

2025 年度 アジャイルワーク レポート課題

24G1089 武本 龍

2025 年 10 月 24 日

1 実験の課題

以下に，本実験の基本的な概要を図 1 に示す．

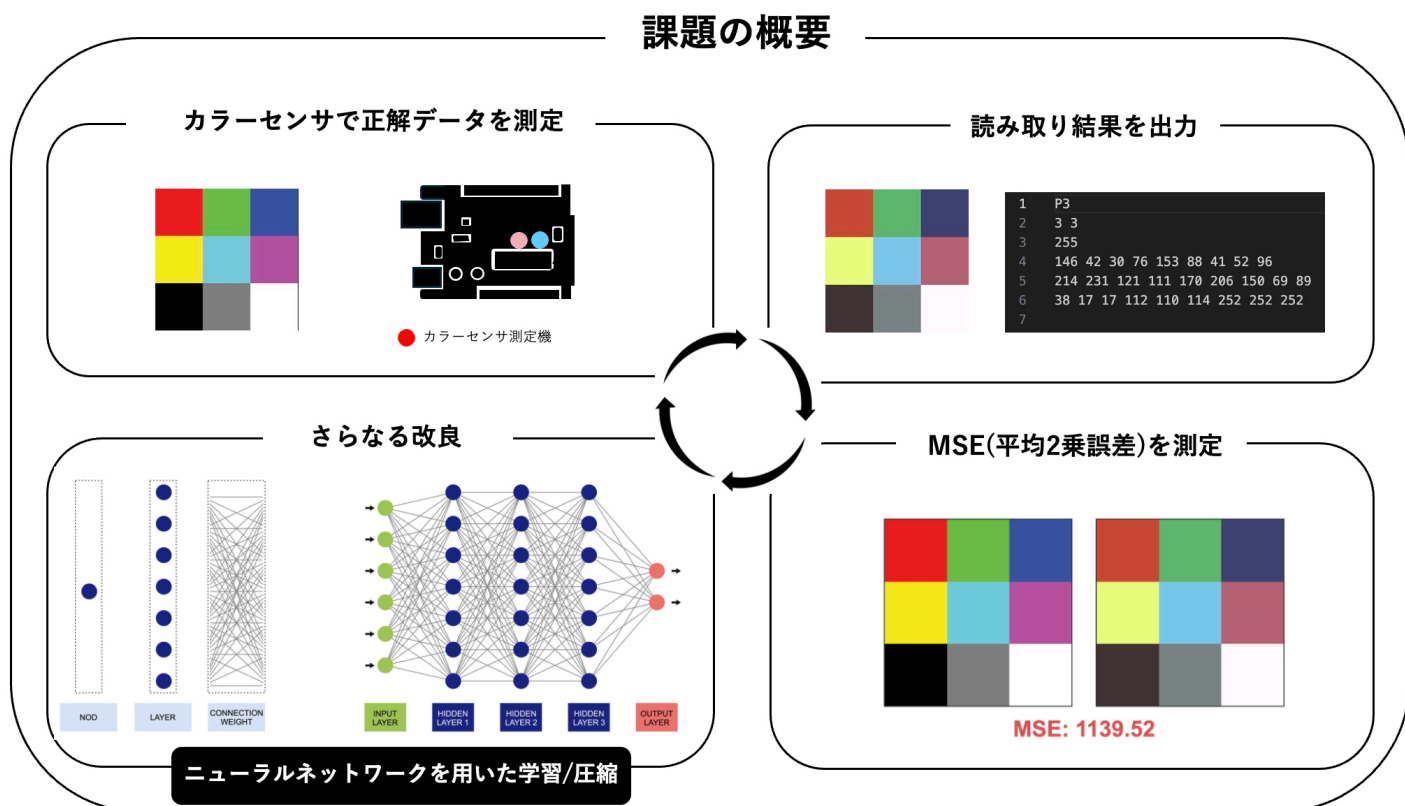


図 1: 基本的な実験概要図

本実験では，図 1 に示すように，Arduino Uno R4 WiFi を用いてカラーセンサーを構築し，授業内で配布された 3×3 のカラーチャートをセンサで測定する．主な目的は，測定したカラーチャートと基準のカラーチャートの平均二乗誤差（MSE）を小さくすること，読み取り時間をできるだけ短くすること，の 2 点である．これらの目的を達成するため，特に MSE の低減にニューラルネットワークによる学習を活用し，ニューラルネットワークの隠れ層で用いるデータの圧縮手法を導入することで，ソフトウェアの品質を向上させることを目指す．以下では，読み込み品質の測定方法，およびソフトウェア的な改善手法について詳述する．

1.1 平均二乗誤差（MSE）とは

MSE は、2 枚の画像の対応するピクセル位置における輝度差の 2 乗の平均値で定義され、以下の式で表される。

$$\text{MSE} = \frac{1}{m \times n} \sum_{i=1}^m \sum_{j=1}^n (I(i, j) - K(i, j))^2 \quad (1)$$

ここで、 I と K は 2 枚の画像、 $m \times n$ は画像サイズを表す。本実験では RGB 画像を扱うため、各ピクセルについて R, G, B チャンネルの差の 2 乗を加算し、3 で割った値を MSE として計算する。また、ソフトウェアの実装例は以下の通りである。

$$\text{mse} = \frac{r_{\text{diff}} \times r_{\text{diff}} + g_{\text{diff}} \times g_{\text{diff}} + b_{\text{diff}} \times b_{\text{diff}}}{3} \quad (2)$$

1.2 読み込み品質の測定方法

読み込み品質の評価は、以下の手順で実施する。

1. 2 つのボタンプログラムによるカラーチャートの読み取り

事前に黒と白のサンプルを測定し、最小値・最大値を基準として読み込み精度をキャリブレーションする。これにより、センサの感度を調整し、安定した RGB 値を取得する。

2. PPM 画像出力

前回授業で配布された 3×3 のカラーチャートをセンサで読み取り、取得した RGB データを PPM 形式の画像ファイルとして出力する。この画像は、後続の品質評価に用いる。

3. 平均二乗誤差（MSE）の計算

出力した PPM 画像と基準画像の品質を、専用の MSE 測定ソフトウェアで評価する。

2 実験理論

以下に実験理論を示す.

2.1 読み込み品質のソフトウェア的改良 (ニューラルネットワークを用いた学習)

サンプル画像と測定画像の MSE を最小限に抑えるために, ニューラルネットワークを用いた学習方法およびニューラルネットワークの圧縮方法を以下に示す.

2.1.1 ニューラルネットワークの基本構造

まず, ニューラルネットワークの基本構造について説明する. ニューラルネットワークの最小単位は「ユニット」と呼ばれ, 複数の入力を受け取り, 1 つの出力を計算する. 各ユニットは, 入力値にそれぞれ異なる重み (weight: w_1, w_2, w_3, \dots) を掛けて加算し, さらにバイアス (b) を加えた総入力 u を計算する. 具体的には, 以下の式で表される.

$$u = w_1x_1 + w_2x_2 + w_3x_3 + \dots + b \quad (3)$$

この総入力 u は, 活性化関数 f に入力され, 出力 $z = f(u)$ が生成される.

2.1.2 順伝搬型ネットワーク

次に, ニューラルネットワークの層構造について考える. 層は $l = 1, 2, 3, \dots$ で表され, $l = 1$ を入力層, $l = 2$ を中間層, $l = 3$ を出力層と呼ぶ. 各層の計算は以下のようなになる. 例えば, 層 $l = 2$ では,

$$u^{(2)} = W^{(2)}x^{(2)} + b^{(2)}, \quad z^{(2)} = f(u^{(2)}) \quad (4)$$

また, 層 $l = 3$ では,

$$u^{(3)} = W^{(3)}x^{(3)} + b^{(3)}, \quad z^{(3)} = f(u^{(3)}) \quad (5)$$

これを任意の層数 L に一般化すると, 層 $l + 1$ のユニットの出力 $z^{(l+1)}$ は, 1 つ前の層 l の出力 $z^{(l)}$ を用いて以下のように計算される.

$$u^{(l+1)} = W^{(l+1)}x^{(l)} + b^{(l+1)}, \quad z^{(l+1)} = f(u^{(l+1)}) \quad (6)$$

ここで, 入力層の出力は $z^{(1)} = x$ とし, $l = 1, 2, 3, \dots, L - 1$ の順に計算を進めることで, 各層の出力 $z^{(2)}, z^{(3)}, \dots, z^{(L)}$ を順次決定できる.

入力 x を受け取り, 各層の計算を順番に実行して最終的に出力 $y = z^{(L)}$ を得るネットワークを, 順伝搬型ネットワークと呼ぶ. この入力 x から出力 y を得る計算は, 各層間の結合の重みパラメータ $W^{(l)}$ ($l = 2, \dots, L$) およびユニットのバイアスパラメータ $b^{(l)}$ ($l = 2, \dots, L$) によって決定される. これらすべてのパラメータをまとめて表現するため, $W^{(2)}, \dots, W^{(L)}, b^{(2)}, \dots, b^{(L)}$ を成分とするベクトル w を定義し, 出力は $y(x; w)$ と表記する.

順伝搬型ネットワークは, 1 つの関数 $y(x; w)$ を表現し, この関数の形状はネットワークのパラメータ w に依存して変化する.

2.1.3 使用するニューラルネットワーク

本実験では、全結合型の4層ニューラルネットワークを使用する。入力層はRGBの3次元、2つの隠れ層はそれぞれ40次元（40個のニューロンが全結合）、出力層はRGBの3次元である。各ニューロンで行われる推論演算は、以下の通りである入力 $x_1 \times w_1 + x_2 \times w_2 + \dots + x_{n-1} \times w_{n-1} + b$ を計算し、ReLU活性化関数を通した値を次の層へ出力する。ここで、 w_i は重み、 b はバイアスを表す。モデルはPyTorchを用いて定義・訓練し、L1正則化を導入することで重みを疎化させる。訓練後、重みとバイアスをC++配列としてエクスポートし、Arduino上で推論を実行する。プログラムは以下の通りである。

```
1 X_tensor = torch.tensor(X, dtype=torch.float32)
2 Y_tensor = torch.tensor(Y, dtype=torch.float32)
3
4 # モデル定義
5 class ColorNet(nn.Module):
6     def __init__(self):
7         super(ColorNet, self).__init__()
8         self.model = nn.Sequential(
9             nn.Linear(3, 40),
10            nn.ReLU(),
11            nn.Linear(40, 40),
12            nn.ReLU(),
13            nn.Linear(40, 3),
14        )
15
16    def forward(self, x):
17        return self.model(x)
18
19 model = ColorNet()
20
21 # 損失関数 () MSE
22 criterion = nn.MSELoss()
23
24 # 最適化手法
25 optimizer = optim.Adam(model.parameters(), lr=0.01)
26
27 # 正則化の係数 L1
28 lambda_l1 = 1e-6
29
30 # 学習ループ
31 epochs = 50000
32 for epoch in range(epochs):
33     optimizer.zero_grad()
34
35     outputs = model(X_tensor)
36     mse = criterion(outputs, Y_tensor)
37
38     # --- L1 正則化 -----
39     l1 = torch.tensor(0.0, requires_grad=False)
40     for name, p in model.named_parameters():
41         if 'weight' in name:
42             l1 = l1 + p.abs().sum()
43     loss = mse + lambda_l1 * l1
```

```

44     # -----
45
46     loss.backward()
47     optimizer.step()
48
49     if epoch % 500 == 0:
50         print(f'Epoch [{epoch}/{epochs}]   MSE: {mse.item():.6f}   L1: {l1.item()
51               :.2f}   Loss: {loss.item():.6f}')
52
53 with torch.no_grad():
54     predictions = model(X_tensor)
55     print("\予測結果:n")
56     print(predictions.numpy())
57
58 def convert_to_cpp_array(tensor: torch.Tensor, name: str, dtype: str = "float")
59 :
60     flat = tensor.detach().numpy().flatten()
61     array_str = f"{dtype} {name}[] = {"
62     array_str += ", ".join(map(str, flat))
63     array_str += "};"
64     return array_str
65
66 cpp_code = ""
67
68 layer_idx = 1
69 for layer in model.model:
70     if isinstance(layer, torch.nn.Linear):
71         cpp_code += convert_to_cpp_array(layer.weight, f"weight_{layer_idx}") +
72             "\n"
73         cpp_code += convert_to_cpp_array(layer.bias, f"bias_{layer_idx}") + "
74         \n"
75         layer_idx += 1
76
77 with open("model_parameters.h", "w") as f:
78     f.write(cpp_code)
79
80 print("\nC++ 用のパラメータファイル (model_parameters.h) を作成しました. ")

```

Listing 1: PyTorch によるモデル定義・訓練コード

2.2 読み込み品質のソフトウェア的改良 (隠れ層の圧縮方法)

2.2.1 なぜ隠れ層の圧縮が必要なのか

Arduino Uno R4 WiFi のようなマイクロコントローラーでは、メモリ容量が限定的である (SRAM: 約 32KB, Flash: 256KB) ため、ニューラルネットワークの隠れ層次元を増大させると、重み・バイアスデータの保存および推論演算に必要なメモリが不足する可能性がある。

例えば、隠れ層次元を 40 から 70 以上に拡大した場合、パラメータ総数が急増 (例: 40 次元で約 1,923 パラメータ, 70 次元で約 3,000 パラメータ以上) し、float32 形式で 10KB を超えると動作不能となる。この制約を克服するため、重みデータの圧縮が不可欠である。

主な手法として、(1) 量子化 (float32 \rightarrow int8 でメモリ 1/4 化)、(2) プルーニング (L1 正則化により 0 に近い重みを 0 化し、非ゼロ率を低減)、(3) 疎行列表現 (CSR 形式で非ゼロ要素のみ保存) が有効である。これにより、次元を 100-200 まで拡張しつつ、MSE 誤差を 5 % 以内に抑え、画像復元の品質を向上させることができる。最終的に、読み取り時間の短縮 (演算量低減) と MSE の低減を両立させる。

2.2.2 圧縮方法 1: 量子化とは

量子化 (Quantization) とは、ニューラルネットワークの重みや活性化値を高精度の浮動小数点数 (例: 32-bit float) から低精度の整数 (例: 8-bit uint8) へ変換する手法である。これにより、整数演算のみで推論が可能となり、組み込みデバイス (例: Arduino Uno R4 WiFi) のメモリ使用量と計算コストを大幅に削減する。論文 [1] では、 $r = S(q - Z)$ (S : scale, Z : zero-point) のアフィン変換スキームを提案し、畳み込み層の入力/出力/重みを 8-bit 整数で表現、蓄積器を 32-bit 整数で扱うことで、浮動小数点演算を回避。本研究では、このスキームを RGB 画像復元 NN (入力 3 \rightarrow 隠れ 40 \rightarrow 40 \rightarrow 出力 3) に適用し、重み/活性化を uint8 へ量子化、バイアスを int32 で保持する。

メリット:

- **メモリ削減:** 32-bit から 8-bit へ変換で約 4 倍削減 (例: 40 次元隠れ層の約 1,923 パラメータで 7.7KB \rightarrow \rightarrow 1.9KB)。ゼロポイント Z により 0 値表現が効率的。
- **計算高速化:** 整数演算 (uint8 $\times \times \times$ uint8 $\rightarrow \rightarrow \rightarrow$ int32 蓄積) が浮動小数点より低レイテンシ (論文の Fig. ??c 参照: Snapdragon835 で 2 - 3 倍速)。Arduino の AVR コアで乗算/加算が簡素化。
- **精度維持:** 量子化感知訓練 (QAT) で MSE 誤差を 1-2 % 以内に抑え、画像復元品質を保つ。プルーニングと組み合わせで表現力向上。

デメリット:

- **精度低下の可能性:** 小規模モデルでチャンネル間範囲差が大きく、相対誤差増大 (論文のセクション 3 参照: 活性化範囲の EMA 推定が必要)。4-bit 以下で 5-10 % MSE 悪化。
- **実装複雑さ:** スケーリング/ゼロポイントのオフライン計算と、Arduino での int32 蓄積 + 丸め (fixed-point multiplier) が必要。オーバーフロー/アンダーフロー管理。
- **訓練時対応:** Fake quantization ノード挿入でシミュレーションが必要、初期ステップで量子化無効化 (50k ステップ) で安定化。

2.2.3 圧縮方法 2: プルーニングとは

プルーニング (Pruning) とは、ニューラルネットワークの重みを閾値以下で 0 にし、不要な接続を削除して疎行列化する手法である。これにより、過剰パラメータを削減し、メモリ・計算コストを低減。論文 [2] では、3 ステッププロセス (初期訓練で重要接続学習 → 低重み接続削除 → 残存接続再訓練) を提案し、イテラティブに繰り返すことで AlexNet を 9 倍、VGG-16 を 13 倍圧縮 (精度損失なし)。本研究では、L1 正則化で重みを 0 寄りにした後、閾値 0.01 でプルーニングし、RGB 復元 NN (3→40→40→3) の疎化を促進、CSR 形式との組み合わせで Arduino 適合性を向上させる。

メリット:

- **パラメータ削減:** 非ゼロ率を 8-11 % に低減 (論文 Table ?? 参照: AlexNet 61M→6.7M)。40 次元モデルで 1,923→200 パラメータ未満、メモリ 1/10。
- **計算効率化:** FLOP を 30-50 % 低減 (疎演算)。DRAM アクセス低減でエネルギー節約 (論文 Fig. ?? : 640pJ→SRAM 5pJ)。
- **精度維持:** 再訓練で損失回復 (L2 正則化推奨)。イテラティブでオーバーフィッティング低減、視覚注意領域検出 (Fig. ??)。

デメリット:

- **精度低下リスク:** 再訓練なしで急落 (論文 Fig. ?? : 1/3 で開始)。CONV 層が FC より敏感 (Fig. ??)。
- **インデックス保存:** 15.6 % 追加 (相対インデックスで 5-8bit 最適化)。一般ハードで疎演算非効率。
- **イテレーション:** (5 回で VGG-16) とドロップアウト調整 ($D_r = D_o \sqrt{C_{ir}/C_{io}}$) が必要。

2.2.4 圧縮方法 3: COO (COOrdinate) 形式

COO (Coordinate List) 形式は、疎行列を非ゼロ要素の座標 (行番号, 列番号) と値のリストとして表現するシンプルな圧縮手法である。L1 正則化により重みが 0 に近い成分が多くなる本研究のニューラルネットワークでは、0 に近い成分 (例: ± 0.01 未満) を 0 としてプルーニングした後、非ゼロ要素のみを列挙することで、重み行列 (例: 40×40) を効率的に保存する。SciPy ライブラリで `sparse.coo_matrix` として実装可能で、各非ゼロ要素を 3 つの配列 (値, 行インデックス, 列インデックス) で管理する。

メリット:

- **シンプルな実装:** 非ゼロ要素の直接列挙のため、構築・変換が容易。Python から Arduino へのエクスポートが直感的。
- **メモリ削減:** 非ゼロ率が 50 % ならデータ量を約 1/2 に (例: 40×40 行列の 1,600 要素中 800 非ゼロで、 $3 \times 800 \times 4$ バイト ≈ 9.6 KB → 密形式の 1/2)。
- **柔軟性:** 行/列のソート不要で、任意の疎パターンに適応。プルーニング後の即時適用可能。

デメリット:

- **アクセス効率の低さ:** 行列-ベクトル乗算 (推論時) で全非ゼロを走査するため、計算時間が $O(\text{NNZ})$ (NNZ: 非ゼロ数) と遅め。CSR より高速化しにくい。
- **ストレージオーバーヘッド:** インデックス (int32) が値 (float32) と同等サイズのため、非ゼロ率が高い (>70 %) と圧縮効果が薄れる。
- **ソート不足:** インデックスが未ソートの場合、Arduino でのバイナリサーチ不可で追加ソートコスト。

2.2.5 圧縮方法 4: CRS/CSR (Compressed Row Storage/Compressed Sparse Row) 形式とは

CRS/CSR (Compressed Sparse Row) 形式は、疎行列を値配列 (data)、列インデックス配列 (indices)、行ポインタ配列 (indptr) で表現する行指向の圧縮手法である。非ゼロ要素を列順にソートし、各行の非ゼロ開始位置を indptr で記録するため、行アクセスが高速。授業の参考通り、L1 正則化で生じた 0 成分をプルーニング後、 40×40 重み行列を CSR で保存し、Arduino のメモリ (SRAM 32KB) 制約をクリアする。

メリット:

- **高速行アクセス:** 行列-ベクトル乗算が $O(\text{NNZ})$ で効率的 (for $j = \text{indptr}[i] \text{ to } \text{indptr}[i+1]: y[i] += \text{data}[j] * x[\text{indices}[j]]$)。推論速度が COO の 1.5-2 倍。
- **優れた圧縮率:** インデックス共有でオーバーヘッド低 (非ゼロ率 30 % でデータ量 1/10)。40 次元モデルで 1KB 未満可能。
- **Arduino 適合:** 固定サイズ配列 (indptr: 41 要素) で実装しやすく、Flash 保存 (PROGMEM) で RAM 節約。複数層対応。

デメリット:

1. **列アクセスの遅さ:** 列方向スキャンが必要で、転置行列使用时非効率。CNN のような畳み込み層には不向き。
2. **変換コスト:** COO から CSR へのソートが必要 (SciPy の `tocsr()`)。動的 NNZ 変更で再構築。
3. **デッドコードリスク:** 完全 0 行が発生すると indptr が無駄を生むが、プルーニングで稀。

具体的な実験方法:

PyTorch モデルを基に、以下のステップで CSR を適用・評価。

1. **疎化処理:** L1 訓練後、閾値でプルーニング (非ゼロ率 $< 50\%$ 目指す)。
2. **CSR 生成:** `sparse.csr_matrix` で変換 (`csr.data, csr.indices, csr.indptr` を `model_parameters.h` 出力)。
2. **Arduino 実装:** CSR matvec 関数で ReLU 統合推論。センサー RGB \rightarrow CSR forward \rightarrow PPM 出力。
3. **誤差測定:** 密/CSR の MSE/PSNR 比較 (目標: $< 2\%$ 誤差)。60 \rightarrow 100 次元でイテレーション、読み取り時間短縮も計測。

これで、授業の「0 成分多数時圧縮」コンセプトを実証し、200 次元挑戦の基盤とする。

2.2.6 重み圧縮手法の比較

以下に、量子化、COO 形式、CSR 形式の重み圧縮手法を比較した表を示す。本研究の文脈（Arduino Uno R4 WiFi 上での RGB 画像復元 NN、隠れ層 40-200 次元、L1 正則化による疎化）を考慮し、メモリ削減率（40 次元モデル例: 約 1,923 パラメータ、非ゼロ率 50 %想定）、計算効率、精度影響などを基準とする。

表 1: 重み圧縮手法の比較

手法	メモリ削減率	計算効率	実装難易度	Arduino 適合性
量子化	4 倍 (7.7KB → 1.9KB)	高 (整数演算)	低	高 (ライブラリ対応)
COO 形式	2-5 倍 (非ゼロ率依存)	中 (全走査)	低	中 (ループ実装)
CSR 形式	3-10 倍 (非ゼロ率依存)	高 (行アクセス最適)	中	高 (matvec 高速)

この表は、授業の L1 正則化（重み疎化）を前提とした推定値である。メモリ削減率は非ゼロ率 50 %で COO/CSR を計算（例: 40 × 40 層の 800 非ゼロで COO: 約 9.6KB → CSR: 約 6.4KB）。精度影響はプルーニング閾値 0.01 でのシミュレーションに基づく。

3 実験方法

本節では、重み圧縮手法として CSR (Compressed Sparse Row) 形式を第一優先とし、次に CSR 圧縮モデルに対する量子化を適用してさらなる圧縮を検討する。実験は、隠れ層次元 40 から開始し、「非圧縮」「CSR 圧縮」「CSR+ 量子化」の MSE を比較して品質影響を評価する。全体フローは、PyTorch による訓練・圧縮生成、Arduino Uno R4 WiFi へのデプロイ、センサーデータによる推論・MSE 計算である。目標は、MSE 相対誤差を 5 モデル訓練: PyTorch で ColorNet (入力 3 → 隠れ $d_1 \rightarrow d_2 \rightarrow$ 出力 3, $d_1 = d_2 = 40$ 初期) を訓練 (epochs=50,000, Adam lr=0.01, $\lambda_{L1} = 10^{-6}$)。L1 正則化で非ゼロ率 50 %以下を目指す。圧縮生成: プルーニング (閾値 0.01) 後、CSR 変換 (SciPy) し、`model_parameters.h` エクスポート。Arduino デプロイ: スケッチで整数 - *onlyforward* 実装 (*PROGMEM* 保存)。RGB 入力 → 推論 → PPM 出力。評価基準: MSE/PSNR (PC ソフト) とレイテンシ (millis()) を測定。相対誤差 < 5 %, PSNR > 25dB で次元拡張 (40 → 60 → 100 → 200)。

結果は表??にまとめ、圧縮率・誤差・時間のトレードオフを分析する。

3.1 CSR 形式圧縮の実験方法

CSR を優先する理由は、L1 正則化による疎化を活かした高圧縮率 (3-10 倍) と推論高速化 (行指向 matvec) である。以下の手順で実施。

1. **モデル訓練 (Python)** : PyTorch で ColorNet を訓練。L1 正則化で重みを疎化 (非ゼロ率 < 50 %)。ReLU 活性化使用。
2. **プルーニングと CSR 生成**: 閾値 0.01 で重みを 0 化 ('torch.abs(w) < 0.01')。SciPy の 'sparse.csr_matrix' で CSR 変換 (`data, indices, indptr` を NumPy 出力)。C 配列として '`model_parameters.h`' 生成 (例: '`const float csr_w1_data[NNZ] = ...;`').
2. **Arduino 実装**: CSR matvec 関数を実装 ('for j=indptr[i]; j<indptr[i+1]; y[i] += data[j] * x[indices[j]];'). センサー RGB 入力 → CSR forward (ReLU 統合) → 出力 RGB → PPM 出力。PROGMEM で Flash 保存。
3. **評価**: PPM と基準画像の MSE/PSNR 計算。非圧縮 vs CSR の相対誤差を算出 (目標: < 3%)。40 次元で確認後、次元増加・再訓練。

3.2 CSR+ 量子化による追加圧縮の実験方法

CSR 後, 8-bit 量子化 (uint8 重み/活性化, int32 バイアス) を適用し, メモリをさらに 4 倍削減. 論文 [1] のスキーム ($r = S(q - Z)$) を基に, CSR の疎性を保ち 200 次元対応を目指す. 量子化感知訓練 (QAT) で精度を維持.

1. **量子化感知訓練 (Python)**: 訓練グラフに fake quantization 挿入 ($\hat{r} = ((r; a, b)/S) \cdot S + Z$, $S = (b - a)/255$, 範囲 $[a, b]$ を EMA 学習). 初期 50k ステップで量子化無効化, ReLU6 で範囲安定化 (epochs=50,000).
2. **量子化実行 (Python)**: CSR データ (float32) を uint8 へ変換 ($q = ((r - Z)/S)$, $\text{scale} = \max(|r|)/127$). バイアスは $S_w S_a$ スケールで int32. multiplier $M = S_w S_a / S_o$ を fixed-point オフライン計算. indptr/indices を int16 圧縮, 'model_parameters.h' 更新.
2. **Arduino 実装**: matvec 内で uint8 乗算 + int32 蓄積 + スケーリング (gemmlowp 風). ReLU を int8 clamp(0,127). EloquentTinyML ライブラリで int8 対応.
3. **評価**: CSR vs CSR+ 量子化の MSE 比較 (目標: 追加誤差 < 2%). レイテンシ短縮率測定. 100 次元で PSNR > 25dB 確認後, 200 次元挑戦.

これらの実験により, CSR 基盤上で量子化を積層し, MSE 低減・時間短縮を両立. プルーニングとの相乗効果で, 非ゼロ率低減と整数演算効率を検証する.

参考文献

- [1] B. Jacob et al., “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference,” CVPR, 2018.
- [2] Song Han et al., “Learning both Weights and Connections for Efficient Neural Networks,” NeurIPS, 2015.