

DEMO

```

1 # PTQ for AV multiplication
2 %matplotlib inline
3 import torch
4 import torch.nn as nn
5 import torch.nn.functional as F
6 import torch.optim as optim
7 from torchvision import datasets, transforms
8 from torch.utils.data import DataLoader, Subset
9 from tqdm import tqdm
10 import numpy as np
11 import matplotlib.pyplot as plt
12 import os
13
14 DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
15 Q15_SCALE = 2**15 # 32768
16 EPOCHS = 10
17 BS = 128
18 LR = 0.002
19
20 # Hardware module parameters (aligned with testbench)
21 A_ROWS = 16
22 V_COLS = 32
23 NUM_COLS = 16
24 TILE_SIZE = 8
25 WIDTH_INT4 = 4
26 WIDTH_INT8 = 8
27 WIDTH_FP16 = 16
28 NUM_TILES = V_COLS // TILE_SIZE # 4
29 TOLERANCE = 0.001
30
31 # — Fixed-point helpers for PTQ —————
32 def q15_round(x):
33     x_clp = torch.clamp(x, -1.0, 1.0 - 1/32768)
34     return (x_clp * Q15_SCALE).round() / Q15_SCALE
35
36 def real_to_q15(x):
37     val = int(x * Q15_SCALE)
38     val = max(min(val, 32767), -32768)
39     return np.int16(val)
40
41 def q15_to_real(x):
42     value = np.array(int(x) & 0xFFFF).astype(np.int16)
43     return float(value) / Q15_SCALE
44
45 def real_to_q1_3(x):
46     val = int(x * (2**3))
47     val = max(min(val, 7), -8)
48     return val & 0xF
49
50 def real_to_q1_7(x):
51     val = int(x * (2**7))
52     val = max(min(val, 127), -128)
53     return val & 0xFF
54
55 def nibble_to_real_q1_3(n):
56     n = n & 0xF
57     if n & 0x8:
58         n -= 0x10
59     return float(n) / (2**3)
60
61 # — Software Precision Assigner —————
62 def assign_precision(A_matrix):
63     sums = np.sum(A_matrix.astype(np.int32), axis=1) # Shape: (B, L)
64     sums_q2_30 = sums * (2**15) # Shift Q1.15 to Q2.30
65     precision_codes = []
66     for batch_sums in sums_q2_30:
67         codes = []
68         for s in batch_sums:
69             if s < 30000 * (2**15): # INT4
70                 codes.append(0)
71             elif s < 40000 * (2**15): # INT8
72                 codes.append(1)
73             else: # FP16

```

```

74         codes.append(2)
75         precision_codes.append(codes)
76     return np.array(precision_codes, dtype=np.int32) # Shape: (B, L)
77
78 # — Software Simulation of attention_av_multiply.sv —————
79 def simulate_av_multiply(a_mem, v_mem, precision_sel):
80     B = a_mem.shape[0]
81     out_mem = np.zeros((B, A_ROWS, V_COLS), dtype=np.int16)
82     for b in range(B):
83         expected_out = np.zeros((A_ROWS, V_COLS), dtype=np.float64)
84         for k in range(NUM_COLS):
85             prec = precision_sel[b, k]
86             for i in range(A_ROWS):
87                 raw_a = int(a_mem[b, i, k])
88                 if prec == 0: # INT4
89                     a_val = nibble_to_real_q1_3(raw_a >> 12)
90                 elif prec == 1: # INT8
91                     a_val = float(np.array((raw_a & 0xFF00) >> 8).astype(np.int8)) / (2**7)
92                 else: # FP16
93                     a_val = q15_to_real(raw_a)
94                 for j in range(V_COLS):
95                     raw_v = int(v_mem[b, k, j])
96                     if prec == 0:
97                         v_val = nibble_to_real_q1_3(raw_v >> 12)
98                     elif prec == 1:
99                         v_val = float(np.array((raw_v & 0xFF00) >> 8).astype(np.int8)) / (2**7)
100                     else:
101                         v_val = q15_to_real(raw_v)
102                     expected_out[i, j] += a_val * v_val
103                     expected_out[i, j] = np.clip(expected_out[i, j], -2.0, 1.999999999)
104             expected_q = np.vectorize(real_to_q15)(np.clip(expected_out, -1.0, 0.999969))
105             out_mem[b] = expected_q.astype(np.int16)
106     return out_mem
107
108 # — Dataset: digits 0 and 1 only —————
109 transform = transforms.Compose([
110     transforms.ToTensor(),
111     transforms.Normalize((0.5,), (0.5,))
112 ])
113
114 mnist = datasets.MNIST(root='.', download=True, train=True, transform=transforms.ToTensor())
115 idx = [i for i, t in enumerate(mnist.targets) if t in (0, 1)]
116 train_set = Subset(mnist, idx)
117
118 mnist_test = datasets.MNIST(root='.', download=True, train=False, transform=transforms.ToTensor())
119 idx_t = [i for i, t in enumerate(mnist_test.targets) if t in (0, 1)]
120 test_set = Subset(mnist_test, idx_t)
121
122 train_loader = DataLoader(train_set, batch_size=BS, shuffle=True, drop_last=True)
123 test_loader = DataLoader(test_set, batch_size=BS, shuffle=False)
124
125 # — One-block ViT with PTQ and Mixed-Precision A·V —————
126 class OneBlockViT(nn.Module):
127     def __init__(self, dim=32, n_patch=7):
128         super().__init__()
129         patch_sz = n_patch * n_patch
130         n_tokens = (28 // n_patch) ** 2 # 16 patches
131         self.n_tokens = n_tokens
132         self.dim = dim
133         self.patch = nn.Linear(patch_sz, dim, bias=False)
134         self.pos_emb = nn.Parameter(torch.empty(1, n_tokens, dim))
135         nn.init.trunc_normal_(self.pos_emb, std=0.02)
136         self.norm1 = nn.LayerNorm(dim)
137         self.norm2 = nn.LayerNorm(dim)
138         self.att_q = nn.Linear(dim, dim, bias=False)
139         self.att_k = nn.Linear(dim, dim, bias=False)
140         self.att_v = nn.Linear(dim, dim, bias=False)
141         self.proj = nn.Linear(dim, dim, bias=False)
142         self.mlp = nn.Sequential(
143             nn.Linear(dim, 4*dim), nn.ReLU(), nn.Linear(4*dim, dim))
144         self.head = nn.Linear(dim, 1, bias=True)
145         self.save_for_verilog = False
146
147     def forward(self, x, quantize=False, use_mixed_precision=False, save_images=None, save_labels=None):
148         B = x.size(0)
149         x = x.reshape(B, 1, 4, 7, 4, 7).permute(0, 2, 4, 1, 3, 5).reshape(B, -1, 49)
150         x = self.patch(x)

```

```

151     if quantize:
152         x = q15_round(x + q15_round(self.pos_emb))
153         x_ln = q15_round(self.norm1(x))
154         q, k, v = self.att_q(q15_round(x_ln)), self.att_k(q15_round(x_ln)), self.att_v(q15_round(x_ln))
155         wq = self.att_q.weight.detach()
156         wk = self.att_k.weight.detach()
157         wv = self.att_v.weight.detach()
158         wo = self.proj.weight.detach()
159         q, k, v = q15_round(q), q15_round(k), q15_round(v)
160         att = q15_round((q @ k.transpose(-2, -1)) / (k.size(-1)*0.5))
161         att = F.softmax(att, dim=-1)
162         if use_mixed_precision:
163             x_ln_np = (x_ln * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
164             att_np = (att * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
165             v_np = (v * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
166             wq_np = (wq * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
167             wk_np = (wk * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
168             wv_np = (wv * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
169             wo_np = (wo * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
170             precision_sel = assign_precision(att_np)
171             if self.save_for_verilog and save_images is not None and save_labels is not None:
172                 os.makedirs("verilog_inputs", exist_ok=True)
173                 np.save(f"verilog_inputs/wq_np.npy", wq_np)
174                 np.save(f"verilog_inputs/wk_np.npy", wk_np)
175                 np.save(f"verilog_inputs/wv_np.npy", wv_np)
176                 np.save(f"verilog_inputs/wo_np.npy", wo_np)
177
178
179             for b in range(min(B, 10)):
180                 np.save(f"verilog_inputs/x_ln_np_{b}.npy", x_ln_np[b])
181
182                 np.save(f"verilog_inputs/att_np_{b}.npy", att_np[b])
183                 np.save(f"verilog_inputs/v_np_{b}.npy", v_np[b])
184                 np.save(f"verilog_inputs/prec_sel_{b}.npy", precision_sel[b])
185                 np.save(f"verilog_inputs/image_{b}.npy", save_images[b].cpu().numpy())
186                 np.save(f"verilog_inputs/label_{b}.npy", save_labels[b].cpu().numpy())
187             att_out = torch.zeros_like(v, device=DEVICE)
188             for chunk in range(self.n_tokens // A_ROWS):
189                 a_chunk = att_np[:, chunk*A_ROWS:(chunk+1)*A_ROWS, :]
190                 v_chunk = v_np
191                 prec_chunk = precision_sel[:, chunk*NUM_COLS:(chunk+1)*NUM_COLS]
192                 out_chunk = simulate_av_multiply(a_chunk, v_chunk, prec_chunk)
193                 att_out[:, chunk*A_ROWS:(chunk+1)*A_ROWS, :] = torch.tensor(out_chunk, dtype=torch.float32, dev:
194             else:
195                 att_out = q15_round(att @ v)
196                 x = q15_round(x + q15_round(self.proj(att_out)))
197                 y = q15_round(self.norm2(x))
198                 x = q15_round(x + q15_round(self.mlp(y)))
199                 x = x.mean(1)
200                 out = q15_round(self.head(x)).squeeze(1)
201         else:
202             x = x + self.pos_emb
203             x_ln = self.norm1(x)
204             q, k, v = self.att_q(x_ln), self.att_k(x_ln), self.att_v(x_ln)
205             att = (q @ k.transpose(-2, -1)) / (k.size(-1)*0.5)
206             att = F.softmax(att, dim=-1)
207             att_out = att @ v
208             x = x + self.proj(att_out)
209             y = self.norm2(x)
210             x = x + self.mlp(y)
211             x = x.mean(1)
212             out = self.head(x).squeeze(1)
213         return out
214
215 # — Training Function —————
216 def train_model(model, model_name):
217     opt = torch.optim.Adam(model.parameters(), lr=LR, weight_decay=1e-5)
218     scheduler = optim.lr_scheduler.StepLR(opt, step_size=5, gamma=0.5)
219     crit = nn.BCEWithLogitsLoss()
220
221     print(f"\nTraining {model_name} (Full Precision)")
222     for epoch in range(1, EPOCHS+1):
223         model.train()
224         running_loss = 0
225         pbar = tqdm(train_loader, desc=f"Epoch {epoch}", leave=False)
226         for img, lbl in pbar:
227             img, lbl = img.to(DEVICE), lbl.float().to(DEVICE)

```

```

228         opt.zero_grad()
229         out = model(img, quantize=False)
230         loss = crit(out, lbl)
231         loss.backward()
232         opt.step()
233         running_loss += loss.item()
234         pbar.set_postfix(loss=float(loss))
235     scheduler.step()
236     print(f"[{model_name}] Epoch {epoch}] Loss: {running_loss / len(train_loader):.3f}")
237
238 # — Evaluation Function: Generate Inputs for Verilog —
239 def evaluate_generate_inputs(model, model_name):
240     model.eval()
241     print(f"\nEvaluating {model_name} (Generate Inputs for Verilog)")
242     saved_count = 0
243     model.save_for_verilog = True
244     with torch.no_grad():
245         for img, lbl in test_loader:
246             img, lbl = img.to(DEVICE), lbl.float().to(DEVICE)
247             print("label", lbl)
248             model(img, quantize=True, use_mixed_precision=True, save_images=img, save_labels=lbl)
249             saved_count += min(lbl.size(0), 10 - saved_count)
250             if saved_count >= 10:
251                 break
252     model.save_for_verilog = False
253     print(f"Saved inputs for {saved_count} examples in 'verilog_inputs'")
254
255 # — Evaluation Function: Load Hardware Outputs and Predict —
256 def evaluate_with_hardware(model, model_name):
257     model.eval()
258     print(f"\nEvaluating {model_name} (Quantized Mixed Precision with Hardware)")
259     correct_hardware = 0
260     total = 0
261     saved_count = 0
262     print("\nComparing Hardware Predictions with Labels:")
263     plt.figure(figsize=(15, 3))
264     with torch.no_grad():
265         for img, lbl in test_loader:
266             img, lbl = img.to(DEVICE), lbl.float().to(DEVICE)
267             B = img.size(0)
268             for b in range(min(B, 10 - saved_count)):
269                 if not os.path.exists(f"verilog_outputs/out_mem_{b}.npy"):
270                     print(f"Hardware output 'out_mem_{b}.npy' not found")
271                     continue
272                 out_mem = np.load(f"verilog_outputs/out_mem_{b}.npy")
273                 image = np.load(f"verilog_inputs/image_{b}.npy")
274                 label = np.load(f"verilog_inputs/label_{b}.npy")
275
276                 att_out = torch.tensor(out_mem, dtype=torch.float32, device=DEVICE) / Q15_SCALE
277                 att_out = att_out.unsqueeze(0)
278
279
280                 image_t = torch.tensor(image, dtype=torch.float32, device=DEVICE) # (1,28,28)
281                 image_t = image_t.unsqueeze(0)
282
283                 patches = image_t.reshape(1, 1, 4, 7, 4, 7).permute(0,2,4,1,3,5).reshape(1, -1, 49)
284                 x = q15_round(model.patch(patches) + model.pos_emb) # add the skip path
285
286
287                 x = q15_round(x + q15_round(model.proj(att_out)))
288                 y = q15_round(model.norm2(x))
289                 x = q15_round(x + q15_round(model.mlp(y)))
290                 x = x.mean(1)
291                 out = q15_round(model.head(x)).squeeze(1)
292                 pred = torch.sign(out).item()
293                 expected = (2 * label - 1)
294                 print(f"Example {b}: Predicted = {0 if pred == -1 else 1}, Expected = {label}, Match = {pred == expected}")
295                 correct_hardware += (pred == expected)
296                 total += 1
297
298                 plt.subplot(1, 10, b+1)
299                 plt.imshow(image.squeeze(), cmap='gray')
300                 plt.title(f"Pred: {0 if pred == -1 else 1}\nTrue: {label}")
301                 plt.axis('off')
302                 saved_count += 1
303             if saved_count >= 10:
304                 break

```

```

305 plt.tight_layout()
306 plt.show()
307 acc_hardware = 100 * correct_hardware / total if total > 0 else 0
308 print(f"[{model_name}] Quantized Mixed Precision Hardware Test Accuracy: {acc_hardware:.2f}%")
309 return acc_hardware
310
311 # — Main Execution —————
312 model = OneBlockViT().to(DEVICE)
313 train_model(model, "ViT")
314 # Run this in the first Colab cell to generate inputs
315 evaluate_generate_inputs(model, "ViT")
316 # After hardware simulation, run evaluate_with_hardware in a second Colab cell
317
318 # — Export Q1.15 weights —————
319 int16_state = {k: torch.round(v * Q15_SCALE).to(torch.int16).cpu()
320               for k, v in model.state_dict().items()}
321 torch.save(int16_state, "vit_q15_mixed_int16.pt")
322 print("Saved quantized weights to vit_q15_mixed_int16.pt")
323

```



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., 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.,
              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

```

1 Start coding or [generate](#) with AI.

```

1 # QAT for self attention layer
2 %matplotlib inline
3 import torch
4 import torch.nn as nn
5 import torch.nn.functional as F
6 import torch.optim as optim
7 from torchvision import datasets, transforms
8 from torch.utils.data import DataLoader, Subset
9 from tqdm import tqdm
10 import numpy as np
11 import matplotlib.pyplot as plt
12 import os
13
14 DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
15 Q15_SCALE = 2**15 # 32768
16 EPOCHS = 10
17 BS = 128
18 LR = 0.002
19
20 # Hardware module parameters (aligned with self_attention_top.sv and testbench)
21 DATA_WIDTH = 16
22 L = 16 # Sequence length
23 E = 32 # Embedding dimension
24 N = 1 # Number of attention heads
25 OUT_DATA_WIDTH = 32 # Q2.30 for matmul outputs
26 TOLERANCE = 0.005
27 SQRT_E = np.sqrt(32.0) # ≈ 5.656854
28 INV_SQRT_E = 1.0 / SQRT_E # ≈ 0.176776695
29 INV_SQRT_E_Q15 = int(INV_SQRT_E * (2**15)) # ≈ 5792 in Q1.15
30

```

```

31 # — Fixed-point helpers (from testbench) —————
32 def real_to_q1_15(x):
33     val = int(x * (2**15))
34     val = max(min(val, 32767), -32768)
35     return np.int16(val)
36
37 def q1_15_to_real(x):
38     value = np.array(int(x) & 0xFFFF).astype(np.int16)
39     return float(value) / (2**15)
40
41 def real_to_q1_30(x):
42     val = int(x * (2**30))
43     val = max(min(val, 2**31 - 1), -(2**31))
44     return np.int32(val)
45
46 def q1_30_to_real(x):
47     value = np.array(int(x) & 0xFFFFFFFF).astype(np.int32)
48     return float(value) / (2**30)
49
50 def real_to_q1_3(x):
51     val = int(x * (2**3))
52     val = max(min(val, 7), -8)
53     return val & 0xF
54
55 def nibble_to_real_q1_3(n):
56     n = n & 0xF
57     if n & 0x8:
58         n -= 0x10
59     return float(n) / (2**3)
60
61 def real_to_q1_7(x):
62     val = int(x * (2**7))
63     val = max(min(val, 127), -128)
64     return val & 0xFF
65
66 def q1_7_to_real(x):
67     x = x & 0xFF
68     if x & 0x80:
69         x -= 0x100
70     return float(x) / (2**7)
71
72
73 def hw_multiply_inv_sqrt_e(matmul_result_q30):
74     matmul_signed = np.array(matmul_result_q30).astype(np.int32)
75     inv_sqrt8_signed = np.int16(INV_SQRT_E_Q15)
76     mult_result = np.int64(matmul_signed) * np.int64(inv_sqrt8_signed)
77     q30_result = (mult_result >> 15) & 0xFFFFFFFF
78     if q30_result & 0x80000000:
79         q30_result = q30_result - 0x100000000
80     return np.int32(q30_result)
81
82 class _RoundSTE(torch.autograd.Function):
83     @staticmethod
84     def forward(ctx, x):
85         # quantise to the nearest Q1.15 integer
86         return torch.round(x)
87
88     @staticmethod
89     def backward(ctx, g):
90         # pretend the derivative of round() is 1 everywhere
91         return g
92
93
94 def q15_round(x):
95     x_clp = torch.clamp(x, -1.0, 1.0 - 1/32768)
96     return _RoundSTE.apply(x_clp * Q15_SCALE) / Q15_SCALE # (x_clp * Q15_SCALE).round() / Q15_SCALE
97
98 # — 4. Quantised layers —————
99 class Q15Linear(nn.Module):
100     def __init__(self, in_f, out_f, bias=True):
101         super().__init__()
102         self.w = nn.Parameter(torch.empty(out_f, in_f))
103         nn.init.trunc_normal_(self.w, std=0.1)
104         self.b = nn.Parameter(torch.zeros(out_f)) if bias else None
105     def forward(self, x):
106         w_q = q15_round(self.w); x_q = q15_round(x)
107         y = F.linear(x_q, w_q, self.b)

```

```

108     return q15_round(y)
109
110 class Q15GELU(nn.Module):
111     def forward(self, x): # gelu in fp32 → clamp → q → return
112         return q15_round(F.gelu(x))
113
114
115
116 # — Custom Self-Attention Layer (based on compute_expected) —————
117 class SelfAttention(nn.Module):
118     def __init__(self, dim=32, n_tokens=16, n_heads=1):
119         super().__init__()
120         self.dim = dim
121         self.n_tokens = n_tokens
122         self.n_heads = n_heads
123         self.att_q = Q15Linear(dim, dim, bias=False)
124         self.att_k = Q15Linear(dim, dim, bias=False)
125         self.att_v = Q15Linear(dim, dim, bias=False)
126         self.proj = Q15Linear(dim, dim, bias=False)
127
128     def compute_attention(self, x_np, wq_np, wk_np, wv_np, wo_np):
129
130         # Step 1: QKV generation
131         def matmul_expected(a, b):
132             expected_out = np.zeros((L * N, E), dtype=float)
133             for k in range(E):
134                 for i in range(L * N):
135                     raw_a = int(a[i, k])
136                     a_val = q1_15_to_real(raw_a)
137                     for j in range(E):
138                         raw_b = int(b[k, j])
139                         b_val = q1_15_to_real(raw_b)
140                         expected_out[i, j] += a_val * b_val
141             expected_out = np.clip(expected_out, -2.0, 1.999999999)
142             expected_q30 = np.vectorize(real_to_q1_30)(expected_out)
143             expected_q15 = []
144             for q30_val in expected_q30.flatten():
145                 sign_bit = (q30_val >> 31) & 1
146                 int_bit = (q30_val >> 30) & 1
147                 if sign_bit == int_bit:
148                     q15_val = (sign_bit << 15) | ((q30_val >> 15) & 0x7FFF)
149                 else:
150                     q15_val = 0x8000 if sign_bit else 0x7FFF
151                 if q15_val & 0x8000:
152                     q15_val = q15_val - 0x10000
153                 expected_q15.append(q15_val)
154             return np.array(expected_q15, dtype=np.int16).reshape(L * N, E)
155
156         q = matmul_expected(x_np, wq_np).reshape(L, N, E)
157         k = matmul_expected(x_np, wk_np).reshape(L, N, E)
158         v = matmul_expected(x_np, wv_np).reshape(L, N, E)
159
160         # Step 2: Attention scores
161         q_np = q.reshape(L * N, E)
162         k_np = k.reshape(L, N, E)
163         matmul_result = np.zeros((L * N, L), dtype=float)
164         for e in range(E):
165             for i in range(L * N):
166                 for j in range(L * N):
167                     q_val = q1_15_to_real(q_np[i, e])
168                     k_val = q1_15_to_real(k_np[j, 0, e]) # N=1
169                     matmul_result[i, j] += q_val * k_val
170             matmul_result = np.clip(matmul_result, -2.0, 1.999999999)
171         matmul_q30 = np.vectorize(real_to_q1_30)(matmul_result)
172         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.:
173
174         a_q15 = []
175         for q30_val in a_q30:
176             sign_bit = (q30_val >> 31) & 1
177             int_bit = (q30_val >> 30) & 1
178             if sign_bit == int_bit:
179                 q15_val = (sign_bit << 15) | ((q30_val >> 15) & 0x7FFF)
180             else:
181                 q15_val = 0x8000 if sign_bit else 0x7FFF
182             if q15_val & 0x8000:
183                 q15_val = q15_val - 0x10000
184             a_q15.append(q15_val)

```

```

185 a = np.array(a_q15, dtype=np.int16).reshape(L, N, L)
186
187 # Step 3: Softmax approximation
188 a_float = np.vectorize(q1_15_to_real)(a)
189 a_softmax = np.zeros((L, N, L), dtype=float)
190 for n in range(N):
191     for i in range(L):
192         relu_row = np.maximum(a_float[i, n, :], 0)
193         row_sum = np.sum(relu_row)
194         if row_sum != 0:
195             a_softmax[i, n, :] = relu_row / row_sum
196         else:
197             a_softmax[i, n, :] = 0
198 a_softmax = np.clip(a_softmax, -1.0, 0.999969482421875)
199 a_softmax_q15 = np.vectorize(real_to_q1_15)(a_softmax).reshape(L * N, L)
200
201 # Step 4: Precision assignment
202 a_sum = np.sum(a_softmax_q15, axis=0)
203 token_precision = []
204 for s in a_sum:
205     if s < 16384:
206         code = 0 # int4
207     elif s < 32768:
208         code = 1 # int8
209     else:
210         code = 2 # fp16
211     token_precision.append(code)
212
213 # Step 5: A*V multiplication
214 av_out = np.zeros((L, E), dtype=float)
215 for k_idx in range(L):
216     prec = token_precision[k_idx]
217     for i in range(L):
218         raw_a = int(a_softmax_q15[i, k_idx])
219         if prec == 0:
220             a_val = nibble_to_real_q1_3(raw_a >> 12)
221         elif prec == 1:
222             a_val = float(np.array((raw_a & 0xFF00) >> 8).astype(np.int8)) / (2**7)
223         else:
224             a_val = q1_15_to_real(raw_a)
225     for j in range(E):
226         raw_v = int(v[k_idx, 0, j]) # N=1
227         if prec == 0:
228             v_val = nibble_to_real_q1_3(raw_v >> 12)
229         elif prec == 1:
230             v_val = float(np.array((raw_v & 0xFF00) >> 8).astype(np.int8)) / (2**7)
231         else:
232             v_val = q1_15_to_real(raw_v)
233         av_out[i, j] += a_val * v_val
234         av_out[i, j] = np.clip(av_out[i, j], -2.0, 1.999999999)
235 av_out = np.clip(av_out, -1.0, 0.999969482421875)
236 av_q15 = np.vectorize(real_to_q1_15)(av_out)
237
238 # Step 6: W_0 multiplication
239 expected_out = np.zeros((L, E), dtype=float)
240 for k_idx in range(E):
241     for i in range(L):
242         raw_av = int(av_q15[i, k_idx])
243         av_val = q1_15_to_real(raw_av)
244         for j in range(E):
245             raw_wo = int(wo_np[k_idx, j])
246             wo_val = q1_15_to_real(raw_wo)
247             expected_out[i, j] += av_val * wo_val
248         expected_out[i, j] = np.clip(expected_out[i, j], -2.0, 1.999999999)
249 out_q30 = np.vectorize(real_to_q1_30)(expected_out)
250
251 out_q15 = []
252 for q30_val in out_q30.flatten():
253     sign_bit = (q30_val >> 31) & 1
254     int_bit = (q30_val >> 30) & 1
255     if sign_bit == int_bit:
256         q15_val = (sign_bit << 15) | ((q30_val >> 15) & 0x7FFF)
257     else:
258         q15_val = 0x8000 if sign_bit else 0x7FFF
259     if q15_val & 0x8000:
260         q15_val = q15_val - 0x10000
261     out_q15.append(q15_val)

```



```

262     out_q15 = np.array(out_q15, dtype=np.int16).reshape(L, E)
263
264
265     return out_q15
266
267 def forward(self, x, quantize=False, save_for_verilog=False, batch_idx=0):
268     if quantize:
269         B = x.size(0)
270         out = torch.zeros(B, L, E, device=DEVICE)
271         x_np = (x * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
272         wq_np = (self.att_q.w.detach() * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
273         wk_np = (self.att_k.w.detach() * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
274         wv_np = (self.att_v.w.detach() * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
275         wo_np = (self.proj.w.detach() * Q15_SCALE).round().clamp(-32768, 32767).to(torch.int16).cpu().numpy()
276
277         if save_for_verilog:
278             os.makedirs("verilog_inputs", exist_ok=True)
279             np.save(f"verilog_inputs/wq_np.npy", wq_np)
280             np.save(f"verilog_inputs/wk_np.npy", wk_np)
281             np.save(f"verilog_inputs/wv_np.npy", wv_np)
282             np.save(f"verilog_inputs/wo_np.npy", wo_np)
283
284         for b in range(min(B, 10)):
285
286             np.save(f"verilog_inputs/x_ln_np_{b}.npy", x_np[b])
287             # print("After:")
288             # print(wq_np.min(), wq_np.max())
289             # print(wo_np.min(), wo_np.max())
290             # print(wv_np.min(), wv_np.max())
291             out_np = self.compute_attention(x_np[b], wq_np, wk_np, wv_np, wo_np)
292             out[b] = torch.tensor(out_np, dtype=torch.float32, device=DEVICE) / Q15_SCALE
293         return out
294     else:
295         q = self.att_q(x)
296         k = self.att_k(x)
297         v = self.att_v(x)
298         att = ((q @ k.transpose(-2, -1)) / self.dim ** 0.5)
299         att = F.relu(att) # ① clip negatives
300         att_sum = (att.sum(-1, keepdim=True) + 1e-5) # ② avoid /0 (Q1.15: eps=1/32768≈0)
301         att = (att / att_sum)
302         #att = F.softmax(att, dim=-1)
303         att_out = (att @ v)
304         return (self.proj(att_out))
305
306 # — Dataset: digits 0 and 1 only —————
307 transform = transforms.Compose([
308     transforms.ToTensor(),
309     transforms.Normalize((0.5,), (0.5,))
310 ])
311
312 mnist = datasets.MNIST(root='.', download=True, train=True, transform=transforms.ToTensor())
313 idx = [i for i, t in enumerate(mnist.targets) if t in (0, 1)]
314 train_set = Subset(mnist, idx)
315
316 mnist_test = datasets.MNIST(root='.', download=True, train=False, transform=transforms.ToTensor())
317 idx_t = [i for i, t in enumerate(mnist_test.targets) if t in (0, 1)]
318 test_set = Subset(mnist_test, idx_t)
319
320 train_loader = DataLoader(train_set, batch_size=BS, shuffle=True, drop_last=True)
321 test_loader = DataLoader(test_set, batch_size=BS, shuffle=False)
322
323 # — Modified OneBlockViT with Custom Self-Attention —————
324 class OneBlockViT(nn.Module):
325     def __init__(self, dim=32, n_patch=7):
326         super().__init__()
327         patch_sz = n_patch * n_patch
328         n_tokens = (28 // n_patch) ** 2 # 16 patches
329         self.n_tokens = n_tokens
330         self.dim = dim
331         self.patch = nn.Linear(patch_sz, dim, bias=False)
332         self.pos_emb = nn.Parameter(torch.empty(1, n_tokens, dim))
333         nn.init.trunc_normal_(self.pos_emb, std=0.02)
334         self.norm1 = nn.LayerNorm(dim)
335         self.norm2 = nn.LayerNorm(dim)
336         self.attention = SelfAttention(dim=dim, n_tokens=n_tokens, n_heads=1)
337         self.mlp = nn.Sequential(
338             nn.Linear(dim, 4*dim), nn.ReLU(), nn.Linear(4*dim, dim))

```

```

339     self.head = nn.Linear(dim, 1, bias=True)
340     self.save_for_verilog = False
341
342     def forward(self, x, quantize=False, save_images=None, save_labels=None):
343         B = x.size(0)
344         x = x.reshape(B, 1, 4, 7, 4, 7).permute(0, 2, 4, 1, 3, 5).reshape(B, -1, 49)
345         x = self.patch(x)
346
347         if quantize:
348             #print(self.patch.weight.min(), self.patch.weight.max())
349             # for b in range(min(B, 10)):
350             #     np.save(f"verilog_inputs/patch_out_software_{b}.npy", x[b].cpu().numpy())
351             # print(x.min(), x.max())
352             # x = q15_round(x + q15_round(self.pos_emb))
353             # x_ln = q15_round(self.norm1(x))
354             x = x + self.pos_emb
355             #print(x.min(), x.max())
356
357             x_ln = self.norm1(x)
358             att_out = self.attention(x_ln, quantize=True, save_for_verilog=self.save_for_verilog, batch_idx=0)
359
360             x = x + att_out
361             #print(x)
362             y = self.norm2(x)
363             x = x + self.mlp(y)
364             x = x.mean(1)
365             out = self.head(x).squeeze(1)
366             # x = q15_round(x + att_out)
367             # y = q15_round(self.norm2(x))
368             # x = q15_round(x + q15_round(self.mlp(y)))
369             # x = x.mean(1)
370             # out = q15_round(self.head(x)).squeeze(1)
371         else:
372             x = x + self.pos_emb
373             x_ln = self.norm1(x)
374             att_out = self.attention(x_ln, quantize=False)
375             x = x + att_out
376             y = self.norm2(x)
377             x = x + self.mlp(y)
378             x = x.mean(1)
379             out = self.head(x).squeeze(1)
380         if self.save_for_verilog and save_images is not None and save_labels is not None:
381             os.makedirs("verilog_inputs", exist_ok=True)
382             for b in range(min(B, 10)):
383                 np.save(f"verilog_inputs/image_{b}.npy", save_images[b].cpu().numpy())
384                 #print(save_images[b].min(), save_images[b].max())
385                 np.save(f"verilog_inputs/label_{b}.npy", save_labels[b].cpu().numpy())
386         return out
387
388 # — Training Function —————
389 def train_model(model, model_name):
390     opt = torch.optim.Adam(model.parameters(), lr=LR, weight_decay=1e-5)
391     scheduler = optim.lr_scheduler.StepLR(opt, step_size=5, gamma=0.5)
392     crit = nn.BCEWithLogitsLoss()
393
394     print(f"\nTraining {model_name} (Full Precision)")
395     for epoch in range(1, EPOCHS+1):
396         model.train()
397         running_loss = 0
398         pbar = tqdm(train_loader, desc=f"Epoch {epoch}", leave=False)
399         for img, lbl in pbar:
400             img, lbl = img.to(DEVICE), lbl.float().to(DEVICE)
401             opt.zero_grad()
402             out = model(img, quantize=True)
403             loss = crit(out, lbl)
404             loss.backward()
405             opt.step()
406             running_loss += loss.item()
407             pbar.set_postfix(loss=float(loss))
408         scheduler.step()
409         print(f"[{model_name} Epoch {epoch}] Loss: {running_loss / len(train_loader):.3f}")
410
411 # — Evaluation Function: Generate Inputs for Verilog —————
412 def evaluate_generate_inputs(model, model_name):
413     model.eval()
414     print(f"\nEvaluating {model_name} (Generate Inputs for Verilog)")
415     saved_count = 0

```

```

416     model.save_for_verilog = True
417     with torch.no_grad():
418         for img, lbl in test_loader:
419             img, lbl = img.to(DEVICE), lbl.float().to(DEVICE)
420             print("label", lbl)
421             model(img, quantize=True, save_images=img, save_labels=lbl)
422             break
423     model.save_for_verilog = False
424     print(f"Saved inputs for {saved_count} examples in 'verilog_inputs'")
425
426 # — Main Execution —————
427 model = OneBlockViT().to(DEVICE)
428 train_model(model, "ViT")
429 evaluate_generate_inputs(model, "ViT")
430 # After hardware simulation, run evaluate_with_hardware(model, "ViT")
431
432 # — Export Q1.15 weights —————
433 int16_state = {k: torch.round(v * Q15_SCALE).to(torch.int16).cpu()
434                for k, v in model.state_dict().items()}
435 torch.save(int16_state, "vit_q15_mixed_int16.pt")
436 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., 1., 1., 0., 1., 0., 1., 1., 1.,
              1., 0., 0., 1., 1., 1., 0., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1.,
              1., 1., 0., 0., 1., 1., 1., 1., 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

```

```

1 a = np.load("verilog_inputs/wq_np.npy")
2 print(a.min(), a.max())
3

```



-10352 12130

```

1 !rm -rf verilog_inputs*
2 !rm -rf verilog_outputs*

```

```

1 # Evaluate with quantization
2 model.eval()
3 model_name = "ViT"
4 print(f"\nEvaluating {model_name} (Quantized Full Precision)")
5 correct_full = 0
6 total = 0
7
8 with torch.no_grad():
9     for img, lbl in test_loader:
10         img, lbl = img.to(DEVICE), lbl.float().to(DEVICE)
11         pred = torch.sign(model(img, quantize=True))
12         print(pred)
13         correct_full += (pred == (lbl*2-1)).sum().item()
14         total += lbl.size(0)
15 acc_full = 100 * correct_full / total
16 print(f"[{model_name}] Quantized Mixed Precision Test Accuracy: {acc_full:.2f}%")
17
18 print(f"\nEvaluating {model_name} (Quantized Mixed Precision)")
19 correct_mixed = 0
20 total = 0
21 with torch.no_grad():

```

```

22     for img, lbl in test_loader:
23         img, lbl = img.to(DEVICE), lbl.float().to(DEVICE)
24         pred = torch.sign(model(img, quantize=False))
25         correct_mixed += (pred == (lbl*2-1)).sum().item()
26         total += lbl.size(0)
27 acc_mixed = 100 * correct_mixed / total
28 print(f"[{model_name}] Quantized Full Precision Test Accuracy: {acc_mixed:.2f}%")
29

```

```

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

```

```

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

```

```

1 !zip -r verilog_inputs.zip verilog_inputs
2

```

```

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%)

```

```

1 !unzip -q verilog_outputs.zip
2

```

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

```

1 import torch
2 import numpy as np
3
4
5 # — Evaluation Function: Load Hardware Outputs and Predict —
6 def evaluate_with_hardware(model, model_name):
7     model.eval()
8     print(f"\nEvaluating {model_name} (Quantized Mixed Precision with Hardware)")
9     correct_hardware = 0
10    total = 0
11    saved_count = 0
12    print("\nComparing Hardware Predictions with Labels:")
13    plt.figure(figsize=(15, 3))
14    with torch.no_grad():
15        for img, lbl in test_loader:
16            img, lbl = img.to(DEVICE), lbl.float().to(DEVICE)
17            B = img.size(0)
18            for b in range(min(B, 10 - saved_count)):
19                out_mem = np.load(f"verilog_outputs/self_attention_out_{b}.npy")
20                image = np.load(f"verilog_inputs/image_{b}.npy")
21
22                label = np.load(f"verilog_inputs/label_{b}.npy")
23
24                image_t = torch.tensor(image, dtype=torch.float32, device=DEVICE) # (1,28,28)
25                image_t = image_t.unsqueeze(0)
26
27                patches = image_t.reshape(1, 1, 4, 7, 4, 7).permute(0,2,4,1,3,5).reshape(1, -1, 49)
28
29                x=model.patch(patches)
30                #-----debug-----
31                # x_hardware = model.patch(patches)
32
33                # # Load software patch output and compare
34                # x_software = np.load(f"verilog_outputs/patch_out_software_{b}.npy")
35                # x_hardware_np = x_hardware[0].cpu().numpy()
36                # diff = np.max(np.abs(x_software - x_hardware_np))
37                # print(f"Patch embedding max difference for example {b}: {diff}")
38
39                # print(x.min(), x.max())
40                #-----debug-----
41
42
43                x = ( x + model.pos_emb) # add the skip path
44                #print(x.min(), x.max())
45
46                att_out = torch.tensor(out_mem.astype(np.int16), dtype=torch.float32, device=DEVICE) / Q15_SCALE
47                att_out = att_out.unsqueeze(0)
48

```

```

49
50     x_ln = (model.norm1(x))
51     x = (x + (att_out))
52     #print(x.min(), x.max())
53
54     y = (model.norm2(x))
55     x = (x + (model.mlp(y)))
56     x = x.mean(1)
57     #print(model.head.weight)
58     out = (model.head(x)).squeeze(1)
59
60     pred = torch.sign(out).item()
61     expected = (2 * label - 1)
62
63
64     # image_t = torch.tensor(image, dtype=torch.float32, device=DEVICE).unsqueeze(0)
65     # patches = image_t.reshape(1, 1, 4, 7, 4, 7).permute(0, 2, 4, 1, 3, 5).reshape(1, -1, 49)
66     # x = q15_round(model.patch(patches) + model.pos_emb)
67     # x = q15_round(x + q15_round(att_out))
68     # y = q15_round(model.norm2(x))
69     # x = q15_round(x + q15_round(model.mlp(y)))
70     # x = x.mean(1)
71     # out = q15_round(model.head(x)).squeeze(1)
72     # pred = torch.sign(out).item()
73     # expected = (2 * label - 1)
74     print(f"Example {b}: Predicted = {0 if pred == -1 else 1}, Expected = {label}, Match = {pred == expected}")
75     correct_hardware += (pred == expected)
76     total += 1
77     plt.subplot(1, 10, b+1)
78     plt.imshow(image.squeeze(), cmap='gray')
79     plt.title(f"Pred: {0 if pred == -1 else 1}\nTrue: {label}")
80     plt.axis('off')
81     saved_count += 1
82     if saved_count >= 10:
83         break
84 plt.tight_layout()
85 plt.show()
86 acc_hardware = 100 * correct_hardware / total if total > 0 else 0
87 print(f"[{model_name}] Quantized Mixed Precision Hardware Test Accuracy: {acc_hardware:.2f}%")
88 return acc_hardware
89
90
91
92 # Load the model and weights (assuming vit_q15_mixed_int16.pt is available)
93 # model = OneBlockViT().to(DEVICE)
94 # state = torch.load("vit_q15_mixed_int16.pt") #no influence
95 # state = {k: v.to(torch.float32) / Q15_SCALE for k, v in state.items()}
96 # model.load_state_dict(state)
97
98 # Run the hardware evaluation
99 evaluate_with_hardware(model, "ViT")

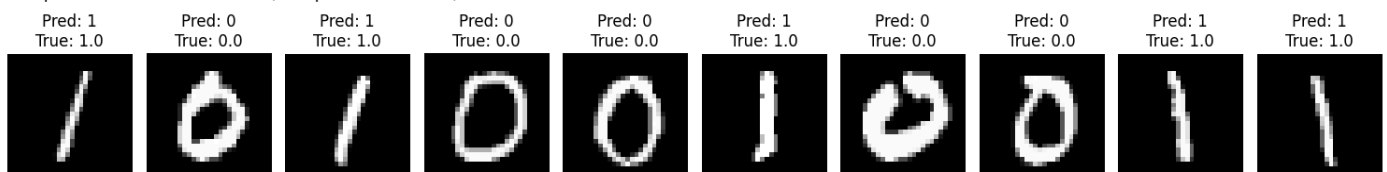
```



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)

