Leveraging the Power of C++ for Efficient Machine Learning on Embedded Devices

ADRIAN STANCIU

CppCon The C++ Conference
2023 October 01 - 06

Leveraging the power of C++ for efficient machine learning on embedded devices

Adrian Stanciu
adrian.stanciu.pub@gmail.com

CppCon, 2023

About me

- I am a software engineer from Romania
- I have a bachelor's degree in computer science from Politehnica University of Bucharest
- I have a master's degree in computer and network security from Politehnica University of Bucharest
- I have 12 years of professional experience in Linux system programming
- My main programming languages are C and C++
- I am interested in algorithms, data structures and operating systems
- I work at Bitdefender, global leader in cybersecurity, where I develop security software solutions which run on routers

Disclaimer

- This presentation is based on a personal project I worked on in my spare time and it is not directly related to the work I do at Bitdefender
- The outcome of this project is just a proof of concept
- Any mistakes are mine

Agenda

- Motivation
- Image classification
- Hand gesture recognition
- Summary
- Q&A

Motivation

Machine Learning (ML)

- Subfield of Artificial Inteligence (AI)
- Enables computers to learn from data and then use that knowledge to make predictions
- Applications:
  - Computer vision
  - Medicine
  - Search engines

Embedded devices

- Computing devices designed to perform specific tasks within larger systems
- Applications:
  - Consumer electronics (e.g. mobile phones, VR headsets)
  - Home automation (e.g. thermostats)
  - Medical equipment (e.g. pacemakers)
- Characteristics:
  - Limited hardware resources
  - Low power consumption
  - May have real-time performance constraints

Machine learning on embedded devices

- Alternative to cloud-based machine learning
- Advantages:
  - Real-time processing
  - Low latency
  - Reduced bandwidth usage
  - Offline operation
  - Improved privacy
- Disadvantages:
  - Compatibility with various hardware and software platforms
  - Slower updates

Using C++ for machine learning on embedded devices

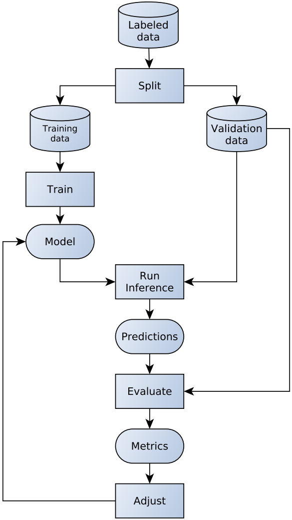
C++ is widely used in embedded systems:

- Was designed with efficiency in mind
- Provides low-level access to hardware resources
- Provides high-level abstractions

Image classification

Supervised learning

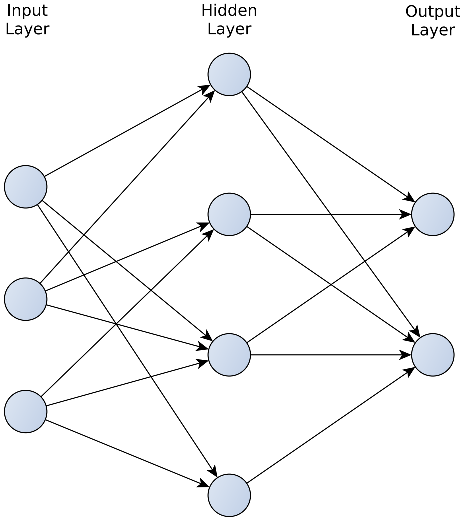
Machine learning paradigm in which an algorithm learns from labeled data to make predictions



Supervised learning

Source: xkcd.com

Neural network (NN)

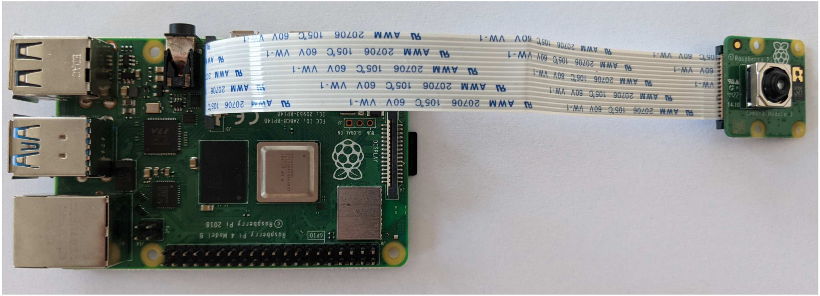


Convolutional neural network (CNN)

- Efficient in image classification
- A convolutional layer can apply filters to detect:
  - Edges
  - Shapes
  - Objects

Hardware setup

- Raspberry Pi 4 model B:
  - Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.8GHz
  - 4GB RAM
- 8GB microSD card
- Raspberry Pi Camera module 3



Software dependencies

- TensorFlow Lite
- OpenCV

MobileNet pre-trained model

- CNN architecture created by Google
- Trained on 1000 classes
- Accepts 224x224-pixel images, with 3 color channels per pixel (RGB)
- Labels are stored separate from the model:
  - `labels_mobilenet_quant_v1_224.txt` (11KB)
  - `mobilenet_v1_1.0_224_quant.tflite` (4.1MB)

Image classification algorithm

1. Load model and labels
2. Build interpreter
3. Allocate input and output tensors
4. Read image
5. Resize image
6. Copy resized image to input tensor
7. Run inference
8. Extract results from output tensor

Steps 1-3 represent the initialization phase
Steps 4-8 can be repeated multiple times

1. Load model and labels

```
// defined and properly initialized elsewhere:
// const char *model_path;
// const char *labels_path;

std::unique_ptr<tflite::FlatBufferModel> model{
  tflite::FlatBufferModel::BuildFromFile(model_path)
};

std::ifstream labels_ifs{labels_path};
std::string label;
std::vector<std::string> labels;
while (getline(labels_ifs, label)) {
  labels.push_back(std::move(label));
}
```

2. Build interpreter

```
// defined and properly initialized elsewhere:
// std::unique_ptr<tflite::FlatBufferModel> model;
// int num_threads;

tflite::ops::builtin::BuiltinOpResolver resolver;
tflite::InterpreterBuilder builder{*model, resolver};

builder.SetNumThreads(num_threads);

std::unique_ptr<tflite::Interpreter> interpreter;
builder(&interpreter);
```

3. Allocate input and output tensors

```
// defined and properly initialized elsewhere:
// std::unique_ptr<tflite::Interpreter> interpreter;

interpreter->AllocateTensors();
```

4. Read image

```
// defined and properly initialized elsewhere:
// const char *image_path;

cv::Mat image{cv::imread(image_path)};
```

5. Resize image

```cpp
// defined and properly initialized elsewhere:
// cv::Mat image;
// int required_image_width;
// int required_image_height;

cv::Mat resized_image;
cv::resize(
  image,
  resized_image,
  cv::Size{required_image_width, required_image_height}
);
```

6. Copy resized image to input tensor

```
// defined and properly initialized elsewhere:
// std::unique_ptr<tflite::Interpreter> interpreter;
// cv::Mat resized_image;

std::memcpy(
  interpreter->typed_input_tensor<uint8_t>(0),
  resized_image.data,
  resized_image.total() * resized_image.elemSize()
);
```

7. Run inference

```
// defined and properly initialized elsewhere:
// std::unique_ptr<tflite::Interpreter> interpreter;

interpreter->Invoke();
```

8. Extract results from output tensor

```
// defined and properly initialized elsewhere:
// std::unique_ptr<tflite::Interpreter> interpreter;
// std::vector<std::string> labels;

std::span<uint8_t> outputs{
  interpreter->typed_output_tensor<uint8_t>(0),
  labels.size()
};

std::vector<float> probabilities;
probabilities.reserve(labels.size());
for (auto output : outputs) {
  probabilities.push_back(
    1.0f * output / std::numeric_limits<uint8_t>::max()
  );
}
```

Demo

```
$ libcamera-jpeg --width=640 --height=480 -o out.jpeg
```

Demo - Good results

`0.96 | tennis ball`

Demo - Bad results

```
0.30 | vase
0.25 | bell pepper
ambiguous results
```

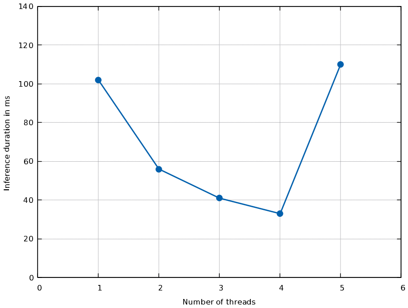
Demo - Bad results explanation

```
$ grep "tomato" labels_mobilenet_quant_v1_224.txt
$
```

Performance

- Compilation duration (on Raspberry Pi): 35s
- Binary size: 67KB
- Running duration: 1s

Number of threads	Inference duration (ms)
1	102
2	56
3	41
4	33
5	110



- Memory consumption with 4 threads: 93MB

Comparision with Python

Comparision made with TensorFlow Lite's `label_image.py`

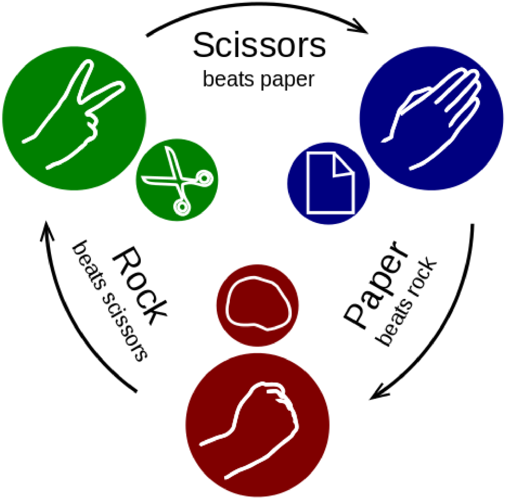
	C++	**Python**
Running duration (s)	1	7

Number of threads	**Inference duration (ms)**	
	C++	**Python**
1	102	118
2	56	62
3	41	42
4	33	33
5	110	170

	C++	**Python**
Memory consumption with 4 threads (MB)	93	320

Hand gesture recognition

Rock-Paper-Scissors



Source: wikipedia.org

Data

- Google MediaPipe's Rock-Paper-Scissors dataset for hand gesture recognition
  - Contains 125 images for each class
  - Images have various sizes
  - Images have 3 color channels per pixel (RGB)
- Laurence Moroney's Rock-Paper-Scissors dataset (published by Sani Kamal on Kaggle)
  - Contains 964 images for each class split into training data (840) and testing data (124)
  - Images are 300x300 pixels
  - Images have 3 color channels per pixel (RGB)

Train a Rock-Paper-Scissors model

- Transfer learning from a ResNet50 model
- 80/20 split between training and validation data
- Run in Python, on a laptop

Dataset	**Train duration**	**Model size (MB)**
Small	2m15s	23
Big	7m10s	23

Test the Rock-Paper-Scissors model

- Using the testing data of the big dataset (124 images per class)
- Small dataset results:

Gesture	Correct predictions	Accuracy (%)
Rock	108	87.09
Paper	124	100
Scissors	8	6.45
All	240 out of 372	64.51

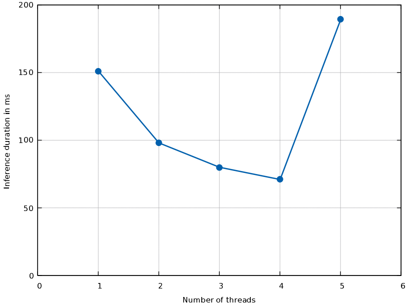
- Big dataset results:

Gesture	Correct predictions	Accuracy (%)
Rock	124	100
Paper	122	98.38
Scissors	97	78.22
All	343 out of 372	92.20

Performance

- Compilation duration (on Raspberry Pi): 35s
- Binary size: 67KB
- Running duration: 1s

Number of threads	Inference duration (ms)
1	151
2	98
3	80
4	71
5	189



- Memory consumption with 4 threads: 118MB

Capture camera stream

- libcamera backend
- gstreamer middleware
- OpenCV frontend

Capture camera stream

```
static constexpr const char *GstreamerPipeline{R"(
    libcamerasrc !
    video/x-raw,
    width=(int)640,
    height=(int)480,
    framerate=(fraction)10/1 !
    videoconvert !
    appsink
)"};

cv::VideoCapture camera{
  GstreamerPipeline,
  cv::CAP_GSTREAMER
};

cv::Mat image;
camera.read(image);
```

Live inference

1. Create image classifier
2. Open camera
3. Do in a loop:
    3.1 Read image from camera
    3.2 Use classifier to run inference on image
    3.3 Show predictions

Rock-Paper-Scissors game

On each round the AI:

1. Chooses Rock, Paper or Scissors uniformly at random
2. Infers the human's hand gesture
3. Computes the outcome

Summary

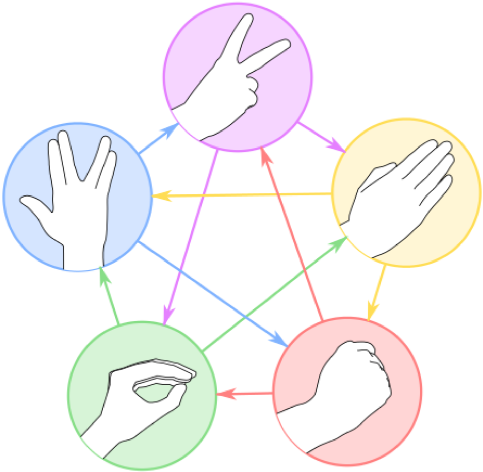
Recap

Task	**Model**	**Inference**
Image classification	pre-trained MobileNet	on-device in C++
Rock-Paper-Scissors hand gesture recognition	transfer learning from ResNet50	

Future work

- Reduce the model size and decrease the inference duration by converting images to grayscale
- Use `libcamera`'s API directly to reduce dependencies
- Run inference on GPU

Future work



Source: wikipedia.org

Conclusions

- Code isn't enough... data matters
- More diverse data leads to better models
- Building accurate models is an expert job
- Running on-device inference is straightforward
- Running on-device inference is here to stay

Repository

https://github.com/adrian-stanciu/cpp-embedded-ml

Resources

- https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/examples/python
- https://kaggle.com/datasets/sanikamal/rock-paper-scissors-dataset
- https://storage.googleapis.com/mediapipe-tasks/gesture_recognizer/rps_data_sample.zip
- https://tensorflow.org/lite/examples

Thank you!