# === QCAA (Optimized) Accuracy vs Training Size (Real Hardware, Two Columns) ===

# - Feature selection via Simulated Annealing (neal) per sample

# - Single SamplerV2 hardware run on ibm\_brisbane; reuse results for all splits

# - Saves logs & outputs into log5/

# - Two columns only

import os, sys, json, time, uuid, logging

from pathlib import Path

from datetime import datetime

# ---------------- Logging ----------------

RUN\_TS = datetime.now().strftime("%Y%m%d\_%H%M%S")

RUN\_ID = f"qcaaopt\_{RUN\_TS}\_{uuid.uuid4().hex[:6]}"

LOG\_DIR = Path("log5"); LOG\_DIR.mkdir(parents=True, exist\_ok=True)

logger = logging.getLogger(RUN\_ID)

logger.setLevel(logging.INFO)

fmt = logging.Formatter("[%(asctime)s] %(levelname)s - %(message)s", "%Y-%m-%d %H:%M:%S")

fh = logging.FileHandler(LOG\_DIR / "run\_opt.log", encoding="utf-8")

fh.setFormatter(fmt); fh.setLevel(logging.INFO); logger.addHandler(fh)

sh = logging.StreamHandler(sys.stdout)

sh.setFormatter(fmt); sh.setLevel(logging.INFO); logger.addHandler(sh)

JSONL\_PATH = LOG\_DIR / "run\_opt.jsonl"

def log\_json(event: str, \*\*kwargs):

rec = {"ts": datetime.now().isoformat(), "run\_id": RUN\_ID, "event": event, \*\*kwargs}

with open(JSONL\_PATH, "a", encoding="utf-8") as f:

f.write(json.dumps(rec, ensure\_ascii=False) + "\n")

def log\_kv(message: str, \*\*kwargs):

logger.info(f"{message} | " + " ".join(f"{k}={v}" for k,v in kwargs.items()))

log\_json(message, \*\*kwargs)

log\_kv("logging\_initialized", run\_id=RUN\_ID, log\_dir=str(LOG\_DIR.resolve()))

# ---------------- Config ----------------

from qiskit\_ibm\_runtime import QiskitRuntimeService, SamplerV2 as Sampler

from qiskit import QuantumCircuit, transpile

import numpy as np

import pandas as pd

from sklearn.preprocessing import MinMaxScaler

from sklearn.model\_selection import StratifiedShuffleSplit, StratifiedKFold

from sklearn.metrics import roc\_curve

# Optimization pieces

from neal import SimulatedAnnealingSampler

from dimod import BinaryQuadraticModel

# Your credentials (as provided)

IBM\_TOKEN = "wZdn6wKKMk4l3XICKoXmAheXQWSsu5JbbBA9Wrh9vpBc"

IBM\_INSTANCE = "crn:v1:bluemix:public:quantum-computing:us-east:a/6148794ccc8942b0b186858407d6ee44:afcd21b9-6c50-40a2-abe9-de5231e2324f::"

DATA\_PATH = r"C:\Users\Sandip Dutta\Downloads\dataset.csv" # keep same path

OUTPUT\_CSV = LOG\_DIR / "accuracy\_progression\_qcaa\_optimized\_two\_columns.csv"

MASK\_CSV = LOG\_DIR / "feature\_masks\_opt.csv"

TRAIN\_SIZES = [10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100]

SEEDS = [0, 1, 2, 3, 4]

SHOTS = 8192

FEATURES = ['flight\_time\_mean', 'hold\_time\_std', 'gyro\_alpha', 'accel\_y']

GLOBAL\_SA\_SEED = 7

SA\_READS = 50

MIN\_ON = 2 # ensure at least 2 active features

log\_kv("config\_set", data\_path=DATA\_PATH, shots=SHOTS, sa\_reads=SA\_READS,

train\_sizes=",".join(map(str, TRAIN\_SIZES)))

# ---------------- Connect backend ----------------

t0 = time.time()

service = QiskitRuntimeService(channel="ibm\_cloud", token=IBM\_TOKEN, instance=IBM\_INSTANCE)

backend = service.backend("ibm\_brisbane")

print(f"✅ Connected to backend: {backend.name}")

log\_kv("backend\_connected", backend=backend.name, seconds=round(time.time()-t0, 3))

# ---------------- Load dataset ----------------

df = pd.read\_csv(DATA\_PATH)

# label column

label\_col = None

for cand in ["label", "y", "target", "Label", "Target"]:

if cand in df.columns:

label\_col = cand

break

if label\_col is None:

raise ValueError("No label column found. Please include 'label' (or y/target).")

# feature columns

if all(c in df.columns for c in FEATURES):

feature\_cols = FEATURES

else:

feature\_cols = [c for c in df.columns if c != label\_col and np.issubdtype(df[c].dtype, np.number)]

if not feature\_cols:

raise ValueError("No numeric feature columns found besides the label.")

X = df[feature\_cols].to\_numpy(dtype=float)

y = df[label\_col].astype(int).to\_numpy()

scaler = MinMaxScaler()

X\_norm = scaler.fit\_transform(X)

log\_kv("dataset\_loaded",

n\_samples=len(X\_norm), n\_features=X\_norm.shape[1],

n\_genuine=int(y.sum()), n\_impostor=int(len(y)-y.sum()),

feature\_cols=",".join(feature\_cols))

print("📊 Dataset Info:")

print(f"Total samples: {len(X\_norm)}")

print(f"Feature shape: {X\_norm.shape}")

print(f"Genuine: {int(y.sum())}, Impostor: {len(y) - int(y.sum())}")

# ---------------- SA-based feature masks ----------------

def get\_feature\_mask\_sa(d, seed=None, reads=50, min\_on=2):

rng = np.random.default\_rng(seed)

# Random quadratic objective (placeholder for QUBO-based selection)

linear = {i: float(rng.uniform(-1, 1)) for i in range(d)}

quadratic = {(i, j): float(rng.uniform(-1, 1)) for i in range(d) for j in range(i+1, d)}

bqm = BinaryQuadraticModel(linear, quadratic, 0.0, vartype='BINARY')

response = SimulatedAnnealingSampler().sample(bqm, num\_reads=reads, seed=seed)

mask = list(response.first.sample.values())

# Enforce minimum number of active features

if sum(mask) < min\_on:

# turn on top-(min\_on) indices by absolute linear bias

by\_bias = sorted(range(d), key=lambda i: abs(linear[i]), reverse=True)[:min\_on]

for i in by\_bias: mask[i] = 1

return mask

feature\_masks = [get\_feature\_mask\_sa(X\_norm.shape[1], seed=GLOBAL\_SA\_SEED + i, reads=SA\_READS, min\_on=MIN\_ON)

for i in range(len(X\_norm))]

pd.DataFrame(feature\_masks, columns=[f"f{j}" for j in range(len(feature\_cols))]).to\_csv(MASK\_CSV, index=False)

log\_kv("feature\_masks\_generated", path=str(MASK\_CSV.resolve()))

# ---------------- Build masked circuits for ALL samples ----------------

def build\_masked\_circuit(x\_vec, mask):

d = len(x\_vec)

qc = QuantumCircuit(d)

# Encode only selected features

active = [i for i, m in enumerate(mask) if m == 1]

for i in active:

xi = float(np.clip(x\_vec[i], 1e-9, 1 - 1e-9))

theta = 2.0 \* np.arcsin(np.sqrt(xi))

qc.ry(theta, i)

# Entangle only along the active chain (keeps structure sparse)

for a, b in zip(active[:-1], active[1:]):

qc.cz(a, b)

qc.measure\_all()

return qc

circuits = [build\_masked\_circuit(x, w) for x, w in zip(X\_norm, feature\_masks)]

log\_kv("circuits\_built", n\_circuits=len(circuits), qubits=len(feature\_cols))

# Transpile once for backend (targets Brisbane's basis/coupling)

t1 = time.time()

circuits\_t = [transpile(c, backend=backend, optimization\_level=3) for c in circuits]

log\_kv("circuits\_transpiled", seconds=round(time.time()-t1, 3))

# ---------------- Run SamplerV2 once (backend at init) ----------------

sampler = Sampler(mode=backend)

print("🚀 Submitting job to real quantum backend...")

job = sampler.run(circuits\_t, shots=SHOTS)

job\_id = job.job\_id()

print(f"🆔 Job ID: {job\_id}")

print("⏳ Waiting for result...")

log\_kv("job\_submitted", job\_id=job\_id, shots=SHOTS, n\_circuits=len(circuits\_t))

res = job.result()

print("✅ Job completed.")

log\_kv("job\_completed", job\_id=job\_id)

# Save meta

with open(LOG\_DIR / "job\_meta\_opt.txt", "w", encoding="utf-8") as f:

f.write(f"RUN\_ID={RUN\_ID}\njob\_id={job\_id}\nbackend={backend.name}\ninstance={IBM\_INSTANCE}\n")

# ---------------- Parse quasi distributions -> prob of '1' per qubit ----------------

def to\_bits(key, width):

if isinstance(key, int):

return format(key, f"0{width}b")

if isinstance(key, str):

s = key.replace(" ", "")

if set(s) <= {"0","1"} and len(s) == width:

return s

try:

return format(int(s, 2), f"0{width}b")

except Exception:

return s.zfill(width)[:width]

if isinstance(key, tuple):

try:

return "".join("1" if bool(v) else "0" for v in key).zfill(width)[-width:]

except Exception:

return "".join(str(v) for v in key).zfill(width)[-width:]

return format(0, f"0{width}b")

num\_qubits = len(feature\_cols)

prob\_ones = np.zeros((len(circuits), num\_qubits), dtype=float)

if hasattr(res, "quasi\_dists"):

qlist = res.quasi\_dists

for i, qdist in enumerate(qlist):

p1 = np.zeros(num\_qubits, dtype=float)

for k, p in dict(qdist).items():

bits = to\_bits(k, num\_qubits)[::-1]

for q in range(num\_qubits):

if bits[q] == "1":

p1[q] += float(p)

prob\_ones[i] = p1

counts\_pub0 = {to\_bits(k, num\_qubits): int(float(v)\*SHOTS) for k, v in dict(qlist[0]).items()}

with open(LOG\_DIR / "counts\_pub0\_opt.json", "w", encoding="utf-8") as f:

json.dump(counts\_pub0, f, ensure\_ascii=False, indent=2)

log\_kv("counts\_pub0\_saved", path=str((LOG\_DIR / "counts\_pub0\_opt.json").resolve()))

else:

log\_kv("warning\_no\_quasi\_dists", note="Result object has no quasi\_dists; API may have changed.")

# ---------------- Reconstruction error per sample (hardware-derived) ----------------

# Inverse of the encoding mapping on all qubits; unencoded qubits will yield ~0.5 p1 → neutral effect after arcsin(sqrt(.)).

x\_hat = np.arcsin(np.sqrt(np.clip(prob\_ones, 1e-12, 1 - 1e-12)))

recon\_errors = ((X\_norm - x\_hat) \*\* 2).mean(axis=1)

pd.DataFrame({

"index": np.arange(len(recon\_errors)),

"label": y,

"mse": recon\_errors

}).to\_csv(LOG\_DIR / "recon\_errors\_opt.csv", index=False)

log\_kv("recon\_errors\_saved", path=str((LOG\_DIR / "recon\_errors\_opt.csv").resolve()), n=len(recon\_errors))

# ---------------- Helper: threshold fit + accuracy ----------------

def fit\_threshold(train\_scores: np.ndarray, train\_labels: np.ndarray) -> float:

fpr, tpr, thr = roc\_curve(train\_labels, -train\_scores) # higher score = more genuine

J = tpr - fpr

idx = int(np.argmax(J))

return float(thr[idx])

def eval\_accuracy(train\_idx, test\_idx) -> float:

thr = fit\_threshold(recon\_errors[train\_idx], y[train\_idx])

y\_pred = (-recon\_errors[test\_idx] >= thr).astype(int)

return float((y\_pred == y[test\_idx]).mean()) \* 100.0

# ---------------- Accuracy progression over training sizes ----------------

rows = []

for s in TRAIN\_SIZES:

accs = []

if s < 100:

for seed in SEEDS:

sss = StratifiedShuffleSplit(n\_splits=1, train\_size=s/100.0, random\_state=seed)

(tr, te), = sss.split(X\_norm, y)

accs.append(eval\_accuracy(tr, te))

else:

skf = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)

for tr, te in skf.split(X\_norm, y):

accs.append(eval\_accuracy(tr, te))

accs = np.array(accs, dtype=float)

mean = accs.mean()

std = accs.std(ddof=1) if len(accs) > 1 else 0.0

rows.append([s, f"{mean:.1f} ± {std:.1f}"])

log\_kv("size\_done\_opt", size\_percent=s, mean\_acc=round(mean,2), std\_acc=round(std,2), reps=len(accs))

# Save final two-column table

out\_df = pd.DataFrame(rows, columns=["Training Size (%)", "QCAA (Optimized) Accuracy (%)"])

out\_df.to\_csv(OUTPUT\_CSV, index=False)

# Summary file

with open(LOG\_DIR / "summary\_opt.txt", "w", encoding="utf-8") as f:

f.write(f"RUN\_ID: {RUN\_ID}\n")

f.write(f"Backend: {backend.name}\n")

f.write(f"Job ID: {job\_id}\n")

f.write(f"Samples: {len(X\_norm)}\n")

f.write(f"Feature columns: {', '.join(feature\_cols)}\n")

f.write(f"Output CSV: {OUTPUT\_CSV}\n")

f.write(f"Feature masks CSV: {MASK\_CSV}\n")

log\_kv("summary\_saved", output\_csv=str(OUTPUT\_CSV.resolve()))

# Print the two-column table

print("\n=== Accuracy Progression (Two Columns) — QCAA (Optimized) ===")

print(out\_df.to\_string(index=False))

print(f"\nSaved to: {OUTPUT\_CSV.resolve()}")