, this comes from previous computation)
```

```
# For simulation, we'll use state preparation
```

```
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_isdo_circuit_b_prime(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_isdo_circuit_b_prime(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
```

```
def verify_isdo_b_prime(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_isdo_circuit_b_prime(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/quantum/isdo/circuits/init.py

File: src/archive/init.py

File: src/archive/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: src/archive/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()

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

# -----
# Batch evaluation
# -----
correct = 0

print(f"\n🐍 Evaluating SWAP-test classifier on {N_SAMPLES} samples\n")

for i in tqdm(range(N_SAMPLES)):
    x = to_quantum_state(X[i])

    F0 = swap_test_fidelity(x, class_state_0, shots=SHOTS)
    F1 = swap_test_fidelity(x, class_state_1, shots=SHOTS)

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

```

```
accuracy = correct / N_SAMPLES

print("\n=====")
print("Measurement-based Quantum SWAP Test")
print(f"Samples: {N_SAMPLES}")
print(f"Shots per test: {SHOTS}")
print(f"Accuracy: {accuracy:.4f}")
print("=====\n")

## output
"""
🌱 Global seed set to 42

🧠 Evaluating SWAP-test classifier on 5000 samples

100%|
██████████████████████████████████████████████████████████████████████████████
██████████████████████████████████████████████████████████████████████████████
██████████████████████████████████████████████████████████████████████████████ | 5000/5000 [03:46<00:00, 22.11it/s]

=====
Measurement-based Quantum SWAP Test
Samples: 5000
Shots per test: 1024
Accuracy: 0.8784
=====
"""
```

File: src/archive/swap_test/init.py

File: src/archive/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:

src/archive/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]))

    F0 = abs(psi.inner(sv_phi0)) ** 2
    F1 = abs(psi.inner(sv_phi1)) ** 2

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

accuracy = correct / N

print("\n=====")
print("Measurement-free (Statevector) Quantum Classifier")
print(f"Samples: {N}")
print(f"Accuracy: {accuracy:.4f}")
print("=====\n")

```

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

🧠 Evaluating measurement-free statevector classifier on 1500 samples

100%|███████████████████████████████████████████████████████████
███████████████████████████████████████████████████████████████
███████████████████████████████████████████████████████████████
██████████ | 1500/1500 [00:00<00:00, 29429.03it/s]


=====
Measurement-free (Statevector) Quantum Classifier
Samples: 1500
Accuracy: 0.8827
=====
"""
```

File: src/archive/statevector_similarity/init.py

File: src/archive/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>
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 <Z> 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: src/archive/rfc/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/**init.py**

File: 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: 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: 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: 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: 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: src/experiments/iqc/run_regime2.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.regime2_trainer import Regime2Trainer
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 = Regime2Trainer(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/verify_transition_backend.py

```

import numpy as np

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

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()
transition_backend = TransitionInterferenceBackend()

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 = transition_backend.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/run_regime3c_consolidation.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.regime3a_trainer import Regime3ATrainer
from src.IQC.inference.regime3b_classifier import Regime3BClassifier
from src.IQC.interference.math_backend import MathInterferenceBackend

# -----
# 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 = Regime3ATrainer(
    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 = Regime3BClassifier(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/run_regime3c_v2.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.interference.circuit_backend_hadamard import
HadamardInterferenceBackend

from src.IQC.training.regime3c_trainer_v2 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()
backend_hadamard = HadamardInterferenceBackend()

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: 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_isdo_bprime_v2_backend.py

```

import numpy as np
from scipy.stats import spearmanr

from src.IQC.interference.circuit_backend_transition import
TransitionInterferenceBackend
from src.IQC.interference.circuit_backend_isdo_bprime_v2 import (
    ISDOBPrimeV2InterferenceBackend
)

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)

ref_backend = TransitionInterferenceBackend()
isdo_v2_backend = ISDOBPrimeV2InterferenceBackend()

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_v2_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' v2 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' v2 vs Transition backend
Sign agreement: 50 / 100
Spearman rank correlation: 0.078019801980198
Mean |difference|: 0.8633964573421116
"""

```

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

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

File: src/IQC/learning/regime2_update.py

```

import numpy as np

def regime2_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 = float(np.real(np.vdot( $\chi$ ,  $\psi$ )))

    if  $y * s \geq 0$ :
        return  $\chi$ , False # correct classification

    delta =  $\eta * y * \psi$ 
     $\chi_{\text{new}} = \chi + \text{delta}$ 
     $\chi_{\text{new}} = \chi_{\text{new}} / \text{np.linalg.norm}(\chi_{\text{new}})$ 

    return  $\chi_{\text{new}}$ , True

```

File: src/IQC/learning/init.py

File: src/IQC/states/class_state.py

```

import numpy as np

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 float(np.real(np.vdot(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/state_init.py

File: src/IQC/states/init.py

File: src/IQC/training/regime2_trainer.py

```
import numpy as np
from ..learning.regime2_update import regime2_update
from ..observable.isdo_score import isdo_score

class Regime2Trainer:
    """
    Online Interference Quantum Classifier (Regime 2)

    Fixed circuit.
    Trainable object:  $|\chi\rangle$ 
    """

    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_score(chi_vec, psi)
        margin = y * s
        y_hat = 1 if s >= 0 else -1

        chi_new, updated = regime2_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/regime3c_trainer_v2.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 = 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 = 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: src/IQC/training/regime3a_trainer.py

```

from ..learning.regime2_update import regime2_update

class Regime3ATrainer:
    """
    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 = regime2_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/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/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: src/IQC/training/**init**.py

File: src/IQC/observable/isdo_score.py

```

import numpy as np

def isdo_score(chi: np.ndarray, psi: np.ndarray) -> float:
    """
    Linear interference score:  $\text{Re} \langle \chi | \psi \rangle$ 
    """
    return float(np.real(np.vdot(chi, psi)))

```

File: src/IQC/observable/**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])))
        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/circuit_backend_hadamard.py

```

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

# If you also want the conceptual/oracle version:
class HadamardInterferenceBackend(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/base.py

```

from abc import ABC, abstractmethod

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

```



```

"""

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

```

File: src/IQC/interference/circuit_backend_isdo_bprime.py

```

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

from .base import InterferenceBackend

class ISDOBPrimeInterferenceBackend(InterferenceBackend):
    """
    ISDO-B' interference backend.

    Implements the observable:
         $S_{\text{ISDO}}(\text{psi}; \text{chi}) = \langle \text{psi} | U_{\text{chi}}^\dagger Z^{\{\otimes n\}} U_{\text{chi}} | \text{psi} \rangle$ 

    Properties:
    - No controlled unitaries
    - Fixed measurement ( $Z^{\{\otimes n\}}$ )
    - chi appears only as a basis change ( $U_{\text{chi}}$ )
    - Hardware-friendly compared to Hadamard test
    """

    def __init__(self):
        # Optional cache for  $U_{\text{chi}}$  to avoid recomputation
        self._cache = {}

    @staticmethod
    def _statevector_to_unitary(vec):
        """
        Build a unitary  $U$  such that  $U |0\dots 0\rangle = |\text{vec}\rangle$ .
        Completed via Gram-Schmidt to a full unitary.
        """
        vec = np.asarray(vec, dtype=np.complex128)
        vec = vec / np.linalg.norm(vec)
        dim = len(vec)

        U = np.zeros((dim, dim), dtype=np.complex128)
        U[:, 0] = vec

```

```

    for i in range(1, dim):
        v = np.zeros(dim, dtype=np.complex128)
        v[i] = 1.0

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

        nrm = np.linalg.norm(v)
        if nrm < 1e-12:
            v = np.random.randn(dim) + 1j * np.random.randn(dim)
            for j in range(i):
                v -= np.vdot(U[:, j], v) * U[:, j]
            v /= np.linalg.norm(v)
        else:
            v /= nrm

        U[:, i] = v

    return U

def _get_U_chi(self, chi):
    """
    Cached construction of U_chi.
    """
    key = chi.tobytes()
    if key not in self._cache:
        U = self._statevector_to_unitary(chi)
        self._cache[key] = UnitaryGate(U)
    return self._cache[key]

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

    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(n)

    # Prepare |psi>
    qc.initialize(psi, list(range(n)))

    # Apply U_chi
    U_chi = self._get_U_chi(chi)
    qc.append(U_chi, list(range(n)))

    # Measure  $Z^{\otimes n}$  expectation exactly (statevector)
    sv = Statevector.from_instruction(qc)
    Z_all = Pauli("Z" + "I" * (n - 1))

```

```
exp_val = sv.expectation_value(Z_all).real
return float(exp_val)
```

File: src/IQC/interference/circuit_backend_isdo_bprime_v2.py

```
import numpy as np
from qiskit import QuantumCircuit
from qiskit.quantum_info import Statevector
from qiskit.circuit.library import UnitaryGate

from .base import InterferenceBackend

class ISDOBPrimeV2InterferenceBackend(InterferenceBackend):
    """
    ISDO-B' v2: Contrastive projector observable.

    Implements:
        
$$S = \langle \psi | U_{\chi}^{\dagger} (|0\rangle\langle 0| - P_{\perp}) U_{\chi} | \psi \rangle$$

        
$$= 2 * |\langle \chi | \psi \rangle|^2 - 1 \quad (\text{with } \alpha = 1)$$


    Measurement procedure:
    1) Prepare  $|\psi\rangle$ 
    2) Apply  $U_{\chi}$ 
    3) Measure probability  $p_0$  of  $|0\dots 0\rangle$ 
    4) Return  $S = 2*p_0 - 1$ 

    No ancilla. No controlled unitaries.
    """

    def __init__(self):
        self._cache = {}

    @staticmethod
    def _statevector_to_unitary(vec):
        """
        Build a unitary U such that  $U |0\dots 0\rangle = |\text{vec}\rangle$ .
        Completed via Gram-Schmidt to a full unitary.
        """
        vec = np.asarray(vec, dtype=np.complex128)
        vec = vec / np.linalg.norm(vec)
        dim = len(vec)

        U = np.zeros((dim, dim), dtype=np.complex128)
        U[:, 0] = vec

        for i in range(1, dim):
            v = np.zeros(dim, dtype=np.complex128)
            v[i] = 1.0
```

```

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

        nrm = np.linalg.norm(v)
        if nrm < 1e-12:
            v = np.random.randn(dim) + 1j * np.random.randn(dim)
            for j in range(i):
                v -= np.vdot(U[:, j], v) * U[:, j]
            v /= np.linalg.norm(v)
        else:
            v /= nrm

        U[:, i] = v

    return U

def _get_U_chi(self, chi):
    key = chi.tobytes()
    if key not in self._cache:
        U = self._statevector_to_unitary(chi)
        self._cache[key] = UnitaryGate(U)
    return self._cache[key]

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

    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(n)

    # Prepare |psi>
    qc.initialize(psi, list(range(n)))

    # Rotate basis with U_chi
    U_chi = self._get_U_chi(chi)
    qc.append(U_chi, list(range(n)))

    # Exact statevector
    sv = Statevector.from_instruction(qc)

    # Probability of |0...0>
    p0 = abs(sv.data[0]) ** 2

    # Contrastive projector score (alpha = 1)
    return float(2.0 * p0 - 1.0)

```

File: src/IQC/interference/circuit_backend.py

File: src/IQC/interference/circuit_backend_transition.py

```

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

class TransitionInterferenceBackend(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 = TransitionInterferenceBackend._statevector_to_unitary(psi)
        U_chi = TransitionInterferenceBackend._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  $|\psi\rangle$  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/math_backend.py

```
import numpy as np
from .base import InterferenceBackend

class MathInterferenceBackend(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/init.py

File: src/IQC/inference/regime3b_classifier.py

```
class Regime3BClassifier:
    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/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: 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: 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
  ],
  "train_acc": [
    0.8622550964355469,
    0.9047431945800781,
    0.9226036071777344,
    0.9327926635742188,
    0.9391098022460938,
    0.9431877136230469,
    0.9461746215820312,
    0.9482002258300781,
    0.9505386352539062,
    0.9581375122070312,
    0.9599342346191406
  ],
  "val_loss": [
    0.7608505549724214,
    0.3770719189342344,
    0.33603281057730783,
    0.4026396208500955,
    0.4809370573348133,
    0.289258603748749,
    0.3426725415774854,
    0.36998813936224906,
    0.7853999140152155,
    0.3571328424004605,
    0.31231515117542585
  ],
  "val_acc": [
    0.689483642578125,
    0.8507080078125,
    0.86328125,
    0.843505859375,
    0.832000732421875,
    0.88818359375,
    0.874786376953125,
    0.866363525390625,
    0.789154052734375,
    0.87628173828125,
    0.884002685546875
  ]
}
```

```
]
}
```

File: results/logs/embedding_baseline_results.json

```
{
  "LogisticRegression": {
    "accuracy": 0.9086666666666666,
    "auc": 0.9706703413940256
  },
  "LinearSVM": {
    "accuracy": 0.912,
    "auc": null
  },
  "kNN": {
    "accuracy": 0.926,
    "auc": 0.9690398293029872
  }
}
```