DEMO

```
# PTQ for AV multiplication
%matplotlib inline
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, Subset
from tqdm import tqdm
import numpy as np
import matplotlib.pyplot as plt
import os

DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
Q15_SCALE = 2**15  # 32768
EPOCHS = 10
BS = 128
LR = 0.002

# Hardware module parameters (aligned with testbench)
A_ROWS = 16
V_COLS = 32
NUM_COLS = 16
TILE_SIZE = 8
WIDTH_INT4 = 4
WIDTH_INT8 = 8
WIDTH_FP16 = 16
NUM_TILES = V_COLS // TILE_SIZE  # 4
TOLERANCE = 0.001

# ── Fixed-point helpers for PTQ ───────────────────────────
def q15_round(x):
    x_clp = torch.clamp(x, -1.0, 1.0 - 1/32768)
    return (x_clp * Q15_SCALE).round() / Q15_SCALE

def real_to_q15(x):
    val = int(x * Q15_SCALE)
    val = max(min(val, 32767), -32768)
    return np.int16(val)

def q15_to_real(x):
    value = np.array(int(x) & 0xFFFF).astype(np.int16)
    return float(value) / Q15_SCALE

def real_to_q1_3(x):
    val = int(x * (2**3))
    val = max(min(val, 7), -8)
    return val & 0xF

def real_to_q1_7(x):
    val = int(x * (2**7))
    val = max(min(val, 127), -128)
    return val & 0xFF

def nibble_to_real_q1_3(n):
    n = n & 0xF
    if n & 0x8:
        n -= 0x10
    return float(n) / (2**3)

# ── Software Precision Assigner ───────────────────────────
def assign_precision(A_matrix):
    sums = np.sum(A_matrix.astype(np.int32), axis=1)  # Shape: (B, L)
    sums_q2_30 = sums * (2**15)  # Shift Q1.15 to Q2.30
    precision_codes = []
    for batch_sums in sums_q2_30:
        codes = []
        for s in batch_sums:
            if s < 30000 * (2**15):  # INT4
                codes.append(0)
            elif s < 40000 * (2**15):  # INT8
                codes.append(1)
            else:  # FP16
```

```
                codes.append(2)
        precision_codes.append(codes)
    return np.array(precision_codes, dtype=np.int32)  # Shape: (B, L)

# —— Software Simulation of attention_av_multiply.sv ————————
def simulate_av_multiply(a_mem, v_mem, precision_sel):
    B = a_mem.shape[0]
    out_mem = np.zeros((B, A_ROWS, V_COLS), dtype=np.int16)
    for b in range(B):
        expected_out = np.zeros((A_ROWS, V_COLS), dtype=np.float64)
        for k in range(NUM_COLS):
            prec = precision_sel[b, k]
            for i in range(A_ROWS):
                raw_a = int(a_mem[b, i, k])
                if prec == 0:  # INT4
                    a_val = nibble_to_real_q1_3(raw_a >> 12)
                elif prec == 1:  # INT8
                    a_val = float(np.array((raw_a & 0xFF00) >> 8).astype(np.int8)) / (2**7)
                else:  # FP16
                    a_val = q15_to_real(raw_a)
                for j in range(V_COLS):
                    raw_v = int(v_mem[b, k, j])
                    if prec == 0:
                        v_val = nibble_to_real_q1_3(raw_v >> 12)
                    elif prec == 1:
                        v_val = float(np.array((raw_v & 0xFF00) >> 8).astype(np.int8)) / (2**7)
                    else:
                        v_val = q15_to_real(raw_v)
                    expected_out[i, j] += a_val * v_val
                    expected_out[i, j] = np.clip(expected_out[i, j], -2.0, 1.999999999)
        expected_q = np.vectorize(real_to_q15)(np.clip(expected_out, -1.0, 0.999969))
        out_mem[b] = expected_q.astype(np.int16)
    return out_mem

# —— Dataset: digits 0 and 1 only ——————————————————————————
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

mnist = datasets.MNIST(root='.', download=True, train=True, transform=transforms.ToTensor())
idx = [i for i, t in enumerate(mnist.targets) if t in (0, 1)]
train_set = Subset(mnist, idx)

mnist_test = datasets.MNIST(root='.', download=True, train=False, transform=transforms.ToTensor())
idx_t = [i for i, t in enumerate(mnist_test.targets) if t in (0, 1)]
test_set = Subset(mnist_test, idx_t)

train_loader = DataLoader(train_set, batch_size=BS, shuffle=True, drop_last=True)
test_loader = DataLoader(test_set, batch_size=BS, shuffle=False)

# —— One-block ViT with PTQ and Mixed-Precision A·V ————————
class OneBlockViT(nn.Module):
    def __init__(self, dim=32, n_patch=7):
        super().__init__()
        patch_sz = n_patch * n_patch
        n_tokens = (28 // n_patch) ** 2  # 16 patches
        self.n_tokens = n_tokens
        self.dim = dim
        self.patch = nn.Linear(patch_sz, dim, bias=False)
        self.pos_emb = nn.Parameter(torch.empty(1, n_tokens, dim))
        nn.init.trunc_normal_(self.pos_emb, std=0.02)
        self.norm1 = nn.LayerNorm(dim)
        self.norm2 = nn.LayerNorm(dim)
        self.att_q = nn.Linear(dim, dim, bias=False)
        self.att_k = nn.Linear(dim, dim, bias=False)
        self.att_v = nn.Linear(dim, dim, bias=False)
        self.proj = nn.Linear(dim, dim, bias=False)
        self.mlp = nn.Sequential(
            nn.Linear(dim, 4*dim), nn.ReLU(), nn.Linear(4*dim, dim))
        self.head = nn.Linear(dim, 1, bias=True)
        self.save_for_verilog = False

    def forward(self, x, quantize=False, use_mixed_precision=False, save_images=None, save_labels=None):
        B = x.size(0)
        x = x.reshape(B, 1, 4, 7, 4, 7).permute(0, 2, 4, 1, 3, 5).reshape(B, -1, 49)
        x = self.patch(x)
```

```
        if quantize:
            x = q15_round(x + q15_round(self.pos_emb))
            x_ln = q15_round(self.norm1(x))
            q, k, v = self.att_q(q15_round(x_ln)), self.att_k(q15_round(x_ln)), self.att_v(q15_round(x_ln))
            wq = self.att_q.weight.detach()
            wk = self.att_k.weight.detach()
            wv = self.att_v.weight.detach()
            wo = self.proj.weight.detach()
            q, k, v = q15_round(q), q15_round(k), q15_round(v)
            att = q15_round((q @ k.transpose(-2, -1)) / (k.size(-1)**0.5))
            att = F.softmax(att, dim=-1)
            if use_mixed_precision:
                x_ln_np = (x_ln * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
                att_np = (att * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
                v_np = (v * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
                wq_np = (wq * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
                wk_np = (wk * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
                wv_np = (wv * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
                wo_np = (wo * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
                precision_sel = assign_precision(att_np)
                if self.save_for_verilog and save_images is not None and save_labels is not None:
                    os.makedirs("verilog_inputs", exist_ok=True)
                    np.save(f"verilog_inputs/wq_np.npy", wq_np)
                    np.save(f"verilog_inputs/wk_np.npy", wk_np)
                    np.save(f"verilog_inputs/wv_np.npy", wv_np)
                    np.save(f"verilog_inputs/wo_np.npy", wo_np)


                    for b in range(min(B, 10)):
                        np.save(f"verilog_inputs/x_ln_np_{b}.npy", x_ln_np[b])

                        np.save(f"verilog_inputs/att_np_{b}.npy", att_np[b])
                        np.save(f"verilog_inputs/v_np_{b}.npy", v_np[b])
                        np.save(f"verilog_inputs/prec_sel_{b}.npy", precision_sel[b])
                        np.save(f"verilog_inputs/image_{b}.npy", save_images[b].cpu().numpy())
                        np.save(f"verilog_inputs/label_{b}.npy", save_labels[b].cpu().numpy())
                att_out = torch.zeros_like(v, device=DEVICE)
                for chunk in range(self.n_tokens // A_ROWS):
                        a_chunk = att_np[:, chunk*A_ROWS:(chunk+1)*A_ROWS, :]
                        v_chunk = v_np
                        prec_chunk = precision_sel[:, chunk*NUM_COLS:(chunk+1)*NUM_COLS]
                        out_chunk = simulate_av_multiply(a_chunk, v_chunk, prec_chunk)
                        att_out[:, chunk*A_ROWS:(chunk+1)*A_ROWS, :] = torch.tensor(out_chunk, dtype=torch.float32, dev
            else:
                att_out = q15_round(att @ v)
            x = q15_round(x + q15_round(self.proj(att_out)))
            y = q15_round(self.norm2(x))
            x = q15_round(x + q15_round(self.mlp(y)))
            x = x.mean(1)
            out = q15_round(self.head(x)).squeeze(1)
        else:
            x = x + self.pos_emb
            x_ln = self.norm1(x)
            q, k, v = self.att_q(x_ln), self.att_k(x_ln), self.att_v(x_ln)
            att = (q @ k.transpose(-2, -1)) / (k.size(-1)**0.5)
            att = F.softmax(att, dim=-1)
            att_out = att @ v
            x = x + self.proj(att_out)
            y = self.norm2(x)
            x = x + self.mlp(y)
            x = x.mean(1)
            out = self.head(x).squeeze(1)
        return out

# ── Training Function ───────────────────────────────
def train_model(model, model_name):
    opt = torch.optim.Adam(model.parameters(), lr=LR, weight_decay=1e-5)
    scheduler = optim.lr_scheduler.StepLR(opt, step_size=5, gamma=0.5)
    crit = nn.BCEWithLogitsLoss()

    print(f"\nTraining {model_name} (Full Precision)")
    for epoch in range(1, EPOCHS+1):
        model.train()
        running_loss = 0
        pbar = tqdm(train_loader, desc=f"Epoch {epoch}", leave=False)
        for img, lbl in pbar:
            img, lbl = img.to(DEVICE), lbl.float().to(DEVICE)
```

```
            opt.zero_grad()
            out = model(img, quantize=False)
            loss = crit(out, lbl)
            loss.backward()
            opt.step()
            running_loss += loss.item()
            pbar.set_postfix(loss=float(loss))
        scheduler.step()
        print(f"[{model_name} Epoch {epoch}] Loss: {running_loss / len(train_loader):.3f}")

# —— Evaluation Function: Generate Inputs for Verilog ———————
def evaluate_generate_inputs(model, model_name):
    model.eval()
    print(f"\nEvaluating {model_name} (Generate Inputs for Verilog)")
    saved_count = 0
    model.save_for_verilog = True
    with torch.no_grad():
        for img, lbl in test_loader:
            img, lbl = img.to(DEVICE), lbl.float().to(DEVICE)
            print("label", lbl)
            model(img, quantize=True, use_mixed_precision=True, save_images=img, save_labels=lbl)
            saved_count += min(lbl.size(0), 10 - saved_count)
            if saved_count >= 10:
                break
    model.save_for_verilog = False
    print(f"Saved inputs for {saved_count} examples in 'verilog_inputs'")

# —— Evaluation Function: Load Hardware Outputs and Predict ——
def evaluate_with_hardware(model, model_name):
    model.eval()
    print(f"\nEvaluating {model_name} (Quantized Mixed Precision with Hardware)")
    correct_hardware = 0
    total = 0
    saved_count = 0
    print("\nComparing Hardware Predictions with Labels:")
    plt.figure(figsize=(15, 3))
    with torch.no_grad():
        for img, lbl in test_loader:
            img, lbl = img.to(DEVICE), lbl.float().to(DEVICE)
            B = img.size(0)
            for b in range(min(B, 10 - saved_count)):
                if not os.path.exists(f"verilog_outputs/out_mem_{b}.npy"):
                    print(f"Hardware output 'out_mem_{b}.npy' not found")
                    continue
                out_mem = np.load(f"verilog_outputs/out_mem_{b}.npy")
                image = np.load(f"verilog_inputs/image_{b}.npy")
                label = np.load(f"verilog_inputs/label_{b}.npy")

                att_out = torch.tensor(out_mem, dtype=torch.float32, device=DEVICE) / Q15_SCALE
                att_out = att_out.unsqueeze(0)


                image_t  = torch.tensor(image, dtype=torch.float32, device=DEVICE)  # (1,28,28)
                image_t  = image_t.unsqueeze(0)

                patches = image_t.reshape(1, 1, 4, 7, 4, 7).permute(0,2,4,1,3,5).reshape(1, -1, 49)
                x = q15_round(model.patch(patches) + model.pos_emb)   # add the skip path


                x = q15_round(x + q15_round(model.proj(att_out)))
                y = q15_round(model.norm2(x))
                x = q15_round(x + q15_round(model.mlp(y)))
                x = x.mean(1)
                out = q15_round(model.head(x)).squeeze(1)
                pred = torch.sign(out).item()
                expected = (2 * label - 1)
                print(f"Example {b}: Predicted = {0 if pred == -1 else 1}, Expected = {label}, Match = {pred == expecte
                correct_hardware += (pred == expected)
                total += 1

                plt.subplot(1, 10, b+1)
                plt.imshow(image.squeeze(), cmap='gray')
                plt.title(f"Pred: {0 if pred == -1 else 1}\nTrue: {label}")
                plt.axis('off')
                saved_count += 1
            if saved_count >= 10:
                break
```

```
    plt.tight_layout()
    plt.show()
    acc_hardware = 100 * correct_hardware / total if total > 0 else 0
    print(f"[{model_name}] Quantized Mixed Precision Hardware Test Accuracy: {acc_hardware:.2f}%")
    return acc_hardware

# ── Main Execution ───────────────────────────────
model = OneBlockViT().to(DEVICE)
train_model(model, "ViT")
# Run this in the first Colab cell to generate inputs
evaluate_generate_inputs(model, "ViT")
# After hardware simulation, run evaluate_with_hardware in a second Colab cell

# ── Export Q1.15 weights ───────────────────────────────
int16_state = {k: torch.round(v * Q15_SCALE).to(torch.int16).cpu()
               for k, v in model.state_dict().items()}
torch.save(int16_state, "vit_q15_mixed_int16.pt")
print("Saved quantized weights to vit_q15_mixed_int16.pt")

```

```
Training ViT (Full Precision)
[ViT Epoch 1] Loss: 0.075
[ViT Epoch 2] Loss: 0.013
[ViT Epoch 3] Loss: 0.008
[ViT Epoch 4] Loss: 0.005
[ViT Epoch 5] Loss: 0.009
[ViT Epoch 6] Loss: 0.004
[ViT Epoch 7] Loss: 0.003
[ViT Epoch 8] Loss: 0.003
[ViT Epoch 9] Loss: 0.002
[ViT Epoch 10] Loss: 0.001

Evaluating ViT (Generate Inputs for Verilog)
label tensor([1., 0., 1., 0., 0., 1., 0., 0., 1., 1., 1., 1., 1., 1., 0., 1., 0., 0.,
        1., 1., 1., 1., 0., 1., 0., 1., 0., 1., 1., 1., 0., 1., 0., 1., 1., 1.,
        1., 0., 0., 1., 1., 1., 0., 0., 1., 1., 1., 1., 0., 1., 1., 1., 0., 1.,
        1., 1., 0., 0., 1., 1., 1., 1., 0., 0., 0., 1., 0., 0., 0., 1., 0., 0.,
        1., 1., 1., 1., 1., 0., 1., 1., 1., 1., 0., 0., 1., 1., 0., 1., 1., 1.,
        0., 1., 1., 1., 0., 1., 1., 0., 0., 0., 1., 0., 1., 1., 1., 0., 1., 0.,
        0., 1., 1., 1., 0., 1., 1., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.,
        0., 0.], device='cuda:0')
Saved inputs for 10 examples in 'verilog_inputs'
Saved quantized weights to vit_q15_mixed_int16.pt
```

```
Start coding or generate with AI.
```

```
# QAT for self attention layer
%matplotlib inline
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, Subset
from tqdm import tqdm
import numpy as np
import matplotlib.pyplot as plt
import os

DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
Q15_SCALE = 2**15  # 32768
EPOCHS = 10
BS = 128
LR = 0.002

# Hardware module parameters (aligned with self_attention_top.sv and testbench)
DATA_WIDTH = 16
L = 16          # Sequence length
E = 32          # Embedding dimension
N = 1           # Number of attention heads
OUT_DATA_WIDTH = 32  # Q2.30 for matmul outputs
TOLERANCE = 0.005
SQRT_E = np.sqrt(32.0)  # ≈ 5.656854
INV_SQRT_E = 1.0 / SQRT_E  # ≈ 0.176776695
INV_SQRT_E_Q15 = int(INV_SQRT_E * (2**15))  # ≈ 5792 in Q1.15

```

```
# —— Fixed-point helpers (from testbench) ————————————————————————————
def real_to_q1_15(x):
    val = int(x * (2**15))
    val = max(min(val, 32767), -32768)
    return np.int16(val)

def q1_15_to_real(x):
    value = np.array(int(x) & 0xFFFF).astype(np.int16)
    return float(value) / (2**15)

def real_to_q1_30(x):
    val = int(x * (2**30))
    val = max(min(val, 2**31 - 1), -(2**31))
    return np.int32(val)

def q1_30_to_real(x):
    value = np.array(int(x) & 0xFFFFFFFF).astype(np.int32)
    return float(value) / (2**30)

def real_to_q1_3(x):
    val = int(x * (2**3))
    val = max(min(val, 7), -8)
    return val & 0xF

def nibble_to_real_q1_3(n):
    n = n & 0xF
    if n & 0x8:
        n -= 0x10
    return float(n) / (2**3)

def real_to_q1_7(x):
    val = int(x * (2**7))
    val = max(min(val, 127), -128)
    return val & 0xFF

def q1_7_to_real(x):
    x = x & 0xFF
    if x & 0x80:
        x -= 0x100
    return float(x) / (2**7)


def hw_multiply_inv_sqrt_e(matmul_result_q30):
    matmul_signed = np.array(matmul_result_q30).astype(np.int32)
    inv_sqrt8_signed = np.int16(INV_SQRT_E_Q15)
    mult_result = np.int64(matmul_signed) * np.int64(inv_sqrt8_signed)
    q30_result = (mult_result >> 15) & 0xFFFFFFFF
    if q30_result & 0x80000000:
        q30_result = q30_result - 0x100000000
    return np.int32(q30_result)

class _RoundSTE(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        # quantise to the nearest Q1.15 integer
        return torch.round(x)

    @staticmethod
    def backward(ctx, g):
        # pretend the derivative of round() is 1 everywhere
        return g


def q15_round(x):
    x_clp = torch.clamp(x, -1.0, 1.0 - 1/32768)
    return _RoundSTE.apply(x_clp * Q15_SCALE) / Q15_SCALE #(x_clp * Q15_SCALE).round() / Q15_SCALE

# —— 4.  Quantised layers ——————————————————————————————————————
class Q15Linear(nn.Module):
    def __init__(self, in_f, out_f, bias=True):
        super().__init__()
        self.w = nn.Parameter(torch.empty(out_f, in_f))
        nn.init.trunc_normal_(self.w, std=0.1)
        self.b = nn.Parameter(torch.zeros(out_f)) if bias else None
    def forward(self, x):
        w_q = q15_round(self.w); x_q = q15_round(x)
        y   = F.linear(x_q, w_q, self.b)
```

```
        return q15_round(y)

class Q15GELU(nn.Module):
    def forward(self, x):            # gelu in fp32 → clamp → q → return
        return q15_round(F.gelu(x))



# —— Custom Self-Attention Layer (based on compute_expected) —————————
class SelfAttention(nn.Module):
    def __init__(self, dim=32, n_tokens=16, n_heads=1):
        super().__init__()
        self.dim = dim
        self.n_tokens = n_tokens
        self.n_heads = n_heads
        self.att_q = Q15Linear(dim, dim, bias=False)
        self.att_k = Q15Linear(dim, dim, bias=False)
        self.att_v = Q15Linear(dim, dim, bias=False)
        self.proj = Q15Linear(dim, dim, bias=False)

    def compute_attention(self, x_np, wq_np, wk_np, wv_np, wo_np):

        # Step 1: QKV generation
        def matmul_expected(a, b):
            expected_out = np.zeros((L * N, E), dtype=float)
            for k in range(E):
                for i in range(L * N):
                    raw_a = int(a[i, k])
                    a_val = q1_15_to_real(raw_a)
                    for j in range(E):
                        raw_b = int(b[k, j])
                        b_val = q1_15_to_real(raw_b)
                        expected_out[i, j] += a_val * b_val
                    expected_out = np.clip(expected_out, -2.0, 1.999999999)
            expected_q30 = np.vectorize(real_to_q1_30)(expected_out)
            expected_q15 = []
            for q30_val in expected_q30.flatten():
                sign_bit = (q30_val >> 31) & 1
                int_bit = (q30_val >> 30) & 1
                if sign_bit == int_bit:
                    q15_val = (sign_bit << 15) | ((q30_val >> 15) & 0x7FFF)
                else:
                    q15_val = 0x8000 if sign_bit else 0x7FFF
                if q15_val & 0x8000:
                    q15_val = q15_val - 0x10000
                expected_q15.append(q15_val)
            return np.array(expected_q15, dtype=np.int16).reshape(L * N, E)

        q = matmul_expected(x_np, wq_np).reshape(L, N, E)
        k = matmul_expected(x_np, wk_np).reshape(L, N, E)
        v = matmul_expected(x_np, wv_np).reshape(L, N, E)

        # Step 2: Attention scores
        q_np = q.reshape(L * N, E)
        k_np = k.reshape(L, N, E)
        matmul_result = np.zeros((L * N, L), dtype=float)
        for e in range(E):
            for i in range(L * N):
                for j in range(L * N):
                    q_val = q1_15_to_real(q_np[i, e])
                    k_val = q1_15_to_real(k_np[j, 0, e])  # N=1
                    matmul_result[i, j] += q_val * k_val
            matmul_result = np.clip(matmul_result, -2.0, 1.999999999)
        matmul_q30 = np.vectorize(real_to_q1_30)(matmul_result)
        a_q30 = np.array([hw_multiply_inv_sqrt_e(matmul_q30[i, j]) for i in range(L * N) for j in range(L)], dtype=np.

        a_q15 = []
        for q30_val in a_q30:
            sign_bit = (q30_val >> 31) & 1
            int_bit = (q30_val >> 30) & 1
            if sign_bit == int_bit:
                q15_val = (sign_bit << 15) | ((q30_val >> 15) & 0x7FFF)
            else:
                q15_val = 0x8000 if sign_bit else 0x7FFF
            if q15_val & 0x8000:
                q15_val = q15_val - 0x10000
            a_q15.append(q15_val)
```

```
    a = np.array(a_q15, dtype=np.int16).reshape(L, N, L)

    # Step 3: Softmax approximation
    a_float = np.vectorize(q1_15_to_real)(a)
    a_softmax = np.zeros((L, N, L), dtype=float)
    for n in range(N):
        for i in range(L):
            relu_row = np.maximum(a_float[i, n, :], 0)
            row_sum = np.sum(relu_row)
            if row_sum != 0:
                a_softmax[i, n, :] = relu_row / row_sum
            else:
                a_softmax[i, n, :] = 0
    a_softmax = np.clip(a_softmax, -1.0, 0.999969482421875)
    a_softmax_q15 = np.vectorize(real_to_q1_15)(a_softmax).reshape(L * N, L)

    # Step 4: Precision assignment
    a_sum = np.sum(a_softmax_q15, axis=0)
    token_precision = []
    for s in a_sum:
        if s < 16384:
            code = 0  # int4
        elif s < 32768:
            code = 1  # int8
        else:
            code = 2  # fp16
        token_precision.append(code)

    # Step 5: A*V multiplication
    av_out = np.zeros((L, E), dtype=float)
    for k_idx in range(L):
        prec = token_precision[k_idx]
        for i in range(L):
            raw_a = int(a_softmax_q15[i, k_idx])
            if prec == 0:
                a_val = nibble_to_real_q1_3(raw_a >> 12)
            elif prec == 1:
                a_val = float(np.array((raw_a & 0xFF00) >> 8).astype(np.int8)) / (2**7)
            else:
                a_val = q1_15_to_real(raw_a)
            for j in range(E):
                raw_v = int(v[k_idx, 0, j])  # N=1
                if prec == 0:
                    v_val = nibble_to_real_q1_3(raw_v >> 12)
                elif prec == 1:
                    v_val = float(np.array((raw_v & 0xFF00) >> 8).astype(np.int8)) / (2**7)
                else:
                    v_val = q1_15_to_real(raw_v)
                av_out[i, j] += a_val * v_val
                av_out[i, j] = np.clip(av_out[i, j], -2.0, 1.999999999)
    av_out = np.clip(av_out, -1.0, 0.999969482421875)
    av_q15 = np.vectorize(real_to_q1_15)(av_out)

    # Step 6: W_O multiplication
    expected_out = np.zeros((L, E), dtype=float)
    for k_idx in range(E):
        for i in range(L):
            raw_av = int(av_q15[i, k_idx])
            av_val = q1_15_to_real(raw_av)
            for j in range(E):
                raw_wo = int(wo_np[k_idx, j])
                wo_val = q1_15_to_real(raw_wo)
                expected_out[i, j] += av_val * wo_val
        expected_out = np.clip(expected_out, -2.0, 1.999999999)
    out_q30 = np.vectorize(real_to_q1_30)(expected_out)

    out_q15 = []
    for q30_val in out_q30.flatten():
        sign_bit = (q30_val >> 31) & 1
        int_bit = (q30_val >> 30) & 1
        if sign_bit == int_bit:
            q15_val = (sign_bit << 15) | ((q30_val >> 15) & 0x7FFF)
        else:
            q15_val = 0x8000 if sign_bit else 0x7FFF
        if q15_val & 0x8000:
            q15_val = q15_val - 0x10000
        out_q15.append(q15_val)
```

```
        out_q15 = np.array(out_q15, dtype=np.int16).reshape(L, E)


        return out_q15

    def forward(self, x, quantize=False, save_for_verilog=False, batch_idx=0):
        if quantize:
            B = x.size(0)
            out = torch.zeros(B, L, E, device=DEVICE)
            x_np = (x * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
            wq_np = (self.att_q.w.detach() * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
            wk_np = (self.att_k.w.detach() * Q15_SCALE).round().clamp( -32768, 32767).to(torch.int16).cpu().numpy()
            wv_np = (self.att_v.w.detach() * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
            wo_np = (self.proj.w.detach() * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()

            if save_for_verilog:
                os.makedirs("verilog_inputs", exist_ok=True)
                np.save(f"verilog_inputs/wq_np.npy", wq_np)
                np.save(f"verilog_inputs/wk_np.npy", wk_np)
                np.save(f"verilog_inputs/wv_np.npy", wv_np)
                np.save(f"verilog_inputs/wo_np.npy", wo_np)

            for b in range(min(B, 10)):

                np.save(f"verilog_inputs/x_ln_np_{b}.npy", x_np[b])
                # print("After:")
                # print(wq_np.min(), wq_np.max())
                # print(wo_np.min(), wo_np.max())
                # print(wv_np.min(), wv_np.max())
                out_np = self.compute_attention(x_np[b], wq_np, wk_np, wv_np, wo_np)
                out[b] = torch.tensor(out_np, dtype=torch.float32, device=DEVICE) / Q15_SCALE
            return out
        else:
            q = self.att_q(x)
            k = self.att_k(x)
            v = self.att_v(x)
            att = ((q @ k.transpose(-2, -1)) / self.dim ** 0.5)
            att = (F.relu(att))                                        # ① clip negatives
            att_sum = (att.sum(-1, keepdim=True) + 1e-5)                   # ② avoid /0  (Q1.15: eps=1/32768≈0)
            att = (att / att_sum)
            #att = F.softmax(att, dim=-1)
            att_out = (att @ v)
            return (self.proj(att_out))

# ── Dataset: digits 0 and 1 only ──────────────────────────
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

mnist = datasets.MNIST(root='.', download=True, train=True, transform=transforms.ToTensor())
idx = [i for i, t in enumerate(mnist.targets) if t in (0, 1)]
train_set = Subset(mnist, idx)

mnist_test = datasets.MNIST(root='.', download=True, train=False, transform=transforms.ToTensor())
idx_t = [i for i, t in enumerate(mnist_test.targets) if t in (0, 1)]
test_set = Subset(mnist_test, idx_t)

train_loader = DataLoader(train_set, batch_size=BS, shuffle=True, drop_last=True)
test_loader = DataLoader(test_set, batch_size=BS, shuffle=False)

# ── Modified OneBlockViT with Custom Self-Attention ─────────
class OneBlockViT(nn.Module):
    def __init__(self, dim=32, n_patch=7):
        super().__init__()
        patch_sz = n_patch * n_patch
        n_tokens = (28 // n_patch) ** 2  # 16 patches
        self.n_tokens = n_tokens
        self.dim = dim
        self.patch = nn.Linear(patch_sz, dim, bias=False)
        self.pos_emb = nn.Parameter(torch.empty(1, n_tokens, dim))
        nn.init.trunc_normal_(self.pos_emb, std=0.02)
        self.norm1 = nn.LayerNorm(dim)
        self.norm2 = nn.LayerNorm(dim)
        self.attention = SelfAttention(dim=dim, n_tokens=n_tokens, n_heads=1)
        self.mlp = nn.Sequential(
            nn.Linear(dim, 4*dim), nn.ReLU(), nn.Linear(4*dim, dim))
```

```
        self.head = nn.Linear(dim, 1, bias=True)
        self.save_for_verilog = False

    def forward(self, x, quantize=False, save_images=None, save_labels=None):
        B = x.size(0)
        x = x.reshape(B, 1, 4, 7, 4, 7).permute(0, 2, 4, 1, 3, 5).reshape(B, -1, 49)
        x = self.patch(x)

        if quantize:
            #print(self.patch.weight.min(), self.patch.weight.max())
            # for b in range(min(B, 10)):
            #   np.save(f"verilog_inputs/patch_out_software_{b}.npy", x[b].cpu().numpy())
            # print(x.min(), x.max())
            # x = q15_round(x + q15_round(self.pos_emb))
            # x_ln = q15_round(self.norm1(x))
            x = x + self.pos_emb
            #print(x.min(), x.max())

            x_ln = self.norm1(x)
            att_out = self.attention(x_ln, quantize=True, save_for_verilog=self.save_for_verilog, batch_idx=0)

            x = x + att_out
            #print(x)
            y = self.norm2(x)
            x = x + self.mlp(y)
            x = x.mean(1)
            out = self.head(x).squeeze(1)
            # x = q15_round(x + att_out)
            # y = q15_round(self.norm2(x))
            # x = q15_round(x + q15_round(self.mlp(y)))
            # x = x.mean(1)
            # out = q15_round(self.head(x)).squeeze(1)
        else:
            x = x + self.pos_emb
            x_ln = self.norm1(x)
            att_out = self.attention(x_ln, quantize=False)
            x = x + att_out
            y = self.norm2(x)
            x = x + self.mlp(y)
            x = x.mean(1)
            out = self.head(x).squeeze(1)
        if self.save_for_verilog and save_images is not None and save_labels is not None:
            os.makedirs("verilog_inputs", exist_ok=True)
            for b in range(min(B, 10)):
                np.save(f"verilog_inputs/image_{b}.npy", save_images[b].cpu().numpy())
                #print(save_images[b].min(), save_images[b].max())
                np.save(f"verilog_inputs/label_{b}.npy", save_labels[b].cpu().numpy())
        return out

# ── Training Function ───────────────────────────────
def train_model(model, model_name):
    opt = torch.optim.Adam(model.parameters(), lr=LR, weight_decay=1e-5)
    scheduler = optim.lr_scheduler.StepLR(opt, step_size=5, gamma=0.5)
    crit = nn.BCEWithLogitsLoss()

    print(f"\nTraining {model_name} (Full Precision)")
    for epoch in range(1, EPOCHS+1):
        model.train()
        running_loss = 0
        pbar = tqdm(train_loader, desc=f"Epoch {epoch}", leave=False)
        for img, lbl in pbar:
            img, lbl = img.to(DEVICE), lbl.float().to(DEVICE)
            opt.zero_grad()
            out = model(img, quantize=True)
            loss = crit(out, lbl)
            loss.backward()
            opt.step()
            running_loss += loss.item()
            pbar.set_postfix(loss=float(loss))
        scheduler.step()
        print(f"[{model_name} Epoch {epoch}] Loss: {running_loss / len(train_loader):.3f}")

# ── Evaluation Function: Generate Inputs for Verilog ───────
def evaluate_generate_inputs(model, model_name):
    model.eval()
    print(f"\nEvaluating {model_name} (Generate Inputs for Verilog)")
    saved_count = 0
```

```
    model.save_for_verilog = True
    with torch.no_grad():
        for img, lbl in test_loader:
            img, lbl = img.to(DEVICE), lbl.float().to(DEVICE)
            print("label", lbl)
            model(img, quantize=True, save_images=img, save_labels=lbl)
            break
    model.save_for_verilog = False
    print(f"Saved inputs for {saved_count} examples in 'verilog_inputs'")

# ── Main Execution ───────────────────────────────
model = OneBlockViT().to(DEVICE)
train_model(model, "ViT")
evaluate_generate_inputs(model, "ViT")
# After hardware simulation, run evaluate_with_hardware(model, "ViT")

# ── Export Q1.15 weights ───────────────────────────────
int16_state = {k: torch.round(v * Q15_SCALE).to(torch.int16).cpu()
               for k, v in model.state_dict().items()}
torch.save(int16_state, "vit_q15_mixed_int16.pt")
print("Saved quantized weights to vit_q15_mixed_int16.pt")
```

```
Training ViT (Full Precision)
[ViT Epoch 1] Loss: 0.129
[ViT Epoch 2] Loss: 0.017
[ViT Epoch 3] Loss: 0.012
[ViT Epoch 4] Loss: 0.010
[ViT Epoch 5] Loss: 0.008
[ViT Epoch 6] Loss: 0.005
[ViT Epoch 7] Loss: 0.004
[ViT Epoch 8] Loss: 0.004
[ViT Epoch 9] Loss: 0.003
[ViT Epoch 10] Loss: 0.003

Evaluating ViT (Generate Inputs for Verilog)
label tensor([1., 0., 1., 0., 0., 1., 0., 0., 1., 1., 1., 1., 1., 1., 0., 1., 0., 0.,
        1., 1., 1., 1., 0., 1., 0., 1., 0., 1., 1., 1., 0., 1., 0., 1., 1., 1.,
        1., 0., 0., 1., 1., 1., 0., 0., 1., 1., 1., 1., 0., 1., 1., 1., 0., 1.,
        1., 1., 0., 0., 1., 1., 1., 1., 0., 0., 0., 1., 0., 0., 0., 1., 0., 0.,
        1., 1., 1., 1., 1., 0., 1., 1., 1., 1., 0., 0., 1., 1., 0., 1., 1., 1.,
        0., 1., 1., 1., 0., 1., 1., 0., 0., 0., 1., 0., 1., 1., 1., 0., 1., 0.,
        0., 1., 1., 1., 0., 1., 1., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.,
        0., 0.], device='cuda:0')
Saved inputs for 0 examples in 'verilog_inputs'
Saved quantized weights to vit_q15_mixed_int16.pt
```

```
a = np.load("verilog_inputs/wq_np.npy")
print(a.min(), a.max())
```

```
-10352 12130
```

```
!rm -rf verilog_inputs*
!rm -rf verilog_outputs*
```

```
# Evaluate with quantization
model.eval()
model_name = "ViT"
print(f"\nEvaluating {model_name} (Quantized Full Precision)")
correct_full = 0
total = 0

with torch.no_grad():
    for img, lbl in test_loader:
        img, lbl = img.to(DEVICE), lbl.float().to(DEVICE)
        pred = torch.sign(model(img, quantize=True))
        print(pred)
        correct_full += (pred == (lbl*2-1)).sum().item()
        total += lbl.size(0)
acc_full = 100 * correct_full / total
print(f"[{model_name}] Quantized Mixed Precision Test Accuracy: {acc_full:.2f}%")

print(f"\nEvaluating {model_name} (Quantized Mixed Precision)")
correct_mixed = 0
total = 0
with torch.no_grad():
```

```
    for img, lbl in test_loader:
        img, lbl = img.to(DEVICE), lbl.float().to(DEVICE)
        pred = torch.sign(model(img, quantize=False))
        correct_mixed += (pred == (lbl*2-1)).sum().item()
        total += lbl.size(0)
acc_mixed = 100 * correct_mixed / total
print(f"[{model_name}] Quantized Full Precision Test Accuracy: {acc_mixed:.2f}%")

```

```
         1.,  1.,  1., -1.,  1.,  1., -1., -1.,  1.,  1.,  1.,  1., -1., -1.,
        -1., -1.,  1., -1.,  1., -1.,  1., -1.,  1., -1., -1.,  1., -1., -1.,
         1., -1., -1.,  1.,  1.,  1.,  1.,  1., -1., -1.,  1., -1., -1.,  1.,
        -1.,  1., -1.,  1., -1., -1.,  1.,  1., -1.,  1., -1., -1.,  1., -1.,
         1., -1., -1.,  1.,  1., -1.,  1.,  1., -1.,  1.,  1., -1.,  1., -1.,
        -1., -1.,  1., -1.,  1., -1.,  1.,  1., -1.,  1.,  1.,  1., -1., -1.,
         1., -1., -1.,  1., -1.,  1., -1.,  1.,  1.,  1., -1.,  1.,  1.,  1.,
        -1., -1.], device='cuda:0')
tensor([ 1., -1.,  1.,  1., -1., -1., -1., -1.,  1., -1., -1.,  1.,  1.,  1.,
        -1.,  1.,  1.,  1.,  1., -1.,  1., -1.,  1., -1., -1.,  1.,  1., -1.,
         1.,  1., -1., -1., -1., -1.,  1., -1.,  1., -1.,  1., -1., -1., -1.,
         1., -1.,  1.,  1., -1.,  1.,  1.,  1.,  1., -1., -1.,  1., -1., -1.,
         1., -1.,  1., -1.,  1.,  1.,  1., -1., -1.,  1.,  1.,  1., -1.,  1.,
        -1., -1., -1., -1., -1., -1.,  1.,  1.,  1., -1.,  1., -1.,  1., -1.,
         1., -1.,  1., -1., -1.,  1.,  1.,  1., -1.,  1.,  1., -1.,  1.,  1.,
        -1., -1.,  1., -1.,  1., -1., -1.,  1., -1.,  1., -1.,  1., -1., -1.,
         1., -1.,  1., -1., -1., -1.,  1.,  1., -1.,  1.,  1., -1.,  1., -1.,
         1.,  1.], device='cuda:0')
tensor([ 1., -1.,  1., -1.,  1., -1.,  1., -1.,  1.,  1.,  1., -1., -1.,  1.,
         1.,  1., -1., -1.,  1., -1.,  1., -1., -1.,  1.,  1., -1., -1., -1.,
         1., -1.,  1., -1.,  1., -1.,  1.,  1.,  1., -1.,  1.,  1., -1.,  1.,
        -1., -1.,  1., -1.,  1., -1.,  1., -1.,  1., -1.,  1.,  1., -1., -1.,
         1., -1., -1., -1., -1.,  1., -1., -1.,  1.,  1.,  1.,  1., -1.,  1.,
        -1.,  1., -1.,  1.,  1., -1., -1., -1.,  1.,  1.,  1.,  1.,  1., -1.,
         1., -1., -1., -1.,  1., -1., -1.,  1., -1.,  1., -1.,  1.,  1., -1.,
         1., -1., -1., -1.,  1.,  1., -1.,  1.,  1., -1.,  1.,  1.,  1.,  1.,
        -1.,  1., -1.,  1., -1.,  1., -1.,  1., -1.,  1.,  1., -1., -1.,  1.,
        -1., -1.], device='cuda:0')
tensor([ 1., -1., -1.,  1.,  1.,  1.,  1.,  1., -1.,  1., -1.,  1., -1.,  1.,
         1., -1.,  1., -1., -1.,  1.,  1., -1., -1., -1., -1., -1.,  1., -1.,
         1.,  1., -1.,  1.,  1.,  1., -1.,  1., -1.,  1., -1.,  1.,  1.,  1.,
         1.,  1.,  1., -1., -1., -1.,  1.,  1., -1., -1.,  1., -1., -1., -1.,
         1., -1.,  1., -1., -1.,  1., -1.,  1., -1.,  1., -1., -1.,  1., -1.,
        -1., -1.,  1., -1., -1.,  1.,  1., -1.,  1.,  1., -1.,  1., -1.,  1.,
        -1.,  1., -1.,  1.,  1., -1.,  1.,  1., -1.,  1.,  1., -1.,  1.,  1.,
        -1., -1.,  1., -1.,  1.,  1., -1.,  1., -1.,  1.,  1.,  1., -1., -1.,
         1.,  1.,  1., -1., -1.,  1., -1.,  1., -1.,  1.,  1., -1.,  1., -1.,
         1.,  1.], device='cuda:0')
tensor([ 1.,  1., -1., -1.,  1.,  1.,  1., -1., -1., -1.,  1., -1., -1.,  1.,
        -1.,  1., -1.,  1., -1., -1.,  1., -1., -1.,  1., -1., -1.,  1., -1.,
         1.,  1., -1.,  1., -1.,  1., -1.,  1.,  1.,  1., -1., -1.,  1., -1.,
        -1.,  1.,  1., -1.,  1.,  1., -1.,  1.,  1., -1.,  1., -1., -1.,  1.,
        -1.,  1., -1.,  1.,  1., -1.,  1., -1.,  1., -1.,  1., -1.,  1.,  1.,
        -1.,  1., -1., -1.,  1.,  1., -1., -1., -1.,  1.,  1., -1.,  1., -1.,
         1., -1.,  1.,  1., -1.,  1., -1., -1.,  1., -1.,  1., -1.,  1.,  1.,
        -1.,  1.,  1.,  1.,  1., -1.,  1., -1.,  1., -1.,  1., -1., -1., -1.,
         1.,  1.,  1., -1., -1.,  1.,  1.,  1.,  1., -1., -1.,  1., -1.,  1.,
        -1., -1.], device='cuda:0')
tensor([ 1.,  1., -1.,  1., -1.,  1., -1.,  1.,  1., -1., -1., -1.,  1., -1.,
        -1.,  1.,  1., -1.,  1.,  1.,  1., -1., -1.,  1., -1.,  1., -1.,  1.,
        -1.,  1.,  1.,  1.,  1.,  1.,  1., -1., -1.,  1.,  1.,  1., -1.,  1.,
        -1.,  1., -1.,  1., -1.,  1.,  1., -1.,  1., -1.,  1., -1.,  1., -1.,
        -1.,  1.,  1., -1., -1.,  1.,  1., -1.,  1., -1.,  1.],
       device='cuda:0')
[ViT] Quantized Mixed Precision Test Accuracy: 99.86%

Evaluating ViT (Quantized Mixed Precision)
[ViT] Quantized Full Precision Test Accuracy: 98.77%
```

```
!zip -r verilog_inputs.zip verilog_inputs

```

```
updating: verilog_inputs/ (stored 0%)
updating: verilog_inputs/image_8.npy (deflated 91%)
updating: verilog_inputs/x_ln_np_0.npy (deflated 11%)
updating: verilog_inputs/label_2.npy (deflated 47%)
updating: verilog_inputs/label_3.npy (deflated 48%)
updating: verilog_inputs/image_7.npy (deflated 84%)
updating: verilog_inputs/image_1.npy (deflated 85%)
updating: verilog_inputs/image_4.npy (deflated 83%)
updating: verilog_inputs/wq_np.npy (deflated 6%)
updating: verilog_inputs/label_5.npy (deflated 47%)
updating: verilog_inputs/wo_np.npy (deflated 6%)
```

```
updating: verilog_inputs/wk_np.npy (deflated 6%)
updating: verilog_inputs/image_3.npy (deflated 83%)
updating: verilog_inputs/label_8.npy (deflated 47%)
updating: verilog_inputs/image_0.npy (deflated 89%)
updating: verilog_inputs/x_ln_np_7.npy (deflated 11%)
updating: verilog_inputs/x_ln_np_1.npy (deflated 10%)
updating: verilog_inputs/x_ln_np_6.npy (deflated 10%)
updating: verilog_inputs/x_ln_np_2.npy (deflated 11%)
updating: verilog_inputs/wv_np.npy (deflated 6%)
updating: verilog_inputs/label_6.npy (deflated 48%)
updating: verilog_inputs/image_2.npy (deflated 89%)
updating: verilog_inputs/label_4.npy (deflated 48%)
updating: verilog_inputs/label_0.npy (deflated 47%)
updating: verilog_inputs/x_ln_np_3.npy (deflated 11%)
updating: verilog_inputs/x_ln_np_8.npy (deflated 10%)
updating: verilog_inputs/x_ln_np_5.npy (deflated 11%)
updating: verilog_inputs/label_9.npy (deflated 47%)
updating: verilog_inputs/x_ln_np_4.npy (deflated 10%)
updating: verilog_inputs/image_5.npy (deflated 95%)
updating: verilog_inputs/x_ln_np_9.npy (deflated 10%)
updating: verilog_inputs/label_1.npy (deflated 48%)
updating: verilog_inputs/image_9.npy (deflated 92%)
updating: verilog_inputs/image_6.npy (deflated 82%)
updating: verilog_inputs/label_7.npy (deflated 48%)
```

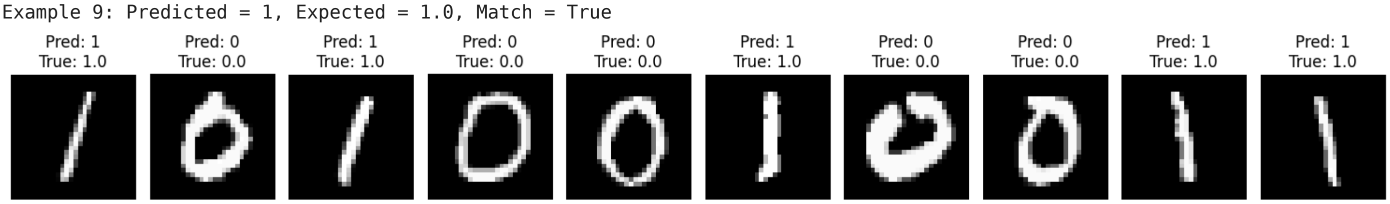
```
!unzip -q verilog_outputs.zip

```

```
replace verilog_outputs/self_attention_out_6.npy? [y]es, [n]o, [A]ll, [N]one, [r]ename: A
```

```
import torch
import numpy as np


# —— Evaluation Function: Load Hardware Outputs and Predict ——
def evaluate_with_hardware(model, model_name):
    model.eval()
    print(f"\nEvaluating {model_name} (Quantized Mixed Precision with Hardware)")
    correct_hardware = 0
    total = 0
    saved_count = 0
    print("\nComparing Hardware Predictions with Labels:")
    plt.figure(figsize=(15, 3))
    with torch.no_grad():
        for img, lbl in test_loader:
            img, lbl = img.to(DEVICE), lbl.float().to(DEVICE)
            B = img.size(0)
            for b in range(min(B, 10 - saved_count)):
                out_mem = np.load(f"verilog_outputs/self_attention_out_{b}.npy")
                image = np.load(f"verilog_inputs/image_{b}.npy")

                label = np.load(f"verilog_inputs/label_{b}.npy")

                image_t  = torch.tensor(image, dtype=torch.float32, device=DEVICE)  # (1,28,28)
                image_t  = image_t.unsqueeze(0)

                patches = image_t.reshape(1, 1, 4, 7, 4, 7).permute(0,2,4,1,3,5).reshape(1, -1, 49)

                x=model.patch(patches)
                #-----------debug---------------------
                # x_hardware = model.patch(patches)

                # # Load software patch output and compare
                # x_software = np.load(f"verilog_outputs/patch_out_software_{b}.npy")
                # x_hardware_np = x_hardware[0].cpu().numpy()
                # diff = np.max(np.abs(x_software - x_hardware_np))
                # print(f"Patch embedding max difference for example {b}: {diff}")

                # print(x.min(), x.max())
                #-----------debug----------------------


                x = ( x + model.pos_emb)   # add the skip path
                #print(x.min, x.max)

                att_out = torch.tensor(out_mem.astype(np.int16), dtype=torch.float32, device=DEVICE) / Q15_SCALE
                att_out = att_out.unsqueeze(0)

```

```

                x_ln = (model.norm1(x))
                x = (x + (att_out))
                #print(x.min(), x.max())

                y = (model.norm2(x))
                x = (x + (model.mlp(y)))
                x = x.mean(1)
                #print(model.head.weight)
                out = (model.head(x)).squeeze(1)

                pred = torch.sign(out).item()
                expected = (2 * label - 1)


                # image_t = torch.tensor(image, dtype=torch.float32, device=DEVICE).unsqueeze(0)
                # patches = image_t.reshape(1, 1, 4, 7, 4, 7).permute(0, 2, 4, 1, 3, 5).reshape(1, -1, 49)
                # x = q15_round(model.patch(patches) + model.pos_emb)
                # x = q15_round(x +  q15_round(att_out))
                # y = q15_round(model.norm2(x))
                # x = q15_round(x + q15_round(model.mlp(y)))
                # x = x.mean(1)
                # out = q15_round(model.head(x)).squeeze(1)
                # pred = torch.sign(out).item()
                # expected = (2 * label - 1)
                print(f"Example {b}: Predicted = {0 if pred == -1 else 1}, Expected = {label}, Match = {pred == expected
                correct_hardware += (pred == expected)
                total += 1
                plt.subplot(1, 10, b+1)
                plt.imshow(image.squeeze(), cmap='gray')
                plt.title(f"Pred: {0 if pred == -1 else 1}\nTrue: {label}")
                plt.axis('off')
                saved_count += 1
            if saved_count >= 10:
                break
    plt.tight_layout()
    plt.show()
    acc_hardware = 100 * correct_hardware / total if total > 0 else 0
    print(f"[{model_name}] Quantized Mixed Precision Hardware Test Accuracy: {acc_hardware:.2f}%")
    return acc_hardware



# Load the model and weights (assuming vit_q15_mixed_int16.pt is available)
# model = OneBlockViT().to(DEVICE)
# state = torch.load("vit_q15_mixed_int16.pt") #no influence
# state = {k: v.to(torch.float32) / Q15_SCALE for k, v in state.items()}
# model.load_state_dict(state)

# Run the hardware evalyuation
evaluate_with_hardware(model, "ViT")
```

```
Evaluating ViT (Quantized Mixed Precision with Hardware)

Comparing Hardware Predictions with Labels:
Example 0: Predicted = 1, Expected = 1.0, Match = True
Example 1: Predicted = 0, Expected = 0.0, Match = True
Example 2: Predicted = 1, Expected = 1.0, Match = True
Example 3: Predicted = 0, Expected = 0.0, Match = True
Example 4: Predicted = 0, Expected = 0.0, Match = True
Example 5: Predicted = 1, Expected = 1.0, Match = True
Example 6: Predicted = 0, Expected = 0.0, Match = True
Example 7: Predicted = 0, Expected = 0.0, Match = True
Example 8: Predicted = 1, Expected = 1.0, Match = True
Example 9: Predicted = 1, Expected = 1.0, Match = True
```




```
[ViT] Quantized Mixed Precision Hardware Test Accuracy: 100.00%
np.float64(100.0)
```