

Project Summary

Directory Structure

```
measurement-free-quantum-classifier/
  configs/
    paths.yaml
  src/
    evaluate_capacity_sweep_quantum_vs_knn.py
    evaluate_all_qmls.py
    evaluate_iqc_vs_classical.py
    __init__.py
    IQL/
      __init__.py
      models/
        fixed_memory_iqc.py
        static_isdo_model.py
        adaptive_memory_model.py
        __init__.py
      learning/
        class_state.py
        update.py
        metrics.py
        prototype.py
        memory_bank.py
        __init__.py
      backends/
        base.py
        prime_b.py
        hadamard.py
        transition.py
        exact.py
        __init__.py
      regimes/
        regime3b_responsible.py
        regime4a_spawn.py
        regime4b_pruning.py
        regime3a_wta.py
        regime2_online.py
        __init__.py
      inference/
        weighted_vote_classifier.py
        __init__.py
      encoding/
        embedding_to_state.py
        __init__.py
      baselines/
        static_isdo_classifier.py
  scripts/
    test_adaptive_memory_trainer_with_frames.py
```

```

        frames_to_video.py
utils/
    common_backup.py
    common.py
    paths.py
    seed.py
    load_data.py
    label_utils.py
    __init__.py
data/
    pcam_loader.py
    transforms.py
    __init__.py
quantum/
    compute_qsvm_kernel.py
    __init__.py
training/
    test_fixed_memory_iqc.py
    validate_backends.py
    test_static_isdo_model.py
    test_adaptive_memory_trainer.py
    protocol_online/
        train_perceptron.py
    protocol_adaptive_pruning_regime4b/
        test_regime4b_pruning.py
    protocol_adaptive_regime4A/
        test_regime4a.py
    protocol_static/
        evaluate_isdo_k_sweep.py
        evaluate_static_isdo.py
classical/
    make_embedding_split.py
    train_embedding_models.py
    extract_embeddings.py
    visualize_embeddings.py
    train_cnn.py
    verify_embbeings.py
    visualize_pcam.py
    protocol_fixed_regime3b_responsible/
        test_regime3b_egime3b_responsible.py
classical/
    cnn.py
    __init__.py

```

File: configs/paths.yaml

```

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

paths:
    dataset: "dataset"
    checkpoints: "results/checkpoints"

```

```

embeddings: "results/embeddings"
figures: "results/figures"
logs: "results/logs"
class_prototypes: "results/embeddings/class_prototypes"
artifacts: "results/artifacts"
frames_adaptive: "results/frames/adaptive"
video_adaptive: "results/video/adaptive.mp4"
frames_fixed: "results/frames/fixed"
video_fixed: "results/video/fixed.mp4"

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

```

File: src/evaluate_capacity_sweep_quantum_vs_knn.py

```

import os
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier

from src.utils.paths import load_paths
from src.utils.label_utils import ensure_polar, ensure_binary
from src.utils.load_data import load_data

# Quantum models
from src.IQL.models.fixed_memory_iqc import FixedMemoryIQC
from src.IQL.models.adaptive_memory_model import AdaptiveMemoryModel
from src.IQL.learning.class_state import ClassState
from src.IQL.learning.memory_bank import MemoryBank
from src.IQL.backends.exact import ExactBackend
from src.IQL.regimes.regime4a_spawn import Regime4ASpawn
from src.IQL.regimes.regime4b_pruning import Regime4BPruning

def eval_fixed_iqc_k_sweep(Xtr, Xte, ytr_pol, yte_pol, K_values):
    accs = []
    for K in K_values:
        model = FixedMemoryIQC(K=K, eta=0.1)
        model.fit(Xtr, ytr_pol)
        y_pred = model.predict(Xte)
        acc = accuracy_score(yte_pol, y_pred)
        accs.append(acc)
        print(f"Fixed IQC | K={K:<2} | Acc={acc:.4f}")
    return accs

def eval_adaptive_initial_k_sweep(Xtr, Xte, ytr_pol, yte_pol, K_values):
    accs, final_sizes = [], []

```

```
for K in K_values:
    backend = ExactBackend()
    class_states = []

    for cls in [-1, +1]:
        idxs = np.where(ytr_pol == cls)[0][:K]
        for idx in idxs:
            chi = Xtr[idx].astype(np.complex128)
            chi /= np.linalg.norm(chi)
            class_states.append(
                ClassState(chi, label=cls, backend=backend)
            )

memory_bank = MemoryBank(class_states)

learner = Regime4ASpawn(
    memory_bank=memory_bank,
    eta=0.1,
    backend=backend,
    delta_cover=0.2,
    spawn_coldown=100,
    min_polarized_per_class=1,
)

pruner = Regime4BPruning(
    memory_bank=memory_bank,
    tau_harm=-0.15,
    min_age=200,
    min_per_class=1,
    prune_interval=200,
)

model = AdaptiveMemoryModel(
    memory_bank=memory_bank,
    learner=learner,
    pruner=pruner,
    tau_responsible=0.1,
    beta=0.98,
)

model.fit(Xtr, ytr_pol)
model.consolidate(Xtr, ytr_pol, epochs=5, eta_scale=0.3)

y_pred = model.predict(Xte)
acc = accuracy_score(yte_pol, y_pred)

accs.append(acc)
final_sizes.append(len(memory_bank.class_states))

print(
    f"Adaptive IQC | init K={K:<2} | "
    f"final mem={final_sizes[-1]:<2} | Acc={acc:.4f}"
)
```

```
    return accs, final_sizes

def eval_knn_k_sweep(Xtr, Xte, ytr_bin, yte_bin, k_values):
    accs = []
    for k in k_values:
        clf = KNeighborsClassifier(n_neighbors=k)
        clf.fit(Xtr, ytr_bin)
        y_pred = clf.predict(Xte)
        acc = accuracy_score(yte_bin, y_pred)
        accs.append(acc)
        print(f"K-NN | k={k} | Acc={acc:.4f}")
    return accs

def main():
    Xtr, Xte, ytr_bin, yte_bin, ytr_pol, yte_pol = load_data("all")

    K = [i for i in range(1,20)]

    print("\n==== FixedMemory IQC sweep ===")
    fixed_acc = eval_fixed_iqc_k_sweep(Xtr, Xte, ytr_pol, yte_pol, K)

    print("\n==== Adaptive IQC sweep ===")
    adapt_acc, adapt_sizes = eval_adaptive_initial_k_sweep(
        Xtr, Xte, ytr_pol, yte_pol, K
    )

    print("\n==== k-NN sweep ===")
    knn_acc = eval_knn_k_sweep(Xtr, Xte, ytr_bin, yte_bin, K)

    # -----
    plt.figure(figsize=(7, 5))
    plt.plot(K, fixed_acc, marker="o", label="FixedMemory IQC")
    plt.plot(K, adapt_acc, marker="s", label="Adaptive IQC")
    plt.plot(K, knn_acc, marker="^", label="k-NN")

    plt.xlabel("Capacity parameter (K or k)")
    plt.ylabel("Test accuracy")
    plt.title("Accuracy vs Capacity: Quantum IQC vs k-NN")
    plt.legend()
    plt.grid(True)

    _, PATHS = load_paths()
    out = os.path.join(PATHS["figures"],
    "capacity_sweep_quantum_vs_knn.png")
    plt.tight_layout()
    plt.savefig(out)
    plt.close()

    print(f"\nPlot saved to: {out}")
```

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

## output

"""

==== FixedMemory IQC sweep ====
🌱 Global seed set to 42
Fixed IQC | K=1 | Acc=0.9000
🌱 Global seed set to 42
Fixed IQC | K=2 | Acc=0.8927
🌱 Global seed set to 42
Fixed IQC | K=3 | Acc=0.8827
🌱 Global seed set to 42
Fixed IQC | K=4 | Acc=0.8947
🌱 Global seed set to 42
Fixed IQC | K=5 | Acc=0.8927
🌱 Global seed set to 42
Fixed IQC | K=6 | Acc=0.8913
🌱 Global seed set to 42
Fixed IQC | K=7 | Acc=0.8900
🌱 Global seed set to 42
Fixed IQC | K=8 | Acc=0.8847
🌱 Global seed set to 42
Fixed IQC | K=9 | Acc=0.8873
🌱 Global seed set to 42
Fixed IQC | K=10 | Acc=0.8893
🌱 Global seed set to 42
Fixed IQC | K=11 | Acc=0.8913
🌱 Global seed set to 42
Fixed IQC | K=12 | Acc=0.8807
🌱 Global seed set to 42
Fixed IQC | K=13 | Acc=0.8867
🌱 Global seed set to 42
Fixed IQC | K=14 | Acc=0.8893
🌱 Global seed set to 42
Fixed IQC | K=15 | Acc=0.8920
🌱 Global seed set to 42
Fixed IQC | K=16 | Acc=0.8860
🌱 Global seed set to 42
Fixed IQC | K=17 | Acc=0.8887
🌱 Global seed set to 42
Fixed IQC | K=18 | Acc=0.8840
🌱 Global seed set to 42
Fixed IQC | K=19 | Acc=0.8853

==== Adaptive IQC sweep ====

⌚ Consolidation phase started (epochs=5, eta_scale=0.3)
✓ Consolidation epoch 1/5
✓ Consolidation epoch 2/5
✓ Consolidation epoch 3/5
```

✓ Consolidation epoch 4/5
✓ Consolidation epoch 5/5
⌚ Consolidation phase completed

Adaptive IQC | init K=1 | final mem=6 | Acc=0.8967

⌚ Consolidation phase started (epochhs=5, eta_scale=0.3)
✓ Consolidation epoch 1/5
✓ Consolidation epoch 2/5
✓ Consolidation epoch 3/5
✓ Consolidation epoch 4/5
✓ Consolidation epoch 5/5
⌚ Consolidation phase completed

Adaptive IQC | init K=2 | final mem=4 | Acc=0.8773

⌚ Consolidation phase started (epochhs=5, eta_scale=0.3)
✓ Consolidation epoch 1/5
✓ Consolidation epoch 2/5
✓ Consolidation epoch 3/5
✓ Consolidation epoch 4/5
✓ Consolidation epoch 5/5
⌚ Consolidation phase completed

Adaptive IQC | init K=3 | final mem=4 | Acc=0.8787

⌚ Consolidation phase started (epochhs=5, eta_scale=0.3)
✓ Consolidation epoch 1/5
✓ Consolidation epoch 2/5
✓ Consolidation epoch 3/5
✓ Consolidation epoch 4/5
✓ Consolidation epoch 5/5
⌚ Consolidation phase completed

Adaptive IQC | init K=4 | final mem=8 | Acc=0.8660

⌚ Consolidation phase started (epochhs=5, eta_scale=0.3)
✓ Consolidation epoch 1/5
✓ Consolidation epoch 2/5
✓ Consolidation epoch 3/5
✓ Consolidation epoch 4/5
✓ Consolidation epoch 5/5
⌚ Consolidation phase completed

Adaptive IQC | init K=5 | final mem=8 | Acc=0.8980

⌚ Consolidation phase started (epochhs=5, eta_scale=0.3)
✓ Consolidation epoch 1/5
✓ Consolidation epoch 2/5
✓ Consolidation epoch 3/5
✓ Consolidation epoch 4/5
✓ Consolidation epoch 5/5
⌚ Consolidation phase completed

Adaptive IQC | init K=6 | final mem=7 | Acc=0.8847

⌚ Consolidation phase started (epochs=5, eta_scale=0.3)
✓ Consolidation epoch 1/5
✓ Consolidation epoch 2/5
✓ Consolidation epoch 3/5
✓ Consolidation epoch 4/5
✓ Consolidation epoch 5/5
⌚ Consolidation phase completed

Adaptive IQC | init K=7 | final mem=10 | Acc=0.8827

⌚ Consolidation phase started (epochs=5, eta_scale=0.3)
✓ Consolidation epoch 1/5
✓ Consolidation epoch 2/5
✓ Consolidation epoch 3/5
✓ Consolidation epoch 4/5
✓ Consolidation epoch 5/5
⌚ Consolidation phase completed

Adaptive IQC | init K=8 | final mem=11 | Acc=0.8620

⌚ Consolidation phase started (epochs=5, eta_scale=0.3)
✓ Consolidation epoch 1/5
✓ Consolidation epoch 2/5
✓ Consolidation epoch 3/5
✓ Consolidation epoch 4/5
✓ Consolidation epoch 5/5
⌚ Consolidation phase completed

Adaptive IQC | init K=9 | final mem=17 | Acc=0.8687

⌚ Consolidation phase started (epochs=5, eta_scale=0.3)
✓ Consolidation epoch 1/5
✓ Consolidation epoch 2/5
✓ Consolidation epoch 3/5
✓ Consolidation epoch 4/5
✓ Consolidation epoch 5/5
⌚ Consolidation phase completed

Adaptive IQC | init K=10 | final mem=21 | Acc=0.8833

⌚ Consolidation phase started (epochs=5, eta_scale=0.3)
✓ Consolidation epoch 1/5
✓ Consolidation epoch 2/5
✓ Consolidation epoch 3/5
✓ Consolidation epoch 4/5
✓ Consolidation epoch 5/5
⌚ Consolidation phase completed

Adaptive IQC | init K=11 | final mem=21 | Acc=0.8780

⌚ Consolidation phase started (epochs=5, eta_scale=0.3)
✓ Consolidation epoch 1/5

- ✓ Consolidation epoch 2/5
- ✓ Consolidation epoch 3/5
- ✓ Consolidation epoch 4/5
- ✓ Consolidation epoch 5/5

⌚ Consolidation phase completed

Adaptive IQC | init K=12 | final mem=21 | Acc=0.8860

- ⌚ Consolidation phase started (epochs=5, eta_scale=0.3)
- ✓ Consolidation epoch 1/5
- ✓ Consolidation epoch 2/5
- ✓ Consolidation epoch 3/5
- ✓ Consolidation epoch 4/5
- ✓ Consolidation epoch 5/5

⌚ Consolidation phase completed

Adaptive IQC | init K=13 | final mem=21 | Acc=0.8853

- ⌚ Consolidation phase started (epochs=5, eta_scale=0.3)
- ✓ Consolidation epoch 1/5
- ✓ Consolidation epoch 2/5
- ✓ Consolidation epoch 3/5
- ✓ Consolidation epoch 4/5
- ✓ Consolidation epoch 5/5

⌚ Consolidation phase completed

Adaptive IQC | init K=14 | final mem=23 | Acc=0.8653

- ⌚ Consolidation phase started (epochs=5, eta_scale=0.3)
- ✓ Consolidation epoch 1/5
- ✓ Consolidation epoch 2/5
- ✓ Consolidation epoch 3/5
- ✓ Consolidation epoch 4/5
- ✓ Consolidation epoch 5/5

⌚ Consolidation phase completed

Adaptive IQC | init K=15 | final mem=24 | Acc=0.8800

- ⌚ Consolidation phase started (epochs=5, eta_scale=0.3)
- ✓ Consolidation epoch 1/5
- ✓ Consolidation epoch 2/5
- ✓ Consolidation epoch 3/5
- ✓ Consolidation epoch 4/5
- ✓ Consolidation epoch 5/5

⌚ Consolidation phase completed

Adaptive IQC | init K=16 | final mem=28 | Acc=0.8727

- ⌚ Consolidation phase started (epochs=5, eta_scale=0.3)
- ✓ Consolidation epoch 1/5
- ✓ Consolidation epoch 2/5
- ✓ Consolidation epoch 3/5
- ✓ Consolidation epoch 4/5
- ✓ Consolidation epoch 5/5

⌚ Consolidation phase completed

Adaptive IQC | init K=17 | final mem=20 | Acc=0.8500

⌚ Consolidation phase started (epochs=5, eta_scale=0.3)

- ✓ Consolidation epoch 1/5
- ✓ Consolidation epoch 2/5
- ✓ Consolidation epoch 3/5
- ✓ Consolidation epoch 4/5
- ✓ Consolidation epoch 5/5

⌚ Consolidation phase completed

Adaptive IQC | init K=18 | final mem=26 | Acc=0.8660

⌚ Consolidation phase started (epochs=5, eta_scale=0.3)

- ✓ Consolidation epoch 1/5
- ✓ Consolidation epoch 2/5
- ✓ Consolidation epoch 3/5
- ✓ Consolidation epoch 4/5
- ✓ Consolidation epoch 5/5

⌚ Consolidation phase completed

Adaptive IQC | init K=19 | final mem=23 | Acc=0.8820

==== k-NN sweep ===

k-NN	k=1	Acc=0.9140
k-NN	k=2	Acc=0.9187
k-NN	k=3	Acc=0.9233
k-NN	k=4	Acc=0.9340
k-NN	k=5	Acc=0.9260
k-NN	k=6	Acc=0.9300
k-NN	k=7	Acc=0.9267
k-NN	k=8	Acc=0.9307
k-NN	k=9	Acc=0.9260
k-NN	k=10	Acc=0.9267
k-NN	k=11	Acc=0.9267
k-NN	k=12	Acc=0.9253
k-NN	k=13	Acc=0.9247
k-NN	k=14	Acc=0.9253
k-NN	k=15	Acc=0.9247
k-NN	k=16	Acc=0.9227
k-NN	k=17	Acc=0.9233
k-NN	k=18	Acc=0.9233
k-NN	k=19	Acc=0.9220

"""

File: src/evaluate_all_qmls.py

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

```
from src.utils.paths import load_paths
from src.utils.label_utils import ensure_polar, ensure_binary
from src.utils.load_data import load_data
# Models
from src.IQL.models.static_isdo_model import StaticISDOModel
from src.IQL.models.fixed_memory_iqc import FixedMemoryIQC
from src.IQL.models.adaptive_memory_model import AdaptiveMemoryModel

# Adaptive components
from src.IQL.learning.class_state import ClassState
from src.IQL.learning.memory_bank import MemoryBank
from src.IQL.backends.exact import ExactBackend
from src.IQL.regimes.regime4a_spawn import Regime4ASpawn
from src.IQL.regimes.regime4b_pruning import Regime4BPruning

def eval_static_isdo(X_train, X_test, y_train_bin, y_test_bin):
    model = StaticISDOModel(K=3)
    model.fit(X_train, y_train_bin)
    y_pred = model.predict(X_test)
    acc = accuracy_score(y_test_bin, y_pred)
    return acc, "static", 6 # 2*K memories

def eval_fixed_iqc(X_train, X_test, y_train_pol, y_test_pol):
    model = FixedMemoryIQC(K=3, eta=0.1)
    model.fit(X_train, y_train_pol)
    y_pred = model.predict(X_test)
    acc = accuracy_score(y_test_pol, y_pred)
    mem = len(model.memory_bank.class_states)
    return acc, "fixed", mem

def eval_adaptive_iqc(X_train, X_test, y_train_pol, y_test_pol):
    backend = ExactBackend()

    # Bootstrap memory (1 per class)
    class_states = []
    for cls in [-1, +1]:
        idx = np.where(y_train_pol == cls)[0][0]
        chi = X_train[idx].astype(np.complex128)
        chi /= np.linalg.norm(chi)
        class_states.append(ClassState(chi, label=cls, backend=backend))

    memory_bank = MemoryBank(class_states)

    learner = Regime4ASpawn(
        memory_bank=memory_bank,
        eta=0.1,
        backend=backend,
        delta_cover=0.2,
        spawn_coldown=100,
        min_polarized_per_class=1,
```

```
)\n\npruner = Regime4BPruning(\n    memory_bank=memory_bank,\n    tau_harm=-0.15,\n    min_age=200,\n    min_per_class=1,\n    prune_interval=200,\n)\n\nmodel = AdaptiveMemoryModel(\n    memory_bank=memory_bank,\n    learner=learner,\n    pruner=pruner,\n    tau_responsible=0.1,\n    beta=0.98,\n)\n\n# Adaptive phase\nmodel.fit(X_train, y_train_pol)\n\n# Consolidation phase\nmodel.consolidate(X_train, y_train_pol, epochs=5, eta_scale=0.3)\n\ny_pred = model.predict(X_test)\nacc = accuracy_score(y_test_pol, y_pred)\nmem = len(memory_bank.class_states)\n\nreturn acc, "adaptive", mem\n\n\ndef main():\n    print("\nUnified Model Evaluation\n")\n\n    Xtr, Xte, ytr_bin, yte_bin, ytr_pol, yte_pol = load_data("all")\n\n    results = []\n\n    acc, typ, mem = eval_static_isdo(Xtr, Xte, ytr_bin, yte_bin)\n    results.append(("Static ISDO", acc, typ, mem))\n\n    acc, typ, mem = eval_fixed_iqc(Xtr, Xte, ytr_pol, yte_pol)\n    results.append(("FixedMemory IQC (K=2)", acc, typ, mem))\n\n    acc, typ, mem = eval_adaptive_iqc(Xtr, Xte, ytr_pol, yte_pol)\n    results.append(("Adaptive IQC (with consolidation)", acc, typ, mem))\n\n    print("\n==== Final Comparison ===")\n    for name, acc, typ, mem in results:\n        print(f"{name:35s} | Acc: {acc:.4f} | Type: {typ:8s} | Memory:\n{mem}")\n\n\nif __name__ == "__main__":\n    /\n
```

```
main()
```

File: src/evaluate_iqc_vs_classical.py

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

# Classical models
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier

# Utilities
from src.utils.paths import load_paths
from src.utils.label_utils import ensure_polar, ensure_binary
from src.utils.load_data import load_data

# Adaptive IQC
from src.IQL.models.adaptive_memory_model import AdaptiveMemoryModel
from src.IQL.learning.class_state import ClassState
from src.IQL.learning.memory_bank import MemoryBank
from src.IQL.backends.exact import ExactBackend
from src.IQL.regimes.regime4a_spawn import Regime4ASpawn
from src.IQL.regimes.regime4b_pruning import Regime4BPruning

def eval_classical_models(X_train, X_test, y_train_bin, y_test_bin):
    results = []

    models = {
        "Logistic Regression": LogisticRegression(
            max_iter=500,
            solver="lbfgs"
        ),
        "Linear SVM": SVC(
            kernel="linear"
        ),
        "RBF SVM": SVC(
            kernel="rbf",
            gamma="scale"
        ),
        "k-NN": KNeighborsClassifier(),
    }

    for name, model in models.items():
        model.fit(X_train, y_train_bin)
        y_pred = model.predict(X_test)
        acc = accuracy_score(y_test_bin, y_pred)
        results.append((name, acc))
```

```
    return results

def eval_adaptive_iqc(X_train, X_test, y_train_pol, y_test_pol):
    backend = ExactBackend()

    # Bootstrap 1 memory per class
    class_states = []
    for cls in [-1, +1]:
        idx = np.where(y_train_pol == cls)[0][0]
        chi = X_train[idx].astype(np.complex128)
        chi /= np.linalg.norm(chi)
        class_states.append(
            ClassState(chi, label=cls, backend=backend)
        )

    memory_bank = MemoryBank(class_states)

    learner = Regime4ASpawn(
        memory_bank=memory_bank,
        eta=0.1,
        backend=backend,
        delta_cover=0.2,
        spawn_coldown=100,
        min_polarized_per_class=1,
    )

    pruner = Regime4BPruning(
        memory_bank=memory_bank,
        tau_harm=-0.15,
        min_age=200,
        min_per_class=1,
        prune_interval=200,
    )

    model = AdaptiveMemoryModel(
        memory_bank=memory_bank,
        learner=learner,
        pruner=pruner,
        tau_responsible=0.1,
        beta=0.98,
    )

    # Adaptive training
    model.fit(X_train, y_train_pol)

    # Consolidation
    model.consolidate(
        X_train,
        y_train_pol,
        epochs=5,
        eta_scale=0.3,
    )
```

```
y_pred = model.predict(X_test)
acc = accuracy_score(y_test_pol, y_pred)
mem = len(memory_bank.class_states)

return acc, mem

def main():
    print("\n\n Adaptive IQC vs Classical Models\n")

    Xtr, Xte, ytr_bin, yte_bin, ytr_pol, yte_pol = load_data("all")

    # Classical models
    classical_results = eval_classical_models(
        Xtr, Xte, ytr_bin, yte_bin
    )

    # Adaptive IQC
    iqc_acc, iqc_mem = eval_adaptive_iqc(
        Xtr, Xte, ytr_pol, yte_pol
    )

    print("\n==== Classical Models ===")
    for name, acc in classical_results:
        print(f"{name:25s} | Acc: {acc:.4f}")

    print("\n==== Adaptive Quantum Model ===")
    print(
        f"Adaptive IQC (consolidated) | "
        f"Acc: {iqc_acc:.4f} | Memory: {iqc_mem}"
    )

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

File: src/**init**.py

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

File: src/IQL/models/fixed_memory_iqc.py

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

import os
import numpy as np

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

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

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

    def __init__(self, K: int, eta: float = 0.1, backend=None, alpha: float
= 0, beta: float = 1):
        self.K = K
        self.eta = eta
        self.backend = backend or ExactBackend()
        self.alpha = alpha
        self.beta = beta

        self.memory_bank = None
        self.trainer = None
        self.classifier = None

    def _ensure_prototypes(self, X, y):
        """
        Generate prototypes if they do not already exist.
        """

        _, PATHS = load_paths()
        proto_base = PATHS["class_prototypes"]
        proto_dir = os.path.join(proto_base, f"K{self.K}")

        os.makedirs(proto_dir, exist_ok=True)
        y_binary = ensure_binary(y)
        generate_prototypes(
            X=X,
            y=y_binary,
```

```

        K=self.K,
        output_dir=proto_dir
    )
    return load_prototypes(K=self.K, output_dir=proto_dir)

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

    #
    # Step 2: initialize memory bank
    #
    class_states = [
        ClassState(v["vector"], backend=self.backend, label=v["label"])
        for v in proto
    ]
    self.memory_bank = MemoryBank(class_states)

    #
    # Step 3: Regime-3A training
    #
    self.trainer = WinnerTakeAll(
        memory_bank=self.memory_bank,
        eta=self.eta,
        backend=self.backend,
        alpha = self.alpha,
        beta = self.beta
    )
    self.trainer.fit(X, y)

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

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

```

File: src/IQL/models/static_isdo_model.py

```

# src/IQL/models/static_isdo_model.py

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

```

```
import os

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

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

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

    def _ensure_prototypes(self, X, y):
        """
        Generate prototypes if they do not already exist.
        """
        _, PATHS = load_paths()
        proto_base = PATHS["class_prototypes"]
        proto_dir = os.path.join(proto_base, f"K{self.K}")
        os.makedirs(proto_dir, exist_ok=True)
        generate_prototypes(
            X=X,
            y=y,
            K=self.K,
            output_dir=proto_dir,
            seed = 42
        )

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

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

```
# src/IQL/training/adaptive_memory_trainer.py
from src.IQL.regimes.regime4a_spawn import Regime4ASpawn
from src.IQL.regimes.regime4b_pruning import Regime4BPruning
from src.IQL.regimes.regime3a_wta import WinnerTakeAll
from src.utils.paths import load_paths
import os

class AdaptiveMemoryModel:
    """
    System-level adaptive controller for IQC.

    Responsibilities:
    - Enforce memory lifecycle invariants
    - Orchestrate learning (Regime-4A)
    - Handle aging, harm tracking, and pruning (Regime-4B)

    This class contains NO learning logic.
    """

    def __init__(
        self,
        memory_bank,
        learner : Regime4ASpawn,           # Regime4ASpawn
        pruner : Regime4BPruning,          # Regime4BPruning
        tau_responsible: float = 0.1,
        beta: float = 0.98,
        frames_dir: str = None,
        fps: int = 50,
    ):
        self.memory_bank = memory_bank
        self.learner = learner
        self.pruner = pruner
        self.tau_responsible = tau_responsible
        self.beta = beta

        self.step_count = 0
        self.history = {
            "action": [],           # spawned | updated | noop
            "memory_size": [],
            "num_pruned": [],
        }
        self.frames_dir = frames_dir
        self.FRAME_EVERY = fps

    def consolidate(
        self,
        X,
        y,
        epochs: int = 5,
        eta_scale: float = 0.3,
    ):
        """
        Post-adaptive consolidation phase.
        
```

```
- Freezes memory structure (no spawn, no prune)
- Refines existing memories using WTA updates
- Improves margins and accuracy

Args:
    X: input states
    y: true labels (polar)
    epochs: number of consolidation passes
    eta_scale: scale factor for learning rate
"""

print(
    f"\n❸ Consolidation phase started "
    f"(epochs={epochs}, eta_scale={eta_scale})"
)

# Winner-Take-All learner (Regime-3A semantics)
consolidator = WinnerTakeAll(
    memory_bank=self.memory_bank,
    eta=self.learner.eta * eta_scale,
    backend=self.learner.backend,
    alpha=0.0,      # update only on error
    beta=1.0,       # full update
)

# IMPORTANT: freeze adaptive structure
original_spawn = self.learner.step
original_prune = self.pruner.step

try:
    # Disable spawn & prune
    self.learner.step = lambda psi, y: "noop"
    self.pruner.step = lambda: []

    for ep in range(epochs):
        consolidator.fit(X, y)
        print(f" ✓ Consolidation epoch {ep+1}/{epochs}")

finally:
    # Restore adaptive behavior
    self.learner.step = original_spawn
    self.pruner.step = original_prune

print("❹ Consolidation phase completed\n")
return self

def step(self, psi, y, frames=False):
    """
    Execute ONE adaptive training step.

    Ordering is STRICT and MUST NOT be changed.
    """

```

```
# -----
# STEP 1: learning + possible memory spawn
# -----
action = self.learner.step(psi, y)

# -----
# STEP 2: age update (MANDATORY)
# -----
self.memory_bank.increment_age()

# -----
# STEP 3: harm EMA update (MANDATORY)
# -----
self.memory_bank.update_harm_ema(
    psi,
    y_true=y,
    tau_responsible=self.tau_responsible,
    beta=self.beta,
)

# -----
# STEP 4: pruning (PERIODIC)
# -----
pruned = self.pruner.step()
num_pruned = len(pruned) if pruned else 0

# -----
# STEP 5: bookkeeping
# -----
self.step_count += 1

if frames:
    if self.step_count % self.FRAME_EVERY == 0:
        frame_path = f'{self.frames_dir}/frame_{self.step_count:05d}.png'
        self.memory_bank.visualize(
            qubit=0,
            title="Adaptive IQC - Memory States (Final Snapshot)",
            save_path=frame_path,
            show=False,
        )

    self.history["action"].append(action)
    self.history["memory_size"].append(
        len(self.memory_bank.class_states)
    )
    self.history["num_pruned"].append(num_pruned)

return action, pruned

def fit(self, X, y):
    """
    Online adaptive training loop.
    /
```

```

"""
    if self.frames_dir is None:
        print("No frames directory specified. Skipping frame saving.")
        frames = False
    else:
        frames = True
        os.makedirs(self.frames_dir, exist_ok=True)
    for psi, label in zip(X, y):
        self.step(psi, label, frames)
    return self

def predict(self, X):
"""
    Inference using winner-take-all interference.
"""
    preds = []
    for psi in X:
        _, score = self.memory_bank.winner(psi)
        preds.append(1 if score >= 0 else -1)
    return preds

def summary(self):
    return {
        "steps": self.step_count,
        "final_memory_size": len(self.memory_bank.class_states),
        "num_spawns": self.history["action"].count("spawned"),
        "num_pruned": sum(self.history["num_pruned"]),
    }

```

File: src/IQL/models/init.py

File: src/IQL/learning/class_state.py

```

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

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

class ClassState:
"""

```

```

Represents the quantum class memory  $|\chi\rangle$ .
Invariant:  $\|\chi\| = 1$  always.
"""

def __init__(self, vector: np.ndarray, label: int, backend: InterferenceBackend = None):
    self.vector = normalize(vector.astype(np.complex128))
    self.backend = backend
    self.label = label
    self.age = 0
    self.harm_ema = 0.0

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

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

```

File: src/IQL/learning/update.py

```

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

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

    If  $y * \text{Re}\langle \chi | \psi \rangle \geq 0$ :
        no update
    else:
         $\chi \leftarrow \text{normalize}(\chi + \eta * y * \psi)$ 
    """
    s = backend.score(chi, psi)

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

```

```
    delta = eta * y * psi
    chi_new = chi + delta
    chi_new = chi_new / np.linalg.norm(chi_new)

    return chi_new, True
```

File: src/IQL/learning/metrics.py

```
import numpy as np

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

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

File: src/IQL/learning/prototype.py

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

from src.utils.seed import set_seed

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

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

    # Create K clusters
    kmeans = KMeans(n_clusters=K, random_state=seed).fit(X)
```

```
"""
set_seed(seed)
os.makedirs(output_dir, exist_ok=True)

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

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

    kmeans = KMeans(
        n_clusters=K,
        random_state=seed,
        n_init=10
    )
    kmeans.fit(X_cls)

    centers = kmeans.cluster_centers_

    for i in range(K):
        proto = to_quantum_state(centers[i])

        path = os.path.join(
            output_dir, f"class{cls}_proto{i}.npz"
        )

        # ----- SAVE VECTOR + LABEL -----
        np.savez(
            path,
            vector=proto,
            label=cls
        )

def load_prototypes(K, output_dir):
    """
    Load prototypes generated by generate_prototypes.

    Returns:
        List[dict]: each dict has keys { "vector", "label" }
    """
    prototypes = []

    for cls in [0, 1]:
        for i in range(K):
            # New format (.npz)
            npz_path = os.path.join(
                output_dir, f"class{cls}_proto{i}.npz"
            )

            if os.path.exists(npz_path):
                data = np.load(npz_path)
```

```
        prototypes.append({
            "vector": data["vector"],
            "label": int(data["label"]),
        })
    else:
        # ----- BACKWARD COMPATIBILITY (.npy) -----
        npy_path = os.path.join(
            output_dir, f"class{cls}_proto{i}.npy"
        )
        vec = np.load(npy_path)
        prototypes.append({
            "vector": vec,
            "label": None,
        })

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

    # Reproducibility
    set_seed(42)

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

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

    # Load embeddings (TRAIN ONLY)
    X = np.load(os.path.join(EMBED_DIR, "val_embeddings.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"]

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

```
)  
print(f"Saved prototypes to {CLASS_DIR}")
```

File: src/IQL/learning/memory_bank.py

```
from src.IQL.learning.class_state import ClassState  
from qiskit.visualization.bloch import Bloch  
import numpy as np  
import matplotlib.pyplot as plt  
  
class MemoryBank:  
    def __init__(self, class_states):  
        self.class_states = class_states  
  
    def scores(self, psi):  
        return [  
            cs.score(psi)  
            for cs in self.class_states  
        ]  
  
    def increment_age(self):  
        """  
        Increment age of all memories by 1.  
        Call once per training step.  
        """  
        for cs in self.class_states:  
            cs.age += 1  
  
    def update_harm_ema(self, psi, y_true, tau_responsible, beta):  
        """  
        Update harm EMA for responsible memories.  
  
        Args:  
            psi: input state  
            tau_responsible: responsibility threshold  
            beta: EMA decay factor  
        """  
        scores = self.scores(psi)  
  
        for cs, s in zip(self.class_states, scores):  
            if abs(s) > tau_responsible and cs.label is not None:  
                harm = -y_true * s  
                cs.harm_ema = beta * cs.harm_ema + (1 - beta) * harm  
  
    def winner(self, psi):  
        scores = self.scores(psi)  
        idx = int(max(range(len(scores)), key=lambda i: abs(scores[i])))  
        #idx = int(max(range(len(scores)), key=lambda i: scores[i])) ##  
        causes lower score ??  
        return idx, scores[idx]
```

```
def add_memory(self, chi_vector, backend, label: int):
    """
    Add a new memory to the bank.

    Args:
        chi_vector: quantum state vector
        backend: interference backend
        label: class label (mandatory)
    """
    self.class_states.append(ClassState(chi_vector, backend=backend,
                                         label=label))

def remove(self, idx):
    """Remove memory at index idx."""
    if 0 <= idx < len(self.class_states):
        del self.class_states[idx]

def prune(self, prune_states):
    """
    Remove given ClassState objects from the memory bank.
    """
    self.class_states = [
        cs for cs in self.class_states
        if cs not in prune_states
    ]

def visualize(
    self,
    qubit: int = 0,
    title: str | None = None,
    save_path: str | None = None,
    show: bool = True,
):
    """
    Visualize the MEMORY-BANK-LEVEL geometry on a single Bloch sphere.

    - Points : individual memory states (projected)
    - Arrows : class centroids
    - Colors : red = class -1 / 0, blue = class +1 / 1
    - STATIC snapshot (no learning, no dynamics)
    """
    if not self.class_states:
        raise RuntimeError("MemoryBank is empty")

    bloch = Bloch()
    bloch.vector_color = []

    red_pts, blue_pts = [], []
    # --- Pauli matrices ---
    X = np.array([[0, 1], [1, 0]], dtype=complex)
    Y = np.array([[0, -1j], [1j, 0]], dtype=complex)
```

```
Z = np.array([[1, 0], [0, -1]], dtype=complex)
I = np.eye(2, dtype=complex)

def pauli_on_qubit(P, q, n):
    ops = [I] * n
    ops[q] = P
    out = ops[0]
    for op in ops[1:]:
        out = np.kron(out, op)
    return out

# --- Project each memory ---
for cs in self.class_states:
    chi = cs.vector
    n = int(np.log2(len(chi)))
    if 2**n != len(chi):
        raise ValueError("State dimension must be 2^n")

    Xq = pauli_on_qubit(X, qubit, n)
    Yq = pauli_on_qubit(Y, qubit, n)
    Zq = pauli_on_qubit(Z, qubit, n)

    v = np.array([
        float(np.real(np.vdot(chi, Xq @ chi))),
        float(np.real(np.vdot(chi, Yq @ chi))),
        float(np.real(np.vdot(chi, Zq @ chi))),
    ])
    bloch.add_vectors(v)
    bloch.vector_color.append(
        "red" if cs.label in [-1, 0] else "blue"
    )

    if cs.label in [-1, 0]:
        red_pts.append(v)
    else:
        blue_pts.append(v)

# --- Add centroid arrows ---
def add_centroid(vectors, color):
    """
    Add a class centroid as an ARROW on the Bloch sphere.

    - vectors : list of Bloch vectors (Nx3)
    - color   : color for the centroid arrow
    """
    if len(vectors) == 0:
        return

    mu = np.mean(vectors, axis=0)
    norm = np.linalg.norm(mu)

    if norm < 1e-9:
```

```
    return

    # Keep centroid inside Bloch ball
    mu = mu / max(1.0, norm)

    # --- Temporarily force arrow rendering ---
    previous_style = bloch.vector_style
    bloch.vector_style = "arrow"

    bloch.add_vectors(mu)
    bloch.vector_color.append(color)

    # --- Restore previous style (points) ---
    bloch.vector_style = previous_style

    add_centroid(red_pts, "darkred")
    add_centroid(blue_pts, "darkblue")

    # --- Title / save / show ---
    if title:
        bloch.title = title

    if save_path:
        bloch.save(save_path)

    if show:
        plt.show()
    else:
        plt.close(bloch.fig)
```

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

File: src/IQL/backends/base.py

```
from abc import ABC, abstractmethod

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

    @abstractmethod
    def score(self, chi, psi) -> float:
        """
        Return Re<chi | psi> as a real scalar.
    
```

```
"""
pass
```

File: src/IQL/backends/prime_b.py

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

from .base import InterferenceBackend


class PrimeBBackend(InterferenceBackend):
    """
    PrimeB (ISDO-B') Backend
    -----
    Observable-engineered, decision-sufficient implementation of ISDO.

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

    Properties:
    - No ancilla qubit
    - No controlled unitaries
    - X appears only as a basis rotation
    - Fixed, hardware-native observable
    - Preserves sign + ordering (not exact inner product)

    Intended role:
    - Fast inference
    - NISQ-friendly deployment backend
    """
    @staticmethod
    def _statevector_to_unitary(state: np.ndarray) -> np.ndarray:
        """
        Construct a unitary U such that:
             $U |0\dots0\rangle = |\text{state}\rangle$ 

        Uses Gram-Schmidt completion.
        """
        state = np.asarray(state, dtype=np.complex128)
        state = state / np.linalg.norm(state)

        dim = len(state)
        U = np.zeros((dim, dim), dtype=np.complex128)
        U[:, 0] = state
```

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

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

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

    U[:, i] = v

return U

def score(self, chi: np.ndarray, psi: np.ndarray) -> float:
    """
    Compute PrimeB interference score.

    Args:
        chi : np.ndarray
            Class memory state  $|x\rangle$ 
        psi : np.ndarray
            Input state  $|\psi\rangle$ 

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

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

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

    # Build circuit
    qc = QuantumCircuit(n)

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

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

```

# Evaluate <Z^{<n}>
sv = Statevector.from_instruction(qc)
observable = Pauli("Z"+"I" * (n-1))

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

```

File: src/IQL/backends/hadamard.py

```

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

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

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

    Computes Re<chi | psi> in oracle model.
    """

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

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

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

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

        # Hadamard on ancilla
        qc.h(anc)

        # Controlled state preparation (ORACLE ASSUMPTION)
        # When anc=0: prepare |psi>
        state_prep_psi = StatePreparation(psi)

```

```

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

# Flip ancilla
qc.x(anc)

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

# Flip back
qc.x(anc)

# Final Hadamard
qc.h(anc)

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

return float(z_exp)

```

File: src/IQL/backends/transition.py

```

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

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

    This is the physically realizable ISDO implementation.
    Computes Re<chi | psi> 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...0>"""
        vec = np.asarray(vec, dtype=np.complex128)
        vec = vec / np.linalg.norm(vec)
        dim = len(vec)

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

```

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

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

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

return U

@staticmethod
def _build_transition_unitary(psi, chi):
    """Build U_chi_psi = U_chi @ U_psi^dagger"""
    U_psi = TransitionBackend._statevector_to_unitary(psi)
    U_chi = TransitionBackend._statevector_to_unitary(chi)

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

    return UnitaryGate(U_chi_psi)

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

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

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

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

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

    # Hadamard on ancilla
    qc.h(anc)

    # Controlled transition unitary
    U_chi_psi = self._build_transition_unitary(psi, chi)
```

```

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

# Final Hadamard
qc.h(anc)

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

return float(z_exp)

```

File: src/IQL/backends/exact.py

```

import numpy as np
from .base import InterferenceBackend

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

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

```

File: src/IQL/backends/init.py

File: src/IQL/regimes/regime3b_responsible.py

```

import numpy as np
from src.IQL.learning.update import update
from src.IQL.backends.exact import ExactBackend

class Regime3BResponsible:
    """
    Regime-3B: Responsible-Set Corrective Learning

    - Same as Regime-3A, but:
      instead of updating only the winner,
      update all RESPONSIBLE memories.

    - Direction still comes from y_true

```

```
- Uses existing update() primitive
- Guard-A: update energy normalized by |responsible set|
"""

def __init__(
    self,
    memory_bank,
    eta,
    backend=ExactBackend(),
    alpha_correct: float = 0.0,
    alpha_wrong: float = 1.0,
    tau: float = 0.1,    # responsibility threshold
):
    self.memory_bank = memory_bank
    self.eta = eta
    self.backend = backend

    self.alpha_correct = alpha_correct
    self.alpha_wrong = alpha_wrong
    self.tau = tau

    self.num_updates = 0

# -----
# Core step
# -----
def step(self, psi, y_true):
    # Compute all scores
    scores = self.memory_bank.scores(psi)

    # Winner (for prediction only)
    idx_star = int(np.argmax(np.abs(scores)))
    score_star = scores[idx_star]

    # Correctness
    misclassified = (y_true * score_star) < 0
    alpha = self.alpha_wrong if misclassified else self.alpha_correct

    # Prediction
    y_hat = 1 if score_star >= 0 else -1

    # -----
    # Responsible set
    # -----
    responsible = [
        cs for cs, s in zip(self.memory_bank.class_states, scores)
        if abs(s) > self.tau
    ]

    # Fallback: always update winner at least
    if not responsible:
        responsible = [self.memory_bank.class_states[idx_star]]

    # Guard-A normalization
    /
```

```
scale = self.eta * alpha / len(responsible)

# -----
# Update ALL responsible memories
# -----
for cs in responsible:
    chi_new, updated = update(
        cs.vector,
        psi,
        y_true,      # ← direction = truth (unchanged)
        scale,
        self.backend,
    )
    if updated:
        cs.vector = chi_new
        self.num_updates += 1

return y_hat

# -----
# Training loop
# -----
def fit(self, X, y):
    correct = 0
    for psi, label in zip(X, y):
        y_hat = self.step(psi, label)
        if y_hat == label:
            correct += 1
    return correct / len(X)

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

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

# -----
# Summary
# -----
def summary(self):
    return {
        "memory_size": len(self.memory_bank.class_states),
        "num_updates": self.num_updates,
        "tau": self.tau,
        "alpha_correct": self.alpha_correct,
        "alpha_wrong": self.alpha_wrong,
    }
```

File: src/IQL/regimes/regime4a_spawn.py

```
import numpy as np
from collections import defaultdict

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

class Regime4ASpawn:
    """
    Regime-4A: Interference-Coverage Adaptive Memory

    Properties:
    - New implementation (does NOT inherit from Regime-3C)
    - Uses EXACT Regime-3A semantics:
        * Winner-Take-All selection
        * Same misclassification rule
        * Same Regime-2 update
    - Adds memory ONLY when:
        * Winner interference is weak (poor coverage)
        * Winner misclassifies the sample
        * Spawn cooldown allows it
    - Early phase: class-polarized spawning
    - Later phase: class-agnostic spawning
    """

    def __init__(
        self,
        memory_bank,
        eta: float = 0.1,
        backend=None,
        delta_cover: float = 0.2,
        spawn_cooldown: int = 100,
        min_polarized_per_class: int = 1,
    ):
        """
        Args:
            memory_bank (MemoryBank): existing memory bank
            eta (float): learning rate (Regime-2 update)
            backend (InterferenceBackend): scoring backend
            delta_cover (float): minimum |interference| required to avoid
            spawning
            spawn_cooldown (int): minimum steps between spawns
            min_polarized_per_class (int): bootstrap polarized memories per
            class
        """
        self.memory_bank = memory_bank
        self.eta = eta
        self.backend = backend or ExactBackend()

        # Regime-4A parameters
```

```
    self.delta_cover = delta_cover
    self.spawn_cooldown = spawn_cooldown
    self.min_polarized_per_class = min_polarized_per_class

    # Internal state
    self.steps_since_spawn = spawn_cooldown
    self.polarized_count = defaultdict(int)

    # Logging / diagnostics
    self.num_spawns = 0
    self.num_updates = 0
    self.history = {
        "action": [],           # "spawned" | "updated" | "noop"
        "winner_score": [],
        "memory_size": [],
    }

# -----
# Core step
# -----
def step(self, psi: np.ndarray, y: int):
    """
    Process a single training example.

    Args:
        psi (np.ndarray): input quantum state |psi>
        y (int): label in {-1, +1}

    Returns:
        action (str): "spawned", "updated", or "noop"
    """

# -----
# 1. Compute interference scores
# -----
scores = self.memory_bank.scores(psi)

if len(scores) == 0:
    raise RuntimeError("MemoryBank is empty – cannot run Regime-4A.")

# -----
# 2. Winner-Take-All (EXACT Regime-3A semantics)
# -----
winner_idx = max(range(len(scores)), key=lambda i: abs(scores[i]))
s_star = scores[winner_idx]

# -----
# 3. Coverage + misclassification checks
# -----
poor_coverage = abs(s_star) < self.delta_cover
misclassified = (y * s_star) < 0
spawn_allowed = self.steps_since_spawn >= self.spawn_cooldown
```

```
# -----
# 4. Regime-4A: Spawn new memory if needed
# -----
if poor_coverage and misclassified and spawn_allowed:
    residual = psi.astype(np.complex128, copy=True)

    # Orthogonalize against existing memory
    for cs in self.memory_bank.class_states:
        proj = np.vdot(cs.vector, psi)
        residual -= proj * cs.vector

    norm = np.linalg.norm(residual)

    if norm > 1e-8:
        residual /= norm

    # -----
    # Polarized → agnostic transition
    # -----
    # Polarized → agnostic transition
    if self.polarized_count[y] < self.min_polarized_per_class:
        chi_new = y * residual
        self.polarized_count[y] += 1
        label = y # ✓ SET LABEL for polarized memories
    else:
        chi_new = residual
        label = y # Agnostic memories now also carry a label
for consistency

    self.memory_bank.add_memory(chi_new, self.backend,
label=label) # ✓ PASS LABEL
    self.steps_since_spawn = 0
    self.num_spawns += 1

    self.history["action"].append("spawned")
    self.history["winner_score"].append(float(s_star))
    self.history["memory_size"].append(
        len(self.memory_bank.class_states)
    )

return "spawned"

# -----
# 5. Otherwise: standard Regime-3A update
# -----
cs = self.memory_bank.class_states[winner_idx]

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

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

```

        self.steps_since_spawn += 1

        self.history["action"].append("updated" if updated else "noop")
        self.history["winner_score"].append(float(s_star))
        self.history["memory_size"].append(
            len(self.memory_bank.class_states)
        )

    return "updated" if updated else "noop"

# -----
# Training loop
# -----
def fit(self, X, y):
    """
    Online training over dataset.

    Args:
        X (Iterable[np.ndarray]): input states
        y (Iterable[int]): labels in {-1, +1}
    """
    for psi, label in zip(X, y):
        self.step(psi, label)
    return self

# -----
# Convenience helpers
# -----
def memory_size(self) -> int:
    return len(self.memory_bank.class_states)

def summary(self) -> dict:
    return {
        "memory_size": self.memory_size(),
        "num_spawns": self.num_spawns,
        "num_updates": self.num_updates,
    }

```

File: src/IQL/regimes/regime4b_pruning.py

```

class Regime4BPruning:
    """
    Regime-4B: Responsible EMA-Based Memory Pruning

    Removes memories that:
    - are old enough
    - are responsible for interference
    - consistently interfere destructively with their own class
    """

```

```
def __init__(  
    self,  
    memory_bank,  
    tau_harm=-0.2,  
    min_age=200,  
    min_per_class=1,  
    prune_interval=200,  
):  
    self.memory_bank = memory_bank  
    self.tau_harm = tau_harm  
    self.min_age = min_age  
    self.min_per_class = min_per_class  
    self.prune_interval = prune_interval  
  
    self.step_count = 0  
    self.num_pruned = 0  
  
# -----  
# Called once per training step  
# -----  
def step(self):  
    self.step_count += 1  
  
    if self.step_count % self.prune_interval != 0:  
        return []  
  
    return self.prune()  
  
# -----  
# Core pruning logic  
# -----  
def prune(self):  
    to_prune = []  
  
    # Count memories per class  
    class_counts = {}  
    for cs in self.memory_bank.class_states:  
        class_counts.setdefault(cs.label, 0)  
        class_counts[cs.label] += 1  
  
    # Identify prune candidates  
    for cs in self.memory_bank.class_states:  
        if cs.age < self.min_age:  
            continue  
  
        if cs.harm_ema < self.tau_harm:  
            # enforce class floor  
            if class_counts.get(cs.label, 0) > self.min_per_class:  
                to_prune.append(cs)  
                class_counts[cs.label] -= 1  
  
    if to_prune:  
        self.memory_bank.prune(to_prune)
```

```
        self.num_pruned += len(to_prune)

    return to_prune

# -----
# Diagnostics
# -----
def summary(self):
    return {
        "num_pruned": self.num_pruned,
        "current_memory_size": len(self.memory_bank.class_states),
        "steps": self.step_count,
    }
```

File: src/IQL/regimes/regime3a_wta.py

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

class WinnerTakeAll:
    """
    Regime 3-A: Winner-Takes-All IQC ( $\alpha$ -scaled formulation)

    Special case:
        alpha_correct = 0
        alpha_wrong   = 1
    reproduces the original Regime-3A exactly.
    """

    def __init__(
        self,
        memory_bank,
        eta,
        backend=ExactBackend(),
        alpha: float = 0.0,
        beta: float = 1.0,
    ):
        self.memory_bank = memory_bank
        self.eta = eta
        self.backend = backend

        # Scaling factors
        self.alpha_correct = alpha
        self.alpha_wrong = beta

        self.num_updates = 0

        self.history = {
```

```
"winner_idx": [],
"scores": [],
"updates": [],
"alpha": [],
}

def step(self, psi, y):
# -----
# Winner selection (unchanged)
# -----
idx, score = self.memory_bank.winner(psi)
cs = self.memory_bank.class_states[idx]

# -----
# Correctness check
# -----
misclassified = (y * score) < 0

# α-scaling (THIS is the only real change)
alpha = self.alpha_wrong if misclassified else self.alpha_correct

# -----
# Scaled update (winner only)
# -----
chi_new, updated = update(
    cs.vector,
    psi,
    y,
    self.eta * alpha,
    self.backend,
)
if updated:
    cs.vector = chi_new
    self.num_updates += 1

# Prediction (unchanged)
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)
self.history["alpha"].append(alpha)

return y_hat, idx, updated

def fit(self, X, y):
correct = 0
for x, label in zip(X, y):
    y_hat, _, _ = self.step(x, label)
    if y_hat == label:
        correct += 1
print(f"Accuracy: {correct / len(y)}")
```

```
        correct += 1
    return correct / len(X)

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

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

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

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

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

    obj = cls(
        memory_bank=payload["memory_bank"],
        eta=payload["eta"],
        backend=payload["backend"],
        alpha_correct=payload.get("alpha_correct", 0.0),
        alpha_wrong=payload.get("alpha_wrong", 1.0),
    )

    obj.num_updates = payload["num_updates"]
    obj.history = payload["history"]

    return obj
```

File: src/IQL/regimes/regime2_online.py

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

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

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

    def __init__(self, class_state, eta: float):
        self.class_state = class_state
        self.eta = eta
        # logs
        self.num_updates = 0
        self.history = {
            "scores": [],
            "margins": [],
            "updates": [],
        }

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

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

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

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

        return y_hat, s, updated

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

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

accuracy = correct / len(X)
return accuracy

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

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

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

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

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

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

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

    return obj
```

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

File: src/IQL/inference/weighted_vote_classifier.py

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

        if weights is None:
            self.weights = [1.0 / self.M] * self.M
        else:
            s = sum(weights)
            self.weights = [w / s for w in weights]

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

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

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

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

File: src/IQL/inference/init.py

File: src/IQL/encoding/embedding_to_state.py

```

import numpy as np

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

```

File: src/IQL/encoding/init.py

File: src/IQL/baselines/static_isdo_classifier.py

```

import os
import numpy as np
from tqdm import tqdm
from src.IQL.backends.exact import ExactBackend
from src.IQL.learning.prototype import load_prototypes

class StaticISDOClassifier:
    def __init__(self, proto_dir, K):
        self.proto_dir = proto_dir
        self.K = K
        self.exact = ExactBackend()
        protos = load_prototypes(
            K=K,
            output_dir=os.path.join(proto_dir, f"K{K}")
        )
        # Binary split (ignore labels even if present)
        self.prototypes = {0: [], 1: []}
        for p in protos:
            # class index is encoded in filename order,
            # OR we can rely on p["label"] if present
            cls = p["label"] if p["label"] is not None else None
            self.prototypes[cls].append(p["vector"])

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

```

```
    chi /= np.linalg.norm(chi)
    return 1 if self.exact.score(chi, psi) < 0 else 0

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

File: src/scripts/test_adaptive_memory_trainer_with_frames.py

```
# src/training/protocol_adaptive/test_adaptive_memory_trainer.py

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

from src.utils.paths import load_paths
from src.utils.load_data import load_data

from src.IQL.learning.class_state import ClassState
from src.IQL.learning.memory_bank import MemoryBank

from src.IQL.backends.exact import ExactBackend
from src.IQL.regimes.regime4a_spawn import Regime4ASpawn
from src.IQL.regimes.regime4b_pruning import Regime4BPruning
from src.IQL.models.adaptive_memory_model import AdaptiveMemoryModel

import matplotlib.pyplot as plt
from collections import Counter

def main():
    print("\n🚀 Testing AdaptiveMemoryTrainer (v0)\n")
    _, paths = load_paths()
    # -----
    # Load data
    # -----
    X_train, X_test, y_train, y_test = load_data("polar")

    print(f"Train samples: {len(X_train)}")
    print(f"Test samples : {len(X_test)}")

    # -----
    # Bootstrap initial memory (1 per class)
    # -----
    backend = ExactBackend()
    class_states = []

    for cls in [-1, +1]:
        idx = np.where(y_train == cls)[0][0]
        chi = X_train[idx].astype(np.complex128)
```

```
chi /= np.linalg.norm(chi)
class_states.append(
    ClassState(chi, label=cls, backend=backend)
)

memory_bank = MemoryBank(class_states)
print("Initial memory size:", len(memory_bank.class_states))

# -----
# Regime-4A (spawn)
# -----
learner = Regime4ASpawn(
    memory_bank=memory_bank,
    eta=0.1,
    backend=backend,
    delta_cover=0.2,
    spawn_coldown=100,
    min_polarized_per_class=1,
)

# -----
# Regime-4B (pruning)
# -----
pruner = Regime4BPruning(
    memory_bank=memory_bank,
    tau_harm=-0.15,
    min_age=200,
    min_per_class=1,
    prune_interval=200,
)

# -----
# Adaptive trainer
# -----
trainer = AdaptiveMemoryModel(
    memory_bank=memory_bank,
    learner=learner,
    pruner=pruner,
    tau_responsible=0.1,
    beta=0.98,
    frames_dir=paths["frames_adaptive"],
)

# -----
# Train
# -----
trainer.fit(X_train, y_train)

# -----
# Consolidation phase
# -----
trainer.consolidate(
    X_train,
    y_train,
```

```
    epochs=5,
    eta_scale=0.3,
)

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

print("\n==== Adaptive Trainer Summary ===")
print(trainer.summary())

print("\n==== Evaluation ===")
print(f"Test Accuracy      : {acc:.4f}")
print(f"Final Memory Size : {len(memory_bank.class_states)}")

print("\n✓ AdaptiveMemoryTrainer test completed.\n")

# -----
# Save adaptive diagnostics
# -----
RESULTS_DIR = paths["frames_adaptive"][:-9]
#save_adaptive_plots(trainer, memory_bank, RESULTS_DIR)
memory_bank.visualize(qubit=0,title="Adaptive IQC - Memory States
(Final Snapshot)", save_path=os.path.join(RESULTS_DIR,
"memory_states.png"), show=True,)

def save_adaptive_plots(trainer, memory_bank, out_dir):
    os.makedirs(out_dir, exist_ok=True)

    # -----
    # 1. Memory size over time
    # -----
    plt.figure(figsize=(6, 4))
    plt.plot(trainer.history["memory_size"])
    plt.xlabel("Training step")
    plt.ylabel("Memory size")
    plt.title("Adaptive Memory Size Over Time")
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(os.path.join(out_dir, "memory_size_over_time.png"))
    plt.close()

    # -----
    # 2. Action distribution
    # -----
    action_counts = Counter(trainer.history["action"])

    plt.figure(figsize=(5, 4))
    plt.bar(action_counts.keys(), action_counts.values())
    plt.xlabel("Action type")
    plt.ylabel("Count")
    plt.title("Adaptive Actions Distribution")
```

```

plt.tight_layout()
plt.savefig(os.path.join(out_dir, "action_distribution.png"))
plt.close()

# -----
# 3. Harm EMA distribution
# -----
harm = [cs.harm_ema for cs in memory_bank.class_states]

plt.figure(figsize=(6, 4))
plt.hist(harm, bins=20)
plt.axvline(x=0.0, linestyle="--")
plt.xlabel("Harm EMA")
plt.ylabel("Count")
plt.title("Harm EMA Distribution (Final)")
plt.tight_layout()
plt.savefig(os.path.join(out_dir, "harm_ema_distribution.png"))
plt.close()

# -----
# 4. Memory age distribution
# -----
ages = [cs.age for cs in memory_bank.class_states]

plt.figure(figsize=(6, 4))
plt.hist(ages, bins=15)
plt.xlabel("Memory age")
plt.ylabel("Count")
plt.title("Memory Age Distribution (Final)")
plt.tight_layout()
plt.savefig(os.path.join(out_dir, "memory_age_distribution.png"))
plt.close()

print(f"\nAll Adaptive plots saved to: {out_dir}")

if __name__ == "__main__":
    main()

```

File: src/scripts/frames_to_video.py

```

# scripts/frames_to_video.py
import cv2
import glob
import os
from src.utils.paths import load_paths

def frames_to_video(
    frames_dir: str,
    output_path: str,

```

```
fps: int = 15,
pattern: str = "frame_*.png",
):
"""
Convert saved Bloch-sphere frames into a video.

Parameters
-----
frames_dir : str
    Directory containing frame images.
output_path : str
    Path to output video (e.g. .mp4).
fps : int
    Frames per second.
pattern : str
    Glob pattern for frame files.
"""

frame_paths = sorted(glob.glob(os.path.join(frames_dir, pattern)))

if not frame_paths:
    raise RuntimeError("No frames found to convert into video.")

# Read first frame to get dimensions
first = cv2.imread(frame_paths[0])
if first is None:
    raise RuntimeError("Failed to read first frame.")

height, width, _ = first.shape

fourcc = cv2.VideoWriter_fourcc(*"mp4v")
video = cv2.VideoWriter(
    output_path,
    cv2.CAP_FFMPEG,
    fourcc,
    fps,
    (width, height),
)

for path in frame_paths:
    img = cv2.imread(path)
    if img is None:
        raise RuntimeError(f"Failed to read frame: {path}")
    video.write(img)

video.release()
print(f"✓ Video written to: {output_path}")

if __name__ == "__main__":
    _, path = load_paths()
    frames_dir = path["frames_adaptive"]
    output_path = path["video_adaptive"]
    os.makedirs(os.path.dirname(output_path), exist_ok=True)
    frames_to_video()
```

```
    frames_dir=frames_dir,
    output_path=output_path,
    fps=15,
)
```

File: src/utils/common_backup.py

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

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

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

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

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

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

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

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

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

```
v[i] = 1.0

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

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

return U

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

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

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

    Args:
        psi: Source statevector
        chi: Target statevector

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

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

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

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

    return UnitaryGate(U_chi_psi)
```

```

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

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

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

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

```

File: src/utils/common.py

```

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

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

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

def statevector_to_unitary(psi):
    """
    Convert a statevector to a unitary operator using Householder
    efficiency.

    Construct a Householder reflection U such that U |e1> = |\psi>
    where e1 = [1, 0, ..., 0]^T.

    This is O(D^2) to build the matrix, compared to O(D^3) for Gram-
    Schmidt.
    """
    psi = np.asarray(psi, dtype=np.complex128)
    norm = np.linalg.norm(psi)

```

```

if norm > 1e-15:
    psi = psi / norm

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

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

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

if w_norm < 1e-12:
    # psi is already phase * e1, so just return identity * phase
    return np.eye(dim, dtype=np.complex128) * phase

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

```

```

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

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

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

    Args:
        psi: Source statevector
        chi: Target statevector

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

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

```

```

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

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

return UnitaryGate(U_chi_psi)

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

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

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

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

```

File: src/utils/paths.py

```

import yaml
import os

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

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

```

File: src/utils/seed.py

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

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

    # NumPy
    np.random.seed(seed)

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

    # cuDNN (important)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

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

File: src/utils/load_data.py

```
import os
import numpy as np
from src.utils.paths import load_paths
from src.utils.seed import set_seed
from src.utils.label_utils import ensure_polar
from src.utils.label_utils import ensure_binary

def load_data(y="all", limit=None):
    set_seed(42)

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

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

    y_bin = ensure_binary(y_bin)
    y_pol = ensure_polar(y_pol)

    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"))
```

```

if limit :
    np.random.shuffle(train_idx)
    np.random.shuffle(test_idx)
    train_idx = train_idx[:limit]
    test_idx = test_idx[:limit]

X_train, X_test = X[train_idx], X[test_idx]
y_train_bin, y_test_bin = y_bin[train_idx], y_bin[test_idx]
y_train_pol, y_test_pol = y_pol[train_idx], y_pol[test_idx]

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

if y == "binary":
    return X_train, X_test, y_train_bin, y_test_bin
elif y == "polar":
    return X_train, X_test, y_train_pol, y_test_pol
elif y == "all":
    return X_train, X_test, y_train_bin, y_test_bin, y_train_pol,
y_test_pol
else:
    raise ValueError("Invalid value for y")

```

File: src/utils/label_utils.py

```

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

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

import numpy as np

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

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

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

def polar_to_binary(labels):
    """

```

```
Convert polar labels {-1, +1} to binary {0, 1}.

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

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

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

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

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

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

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

File: src/data/pcam_loader.py

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

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

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

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

File: src/data/transforms.py

```
from torchvision import transforms

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

```
])
```

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

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

File: src/quantum/compute_qsvm_kernel.py

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

print("QSVM kernel computation complete.")
```

File: src/quantum/init.py

File: src/training/test_fixed_memory_iqc.py

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

from src.utils.load_data import load_data
from src.IQL.models.fixed_memory_iqc import FixedMemoryIQC


def main():
    # -----
    # Load data
    # -----
    X_train, X_test, y_train, y_test = load_data("polar")

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

    # -----
    # Train Fixed-Memory IQC
    # -----
    K = 1
    model = FixedMemoryIQC(K=K, eta=0.1)#, alpha=0.3, beta=1.5)
    model.fit(X_train, y_train)

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

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

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

File: src/training/validate_backends.py

```
import numpy as np

from src.IQL.backends.exact import ExactBackend
```

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

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

def run_backend_tests(n_qubits=3, n_tests=20):
    backends = {
        "Exact": ExactBackend(),
        "Hadamard": HadamardBackend(),
        "Transition": TransitionBackend(),
        "PrimeB": PrimeBBackend(),
    }

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

    # Fix X
    chi = random_state(n_qubits, seed=42)

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

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

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

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

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

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

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

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

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

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

"""
Test 194
Exact      : -0.224492
Hadamard   : -0.224492
Transition: -0.224492
PrimeB     : +0.095676

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

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

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

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

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

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

```
== Backend Agreement Analysis ==

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

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

File: src/training/test_static_isdo_model.py

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

from src.utils.load_data import load_data
from src.IQL.models.static_isdo_model import StaticISDOModel

def main():
    # -----
    # Load data
    # -----
    X_train, X_test, y_train, y_test = load_data("polar")

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

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

    y_pred = model.predict(X_test)

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

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

File: src/training/test_adaptive_memory_trainer.py

```
# src/training/protocol_adaptive/test_adaptive_memory_trainer.py

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

from src.utils.paths import load_paths
from src.utils.load_data import load_data

from src.IQL.learning.class_state import ClassState
from src.IQL.learning.memory_bank import MemoryBank

from src.IQL.backends.exact import ExactBackend
from src.IQL.regimes.regime4a_spawn import Regime4ASpawn
from src.IQL.regimes.regime4b_pruning import Regime4BPruning
from src.IQL.models.adaptive_memory_model import AdaptiveMemoryModel

import matplotlib.pyplot as plt
from collections import Counter

def main():
    print("\n🚀 Testing AdaptiveMemoryTrainer (v0)\n")

    # -----
    # Load data
    # -----
    X_train, X_test, y_train, y_test = load_data("polar")

    print(f"Train samples: {len(X_train)}")
    print(f"Test samples : {len(X_test)}")

    # -----
    # Bootstrap initial memory (1 per class)
    # -----
    backend = ExactBackend()
    class_states = []

    for cls in [-1, +1]:
        idx = np.where(y_train == cls)[0][0]
        chi = X_train[idx].astype(np.complex128)
        chi /= np.linalg.norm(chi)
        class_states.append(
            ClassState(chi, label=cls, backend=backend)
        )

    memory_bank = MemoryBank(class_states)
    print("Initial memory size:", len(memory_bank.class_states))

    # -----
    # Regime-4A (spawn)
    # -----
    learner = Regime4ASpawn(
```

```
    memory_bank=memory_bank,
    eta=0.1,
    backend=backend,
    delta_cover=0.2,
    spawn_coldown=100,
    min_polarized_per_class=1,
)

# -----
# Regime-4B (pruning)
# -----
pruner = Regime4BPruning(
    memory_bank=memory_bank,
    tau_harm=-0.15,
    min_age=200,
    min_per_class=1,
    prune_interval=200,
)

# -----
# Adaptive trainer
# -----
trainer = AdaptiveMemoryModel(
    memory_bank=memory_bank,
    learner=learner,
    pruner=pruner,
    tau_responsible=0.1,
    beta=0.98,
)

# -----
# Train
# -----
trainer.fit(X_train, y_train)

# -----
# Consolidation phase
# -----
trainer.consolidate(
    X_train,
    y_train,
    epochs=5,
    eta_scale=0.3,
)

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

print("\n== Adaptive Trainer Summary ==")
print(trainer.summary())
```

```
print("\n==== Evaluation ===")
print(f"Test Accuracy      : {acc:.4f}")
print(f"Final Memory Size : {len(memory_bank.class_states)}")

print("\n✓ AdaptiveMemoryTrainer test completed.\n")

# -----
# Save adaptive diagnostics
# -----
RESULTS_DIR = "results/figures/adaptive"
save_adaptive_plots(trainer, memory_bank, RESULTS_DIR)
memory_bank.visualize(
    qubit=0,
    title="Adaptive IQC - Memory States (Final Snapshot)",
    save_path=os.path.join(RESULTS_DIR, "memory_states.png"),
    show=True,
)

def save_adaptive_plots(trainer, memory_bank, out_dir):
    os.makedirs(out_dir, exist_ok=True)

    # -----
    # 1. Memory size over time
    # -----
    plt.figure(figsize=(6, 4))
    plt.plot(trainer.history["memory_size"])
    plt.xlabel("Training step")
    plt.ylabel("Memory size")
    plt.title("Adaptive Memory Size Over Time")
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(os.path.join(out_dir, "memory_size_over_time.png"))
    plt.close()

    # -----
    # 2. Action distribution
    # -----
    action_counts = Counter(trainer.history["action"])

    plt.figure(figsize=(5, 4))
    plt.bar(action_counts.keys(), action_counts.values())
    plt.xlabel("Action type")
    plt.ylabel("Count")
    plt.title("Adaptive Actions Distribution")
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(os.path.join(out_dir, "action_distribution.png"))
    plt.close()

    # -----
    # 3. Harm EMA distribution
    # -----
    harm = [cs.harm_ema for cs in memory_bank.class_states]

    plt.figure(figsize=(6, 4))
```

```
plt.hist(harm, bins=20)
plt.axvline(x=0.0, linestyle="--")
plt.xlabel("Harm EMA")
plt.ylabel("Count")
plt.title("Harm EMA Distribution (Final)")
plt.tight_layout()
plt.savefig(os.path.join(out_dir, "harm_ema_distribution.png"))
plt.close()

# -----
# 4. Memory age distribution
# -----
ages = [cs.age for cs in memory_bank.class_states]

plt.figure(figsize=(6, 4))
plt.hist(ages, bins=15)
plt.xlabel("Memory age")
plt.ylabel("Count")
plt.title("Memory Age Distribution (Final)")
plt.tight_layout()
plt.savefig(os.path.join(out_dir, "memory_age_distribution.png"))
plt.close()

print(f"\nAll Adaptive plots saved to: {out_dir}")

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

File: src/training/protocol_online/train_perceptron.py

```
import numpy as np
import os

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

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

def main():
    X_train, X_test, y_train, y_test = load_data("polar")
```

```

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

class_state = ClassState(chi0, backend=ExactBackend(), label=+1)
trainer = OnlinePerceptron(class_state, eta=0.1)

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

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

if __name__ == "__main__":
    main()

### output
"""
🌱 Global seed set to 42
Loaded train embeddings: (3500, 32)
Final accuracy: 0.8562857142857143
Training stats: {'mean_margin': 0.14930659062683652, 'min_margin':
-0.7069261085786833, 'num_updates': 503, 'update_rate': 0.1437142857142857}
"""

```

File:

src/training/protocol_adaptive_pruning_regime4b/test_regime4b_pruning.py

```

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

from src.utils.paths import load_paths
from src.utils.load_data import load_data
from src.utils.label_utils import ensure_polar
from src.IQL.learning.class_state import ClassState
from src.IQL.learning.memory_bank import MemoryBank
from src.IQL.backends.exact import ExactBackend
from src.IQL.regimes.regime3b_responsible import Regime3BResponsible
from src.IQL.regimes.regime4b_pruning import Regime4BPruning

def main():
    print("\n🚀 Testing Regime-4B (EMA-Based Pruning)\n")

    X_train, X_test, y_train, y_test = load_data("polar")

```

```
# -----
# Initialize memory with extra capacity
# -----
backend = ExactBackend()
class_states = []

for cls in [-1, +1]:
    idxs = np.where(y_train == cls)[0][:4] # 4 per class
    for idx in idxs:
        chi = X_train[idx].astype(np.complex128)
        chi /= np.linalg.norm(chi)
        class_states.append(
            ClassState(chi, backend=backend, label=cls)
        )

memory_bank = MemoryBank(class_states)

print("Initial memory size:", len(memory_bank.class_states))

# -----
# Regime-3B (learning)
# -----
learner = Regime3BResponsible(
    memory_bank=memory_bank,
    eta=0.1,
    alpha_correct=0.0,
    alpha_wrong=1.0,
    tau=0.1,
)

# -----
# Regime-4B (pruning)
# -----
pruner = Regime4BPruning(
    memory_bank=memory_bank,
    tau_harm=-0.15,
    min_age=200,
    min_per_class=1,
    prune_interval=200,
)

# -----
# Training loop
# -----
for step, (psi, label) in enumerate(zip(X_train, y_train)):
    learner.step(psi, label)

    # update metadata
    memory_bank.increment_age()
    memory_bank.update_harm_ema(
        psi,
        y_true=label,
        tau_responsible=0.1,
        beta=0.98,
```

```

    )

    pruned = pruner.step()

    if pruned:
        print(
            f"Step {step}: pruned {len(pruned)} memories "
            f"(current size = {len(memory_bank.class_states)})"
        )

# -----
# Evaluation
# -----
y_pred = [learner.predict_one(x) for x in X_test]
test_acc = accuracy_score(y_test, y_pred)

print("\n==== Evaluation ===")
print(f"Test Accuracy : {test_acc:.4f}")
print(f"Final Memory Size : {len(memory_bank.class_states)}")

print("\n==== Pruning Summary ===")
print(pruner.summary())

print("\n🚀 Regime-4B pruning test completed.\n")

if __name__ == "__main__":
    main()

```

File: src/training/protocol_adaptive_regime4A/test_regime4a.py

```

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

from src.utils.paths import load_paths
from src.utils.load_data import load_data
from src.utils.label_utils import ensure_polar
from src.IQL.learning.class_state import ClassState
from src.IQL.learning.memory_bank import MemoryBank
from src.IQL.backends.exact import ExactBackend
from src.IQL.regimes.regime4a_spawn import Regime4ASpawn
from src.IQL.inference.weighted_vote_classifier import
WeightedVoteClassifier


def main():
    print("\n🚀 Testing Regime-4A (Coverage-Based Adaptive Memory)\n")

    #
    /

```

```
# Load data
# -----
X_train, X_test, y_train, y_test = load_data("polar")

print(f"Train samples: {len(X_train)}")
print(f"Test samples : {len(X_test)}")

# -----
# Initialize memory bank (bootstrap like Regime-3A)
# -----
backend = ExactBackend()

# Simple bootstrap: one memory per class
class_states = []

for cls in [-1, +1]:
    idx = np.where(y_train == cls)[0][0]
    chi0 = X_train[idx].copy()
    chi0 /= np.linalg.norm(chi0)
    class_states.append(ClassState(chi0, label=cls, backend=backend))

memory_bank = MemoryBank(class_states)

print("Initial memory size:", len(memory_bank.class_states))

# -----
# Train Regime-4A
# -----
model = Regime4ASpawn(
    memory_bank=memory_bank,
    eta=0.1,
    backend=backend,
    delta_cover=0.2,
    spawn_coldown=100,
    min_polarized_per_class=1,
)

model.fit(X_train, y_train)

print("\n==== Regime-4A Training Summary ===")
print(model.summary())

# -----
# Inference (Regime-3B style)
# -----
classifier = WeightedVoteClassifier(memory_bank)

y_pred = [classifier.predict(x) for x in X_test]
acc = accuracy_score(y_test, y_pred)

print("\n==== Regime-4A Evaluation ===")
print(f"Test Accuracy      : {acc:.4f}")
print(f"Final Memory Size : {len(memory_bank.class_states)})")
```

```

# -----
# Sanity checks
# -----
print("\n==== Sanity Checks ====")

actions = model.history["action"]
num_spawned = actions.count("spawned")
num_updated = actions.count("updated")

print(f"Spawn events : {num_spawned}")
print(f"Update events : {num_updated}")

if num_spawned == 0:
    print("⚠ No memories spawned – try lowering delta_cover")
elif len(memory_bank.class_states) > 50:
    print("⚠ Memory may be growing too fast")
else:
    print("✓ Memory growth appears controlled")

print("\n✓ Regime-4A test completed successfully.\n")

if __name__ == "__main__":
    main()

```

File: src/training/protocol_static/evaluate_isdo_k_sweep.py

```

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

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

BASE_ROOT, PATHS = load_paths()

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

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

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

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

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

```

```

clf = StaticISDOClassifier(PROTO_BASE, K)

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

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

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

```

File: src/training/protocol_static/evaluate_static_isdo.py

```

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

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

BASE_ROOT, PATHS = load_paths()

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

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

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

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

```

```

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

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

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

```

File: src/training/classical/make_embedding_split.py

```

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

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

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

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

indices = np.arange(len(y))

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

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

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

```

File: src/training/classical/train_embedding_models.py

```

import os
import json
import numpy as np

```

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

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

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

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

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

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

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

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

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

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

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

results = {}
```

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

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

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

# =====
# [2] Linear SVM (Max-margin)
# =====
print("Training Linear SVM...")
svm = LinearSVC()
svm.fit(Xtr, ytr)

pred_svm = svm.predict(Xte)

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

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

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

# -----
# Save results
# -----
```

```

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

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

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

Training Logistic Regression...
Training Linear SVM...
Training k-NN...
Knn neighbors: 5

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

```

File: src/training/classical/extract_embeddings.py

```

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

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

set_seed(42)

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

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

```

```

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

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

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

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

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

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

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

```

File: src/training/classical/visualize_embeddings.py

```

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

set_seed(42)

_, PATHS = load_paths()

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

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

plt.figure(figsize=(7, 6))
plt.scatter(X2[y == 0, 0], X2[y == 0, 1], s=8, label="Benign")

```

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

File: src/training/classical/train_cnn.py

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

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

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

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

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

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

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

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

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

return running_loss / total, correct / total

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

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

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

    return running_loss / total, correct / total

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

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

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

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

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

```
"val_acc"]}

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

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

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

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

        if val_acc > best_val_acc:
            best_val_acc = val_acc
            torch.save(model.state_dict(),
os.path.join(PATHS["checkpoints"], "pcam_cnn_best.pt"))
            print("✓ Best validation accuracy reached : Saved checkpoint")
            wait = 0
        else:
            wait += 1

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

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

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

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

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

File: src/training/classical/verify_embbeings.py

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

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

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

    print(f"Verifying: {file_path}")
    X = np.load(file_path)
    print(f"Shape: {X.shape}, Dtype: {X.dtype}")

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

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

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

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

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

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

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

File: src/training/classical/visualize_pcam.py

```

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

set_seed(42)

_, PATHS = load_paths()

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

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

plt.show()

```

File:

src/training/protocol_fixed_regime3b_responsible/test_regime3b_egime3b_responsible.py

```

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

from src.utils.paths import load_paths
from src.utils.load_data import load_data
from src.utils.label_utils import ensure_polar
from src.IQL.learning.class_state import ClassState
from src.IQL.learning.memory_bank import MemoryBank
from src.IQL.backends.exact import ExactBackend
from src.IQL.regimes.regime3b_responsible import Regime3BResponsible


def main():
    print("\n🚀 Testing Regime-3B (Responsible-Set)\n")

    X_train, X_test, y_train, y_test = load_data("polar")

    # Initial polarized memory
    backend = ExactBackend()
    class_states = []

    for cls in [-1, +1]:
        idx = np.where(y_train == cls)[0][0]
        /

```

```

        chi = X_train[idx].astype(np.complex128)
        chi /= np.linalg.norm(chi)
        class_states.append(
            ClassState(chi, backend=backend, label=cls)
        )

    memory_bank = MemoryBank(class_states)

    model = Regime3BResponsible(
        memory_bank=memory_bank,
        eta=0.1,
        alpha_correct=0.0,
        alpha_wrong=1.0,
        tau=0.1,
    )

    train_acc = model.fit(X_train, y_train)

    print("\n==== Training Summary ===")
    print(model.summary())
    print(f"Train Accuracy : {train_acc:.4f}")

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

    print("\n==== Evaluation ===")
    print(f"Test Accuracy : {test_acc:.4f}")
    print(f"Memory Size : {len(memory_bank.class_states)}")

    print("\n✓ Regime-3B (Responsible-Set) test completed.\n")

if __name__ == "__main__":
    main()

```

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