RP2040 Pizza-Erkennungssystem - Dokumentation

1. Projektziel

Entwicklung eines optimierten Bildklassifizierungsmodells für den RP2040-Mikrocontroller, das verschiedene Phasen des Pizzabackens erkennen kann. Das System soll mit minimalen Ressourcen arbeiten und auf einem batteriebetriebenen Gerät mit einer OV2640-Kamera laufen.

2. Hardware-Spezifikationen

- **Mikrocontroller**: RP2040
  - CPU: Dual-Core Arm Cortex M0+ @ 133MHz
  - RAM: 264KB
  - Flash: 2048KB (2MB)
- **Kamera**: OV2640
  - Auflösung: 320x240
  - Framerate: 7 FPS (Batteriebetrieb)
- **Stromversorgung**: CR123A Batterie
  - Kapazität: 1500mAh
  - Aktive Laufzeit: ~8.33h
  - Standby-Laufzeit: ~3000h
- **Speicherbeschränkungen**:
  - Maximale Modellgröße: 180KB
  - Maximaler Laufzeit-RAM: 100KB

3. Datensatz

Der Datensatz besteht aus 57 Bildern in 6 Klassen:

- `basic`: 29 Bilder
- `combined`: 17 Bilder
- `mixed`: 7 Bilder
- `burnt`: 4 Bilder
- `progression`: 0 Bilder (leere Klasse)
- `segment`: 0 Bilder (leere Klasse)

Datensatzprobleme:

- Sehr kleine Datensatzgröße
- Unbalancierte Klassenverteilung
- Zwei Klassen ohne Beispiele

4. Modellarchitektur - MicroPizzaNet

Ein speziell für Mikrocontroller optimiertes CNN mit minimaler Parameteranzahl:

python

Copy

```python
class MicroPizzaNet(nn.Module):
    def __init__(self, num_classes=4, dropout_rate=0.2):
        super(MicroPizzaNet, self).__init__()

        # Erster Block: 3 -> 8 Filter
        self.block1 = nn.Sequential(
            nn.Conv2d(3, 8, kernel_size=3, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(8),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)  # Ausgabe: 8x12x12
        )

        # Zweiter Block: 8 -> 16 Filter mit depthwise separable Faltung
        self.block2 = nn.Sequential(
            # Depthwise Faltung
            nn.Conv2d(8, 8, kernel_size=3, stride=1, padding=1, groups=8, bias=False),
            nn.BatchNorm2d(8),
            nn.ReLU(inplace=True),
            # Pointwise Faltung (1x1) zur Kanalexpansion
            nn.Conv2d(8, 16, kernel_size=1, bias=False),
            nn.BatchNorm2d(16),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)  # Ausgabe: 16x6x6
        )

        # Global Average Pooling
        self.global_pool = nn.AdaptiveAvgPool2d(1)  # Ausgabe: 16x1x1

        # Kompakter Klassifikator
        self.classifier = nn.Sequential(
            nn.Flatten(),  # 16
            nn.Dropout(dropout_rate),
            nn.Linear(16, num_classes)
        )

    def forward(self, x):
        x = self.block1(x)
        x = self.block2(x)
        x = self.global_pool(x)
        x = self.classifier(x)
        return x
```

Modellspezifikationen:

- Eingabegröße: 48x48x3 (RGB-Bild)
- Parameteranzahl: 582
- Modellgröße (Float32): 2.54 KB
- Modellgröße (Int8): 0.63 KB
- Aktivierungsspeicher: 101.46 KB
- Gesamter Laufzeitspeicher: 102.09 KB (38.7% des verfügbaren RAM)

5. Trainingsparameter

- Batch-Größe: 16
- Maximale Epochen: 50
- Lernrate: 0.002 mit OneCycle-Scheduling
- Früher Abbruch: Nach 10 Epochen ohne Verbesserung
- Klassengewichtung für unbalancierte Daten
- Gewichtsverfall: 1e-5
- Optimierer: Adam

6. Aufgetretene Fehler und Lösungen

Problem 1: RAM-Überschreitung

Fehler: Laufzeitspeicher überschreitet RAM-Beschränkung (102.09KB > 100KB)

Mögliche Lösung (nicht implementiert):

- Reduzierung der Eingabegröße von 48x48 auf 32x32
- Verringerung der Filteranzahl im ersten Layer von 8 auf 4
- Entfernung der BatchNorm-Schichten

Problem 2: Quantisierung fehlgeschlagen

Fehler:

Copy

```
Quantisierung fehlgeschlagen: Quantized cudnn conv2d is currently limited to groups =
```

Ursache: Die depthwise separable Faltung verwendet Gruppenkonvolution (groups=8), die von der cuDNN-Quantisierung nicht unterstützt wird.

Implementierte Lösung: Verwendung des nicht-quantisierten Modells für den Export

Bessere Lösung (nicht implementiert):

- Ersetzung der Gruppenkonvolution durch Standardkonvolution
- Implementierung einer benutzerdefinierten Quantisierungsmethode

Problem 3: Export-Fehler mit KeyError

Fehler:

Copy

```
KeyError: 'mean'
Traceback (most recent call last):
  File "/home/emilio/Documents/ai/pizza/pizza-baking-detection-final.py", line 986, in
    f.write(", ".join(["{:.6f}f".format(x) for x in preprocess_params['mean']]))
KeyError: 'mean'
```

Ursache: Die Funktion `export_to_microcontroller` versucht auf den Schlüssel 'mean' im Dictionary `preprocess_params` zuzugreifen, aber dieser Schlüssel existiert nicht, da in `preprocess_params` nur 'mean_rgb' vorhanden ist.

Implementierte Lösung: Verwendung der bereits extrahierten lokalen Variablen `mean` und `std` anstelle von `preprocess_params['mean']` und `preprocess_params['std']`:

python Copy

```python
# In der Funktion export_to_microcontroller:

# Alte Version (fehlerhaft):
f.write(", ".join(["{:.6f}f".format(x) for x in preprocess_params['mean']]))

# Neue Version (korrigiert):
f.write(", ".join(["{:.6f}f".format(x) for x in mean]))
```

Die Korrektur bezieht sich auf die Zeilen, in denen auf die Schlüssel `'mean'` und `'std'` zugegriffen wird, indem stattdessen die lokalen Variablen verwendet werden, die zu Beginn der Funktion korrekt extrahiert wurden:

python

Copy

```
mean = preprocess_params.get('mean', preprocess_params.get('mean_rgb', [0.485, 0.456,
std = preprocess_params.get('std', preprocess_params.get('std_rgb', [0.229, 0.224, 0.2
```

7. Trainings- und Evaluierungsergebnisse

Das Training wurde nach 22 Epochen mit frühem Abbruch beendet.

Trainingsstatistiken:

- Beste Validierungsgenauigkeit: 25.00%
- Bester Validierungsverlust: ~1.26
- Trainingszeit: 3.34 Sekunden

Evaluierungsergebnisse:

- Gesamtgenauigkeit: 25.00%
- Makro-Präzision: 0.0417
- Makro-Recall: 0.1667
- Makro-F1: 0.0667
- Mikro-F1 (Genauigkeit): 0.2500

Klassenweise Leistung:

- `basic`: Precision=0.0000, Recall=0.0000, F1=0.0000, Support=8
- `combined`: Precision=0.2500, Recall=1.0000, F1=0.4000, Support=3
- `mixed`: Precision=0.0000, Recall=0.0000, F1=0.0000, Support=1

Hauptprobleme bei der Klassifikation:

1. Das Modell klassifiziert alle Beispiele als 'combined' (Klasse, die am zweithäufigsten im Datensatz vorkommt)
2. 'basic'-Samples werden immer falsch klassifiziert, obwohl sie die häufigste Klasse sind
3. Die Konfusionsmatrix zeigt, dass 8 'basic'- und 1 'mixed'-Beispiele als 'combined' fehlklassifiziert wurden

8. Export für RP2040

Die exportierten Dateien für den RP2040-Mikrocontroller umfassen:

- Header-Datei (`pizza_model.h`)

- Implementierungsdatei (`pizza_model.c`)
- Python-Referenzcode für die Verifikation (`verify_model.py`)
- README mit Implementierungsanleitung
- Beispiel-Makefile
- Hilfsklassen für Batteriemonitor und Kamerasteuerung

9. Probleme und Verbesserungsvorschläge

Datensatzprobleme:

- **Lösung**: Vergrößerung des Datensatzes und Ausbalancierung der Klassen
- **Lösung**: Mehr Augmentierungstechniken für die vorhandenen Bilder
- **Lösung**: Entfernung der leeren Klassen oder Hinzufügung von Beispielen

Modellarchitekturprobleme:

- **Lösung**: Ersetzung der Gruppenkonvolution durch Standardkonvolution für Quantisierung
- **Lösung**: Reduzierung der Eingabegröße für geringeren Speicherverbrauch
- **Lösung**: Vereinfachung der Architektur (weniger Filter, keine BatchNorm)

Trainingsprobleme:

- **Lösung**: Aggressivere Klassengewichtung für bessere Balance
- **Lösung**: Mehr Dropout für bessere Generalisierung
- **Lösung**: Transfer Learning von einem vortrainierten Modell

RAM-Optimierungsmöglichkeiten:

- **Lösung**: Int4-Quantisierung für noch kompaktere Modellgröße
- **Lösung**: Direktes Training mit reduzierten Bitstärken
- **Lösung**: Pruning (Ausdünnung) der Modellparameter nach dem Training

10. Zusammenfassung

Das RP2040 Pizza-Erkennungssystem ist ein hochoptimiertes CNN für Ressourcen-beschränkte Mikrocontroller. Das Modell hat nur 582 Parameter und benötigt minimal Speicherplatz (0.63 KB für das quantisierte Modell).

Trotz technischer Erfolge bei der Optimierung zeigt das Modell nur mäßige Klassifikationsleistung (25% Genauigkeit) aufgrund Datensatzeinschränkungen und Trainingsherausforderungen. Es gibt Raum für Verbesserungen sowohl beim Datensatz als auch bei der Modellarchitektur.

Der Exportprozess für den RP2040 wurde erfolgreich abgeschlossen, einschließlich der notwendigen Header, Implementierungsdateien und Dokumentation für die erfolgreiche Integration in ein eingebettetes System.