UNIVERSIDADE FEDERAL DE SANTA CATARINA - JOINVILLE

Pós-Graduação em Engenharia de Sistemas Eletrônicos

Análise e Concepção de Sistemas Eletrônicos

Kelvin César de Andrade

Trabalho 01 - TinyML

Jaraguá do Sul

2023

Resumo

Este relatório apresenta o desenvolvimento de um algoritmo de aprendizado de máquina com o propósito de classificar dados de temperatura e utilizando conceitos de (*TinyML*). Inicialmente, o algoritmo foi treinado em um ambiente de computação em nuvem (*cloud*) e posteriormente embarcado em um microcontrolador ESP32, que foi integrado a um sensor de temperatura DHT11. As leituras de temperatura foram então inseridas no modelo para realizar a classificação. Para o desenvolvimento e conversão do modelo, foram utilizadas as ferramentas *TensorFlow* e *TensorFlow Lite*.

Lista de ilustrações

Lista de tabelas

Sumário

1 Descrição do problema

Visando desenvolver e testar conceitos de TinyML, foi definido um problema de classificação de temperatura para ser utilizado como exemplo. Esse problema consiste em classificar a temperatura em três classes distintas, sendo elas: frio, agradável e quente, baseado nos valores de temperatura medidos.

Cada uma dessas classes corresponde a um intervalo de temperatura específico, conforme descrito abaixo:

- Frio: valores menores que 20°C;
- Agradável: valores entre 20°C e 25°C;
- Quente: valores acima de 25°C.

Para testar o conceito de TinyML, o algoritmo de aprendizado de máquina foi desenvolvido e treinado em um ambiente de computação em nuvem, e posteriormente embarcado em um microcontrolador integrado a um sensor de temperatura DHT11 para realizar a classificação em tempo real.

Esse problema de classificação será utilizado como uma forma de estudo e entendimento da eficiência e limitações de se executar um algoritmo de aprendizado de máquina em sistemas embarcados.

2 Materiais utilizados

Este capítulo apresenta os materiais utilizados no desenvolvimento do estudo.

2.1 Microcontrolador ESP32

O ESP32 é um microcontrolador de baixo custo e baixa potência que é produzido pela empresa Espressif Systems e possui uma ampla gama de recursos. O microcontrolador ESP32-WROOM, usado neste estudo, possui os seguintes recursos de processador e memória:

- Possui 2 núcleos de processamento de 32 bits, rodando a uma velocidade de até 240 MHz;
- Possui 520 KB de memória SRAM disponível para aplicações e 448 KB de memória ROM.

No contexto deste trabalho, o ESP32 foi utilizado para realizar interface com o sensor de temperatura e execução do modelo de aprendizagem de máquina, através da biblioteca Tensorflow Lite. O modelo utilizado pode ser visto na Figura 1.

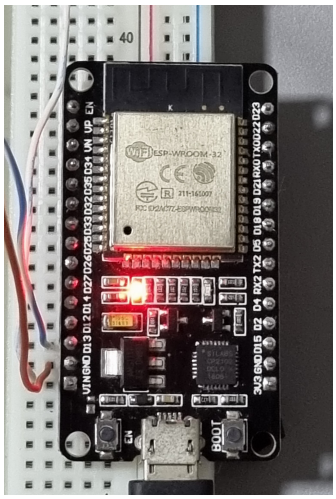

Figura 1 – Módelo ESP32-WROOM DevKit utilizado no estudo

2.2 Sensor de temperatura DHT11

O DHT11 é um sensor de umidade e temperatura que utiliza um único pino para comunicação com o microcontrolador. Ele é capaz de medir temperaturas entre 0 e 50 °C

com precisão de ±2 °C e umidade relativa do ar entre 20 e 90% com precisão de ±5% [1].

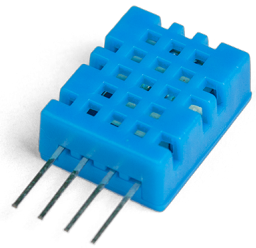

Figura 2 – Sensor de temperatura DHT11 utilizado no estudo

No contexto deste estudo, o DHT11 foi utilizado para medir a temperatura do ambiente em que o microcontrolador ESP32 estava operando. As leituras de temperatura foram usadas como entrada para o modelo de aprendizado de máquina desenvolvido usando o TensorFlow.

2.3 Tensorflow e Keras

O Tensorflow e o Keras, são bibliotecas de alto nível em linguagem Python, para a criação e treinamento de modelos de aprendizado de máquina. Ela fornece uma interface simples e intuitiva para a criação de modelos de forma rápida e fácil.

A biblioteca Tensorflow também está disponível em C++ através do pacote Tensorflow Lite. Desta maneira, é possível ter um ambiente mínimo para a execução de modelos de aprendizagem de máquina em dispositivos com menos recursos computacionais.

2.4 Open Meteo

O Open Meteo é um serviço online e gratuito que fornece dados meteorológicos de várias localidades em todo o mundo, possibilitando o acesso a informações em tempo real ou em arquivos históricos sobre temperatura e umidade, entre outros dados.

No âmbito deste trabalho, o Open Meteo foi utilizado para coletar dados para a construção e treinamento do modelo de classificação de temperatura.

3 Desenvolvimento

Neste capítulo serão abordados o procedimento de construção do modelo de aprendizagem de máquina até a implementação no dispositivo embarcado.

3.1 Tensorflow Lite

O Tensorflow Lite para microcontroladores, é um pacote do Tensorflow e foi projetado para executar modelos de aprendizado de máquina em microcontroladores e outros dispositivos com apenas alguns kilobytes de memória. O tempo de execução principal ocupa apenas 16 KB em um Arm Cortex M3 e pode executar muitos modelos básicos. Não requer suporte de sistema operacional, quaisquer bibliotecas padrão de C ou C++, ou alocação dinâmica de memória [2].

Visando compreender o uso da ferramenta Tensorflow Lite e sua compatibilidade com o dispositivo ESP32, foram realizados testes iniciais com o exemplo "Hello World" disponível em <https://github.com/espressif/tflite-micro-esp-examples>.

O modelo "Hello World" do Tensorflow Lite é um modelo de aprendizado de máquina simples treinado para replicar uma função seno.

No entanto, ao tentar compilar os arquivos e transferi-los para o ESP32, foram gerados vários erros. Ao ler a documentação do repositório e do exemplo, verificou-se que a biblioteca do Tensorflow Lite é compatível apenas com a versão 4.4 do *esp-idf*, que é a plataforma de desenvolvimento do ESP32.

Assim, foi necessário desinstalar a versão atual (5.0) e instalar a versão mais antiga por meio dos seguintes comandos:

Código 3.1 – Comandos para instalar versão 4.4 da ESP-IDF

```
1 mkdir -p $HOME/.local/opt/esp
2 cd $HOME/.local/opt/esp
3 git clone --recursive https://github.com/espressif/esp-idf.git
4 cd esp-idf
5 git checkout -b v4.4.4 v4.4.4
6 git submodule update --init --recursive
7 bash install.sh esp32
8 . ./export.sh
```

Com a versão anterior do *esp-idf* instalada, foi possível compilar a biblioteca Tensorflow Lite juntamente com o código do modelo e transferi-los para o microcontrolador ESP32. Ao executar o programa, o microcontrolador começou a reportar via porta serial

os valores da função senoide, indicando que a biblioteca estava funcionando corretamente. A Figura 3 exemplifica o funcionamento do modelo.

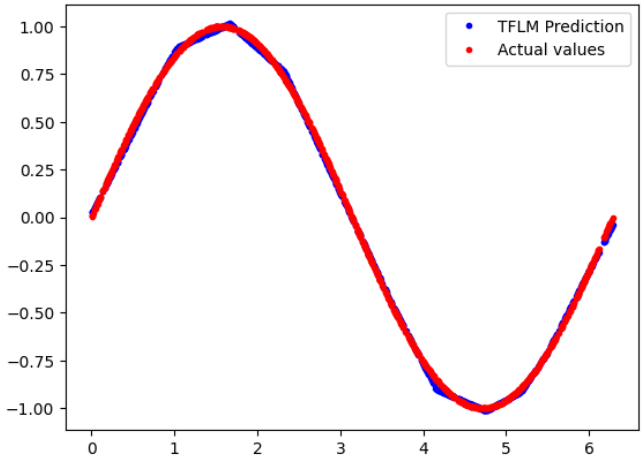

Figura 3 – Comparação da saída do modelo "Hello World" com o valor esperado da senóide disponível na documentação do exemplo

Com o ambiente sendo executado, foi possível então estudar a estrutura da biblioteca do Tensorflow Lite e como foi realizado para inserir o modelo criado no programa em questão. Nas próximas seções, será detalhado o processo adotado para realizar essas tarefas.

3.2 Desenvolvimento do modelo

Após compilar e executar o exemplo do Tensorflow Lite, foi iniciado o desenvolvimento do modelo para categorizar os dados de temperatura. Este desenvolvimento foi realizado no ambiente do Google Colaboratory, que permite executar Jupyter Notebooks em linguagem Python, e conta com suporte à GPUs quando necessário. O desenvolvimento foi realizado com base no exemplo "Hello World" principalmente nas etapas de conversão para Tensorflow Lite.

3.2.1 Obtenção e categorização das amostras

O processo de construção do modelo iniciou-se com a obtenção dos dados que seriam utilizados no treinamento. Para isso, foram coletados dados históricos de temperatura da cidade de Jaraguá do Sul, localizada em Santa Catarina, durante o período de 2022. A coleta dos dados foi realizada por meio do site Open Meteo utilizando o código em Python mostrado a seguir.

Código 3.2 – Coleta dos dados de temperatura da API do Open Meteo

```
1 import requests
2 url = "https://archive-api.open-meteo.com/v1/era5"
```

```
params = {
    "latitude": -26.49,
    "longitude": -49.07,
    "start_date": "2022-01-01",
    "end_date": "2022-12-31",
    "hourly": "temperature_2m"
}

response = requests.get(url, params=params)

if response.status_code == 200:
    data = response.json()
    df = pd.DataFrame(data["hourly"])
else:
    print("Error:", response.status_code, response.text)
```

Como resultado da API, foram retornados 8760 pontos de temperatura, correspondendo a uma medição a cada hora durante o período de um ano. Para uma melhor manipulação dos dados na linguagem Python, os valores foram convertidos para um Pandas Dataframe, que transforma os dados em uma estrutura de tabela.

Em seguida, os dados de temperatura foram classificados em três categorias distintas: "frio" , "agradável" e "quente" conforme mencionado no Capítulo 1. A faixa de valores para cada categoria foi selecionada de forma arbitrária para fins de exemplo. As categorias foram mapeadas em um *enum* de zero a dois, como ilustrado no código a seguir, que foi utilizado para classificar os dados de temperatura coletados.

Código 3.3 – Código para rotulação dos dados de temperatura

```
from enum import Enum

class ClassesTemperatura(Enum):
    Frio = 0
    Agradavel = 1
    Quente = 2

def classifica_temperatura(temp):
    if temp < 20:
        return ClassesTemperatura.Frio.value
    elif temp >= 20 and temp <= 25:
        return ClassesTemperatura.Agradavel.value
    else:
        return ClassesTemperatura.Quente.value

df["classe"] = df["temperatura"].apply(classifica_temperatura)
```

A classificação dos dados de temperatura com base nos limites dimensionados

acima resultou em 4352 amostras classificadas como “frio”, 3191 amostras classificadas como “agradável” e 1217 amostras classificadas como “quente”. A Tabela 1 apresenta a estrutura final dos dados coletados e classificados.

time	temperatura	classe
2022-01-01T00:00	21.7	1
2022-01-01T01:00	21.4	1
2022-01-01T02:00	21.0	1
2022-01-01T03:00	20.7	1
2022-01-01T04:00	20.4	1
...	...	...

Tabela 1 – Estrutura dos dados de temperatura e suas classes

3.2.2 Desenvolvimento do modelo e treinamento

Com os dados de temperatura coletados e processados, iniciou-se o desenvolvimento do algoritmo de aprendizado de máquina para classificação.

O primeiro passo foi realizar a normalização dos valores de temperatura, utilizando a função *StandardScaler* da biblioteca scikit-learn. A normalização é uma etapa importante para garantir que os dados estejam em uma escala contínua, o que pode melhorar a precisão e o tempo de convergência do modelo [3].

Em seguida, os dados foram separados aleatoriamente em conjuntos de treinamento e teste, com 75% das amostras para treinamento e 25% para teste. Essa separação é fundamental para evitar o *overfitting* do modelo. O código para normalização e separação dos dados é apresentado a seguir.

Código 3.4 – Normalização e separação dos dados em treinamento e teste

```
from sklearn.preprocessing import StandardScaler, LabelEncoder
X = df['temperatura'].values.reshape(-1, 1)
y = df['classe'].values
scaler = StandardScaler()
X = scaler.fit_transform(X)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.25)
```

Em seguida, iniciou-se a construção do modelo. Primeiramente era pretendido utilizar a técnica de *Support Vector Machine* (SVM), que é um algoritmo de aprendizado supervisionado muito utilizado para classificação de dados [3]. Porém, utilizando a biblioteca do *skilearn*, onde este algoritmo já está disponível, verificou-se que não seria possível realizar a conversão do modelo para Tensorflow Lite.

Assim, o modelo foi então construído utilizando a biblioteca Keras que possui compatibilidade para conversão em um modelo Tensorflow Lite. Esta biblioteca possui uma interface de alto nível para a criação de uma Rede Neural Artificial (RNA). Desta maneira, foi criada uma RNA com saída logística (mediante uma função sigmoide), que é uma função de ativação comumente usada em RNAs para problemas de classificação binária ou multiclasse. Esta função estrutura a saída da rede em um intervalo de 0 a 1, permitindo que a saída seja interpretada como uma probabilidade de pertencer a cada classe.

O modelo desenvolvido possui duas camadas totalmente conectadas, podendo ser visto na Figura 4. Deste modo, cada camada possui as seguintes características:

- A primeira camada tem 5 neurônios e utiliza a função de ativação ReLU (*Rectified Linear Unit*), que é comumente usada em camadas ocultas de redes neurais profundas, pois ajuda a lidar com o problema de desaparecimento de gradiente e pode acelerar o processo de treinamento [4]. A entrada para esta camada é uma única dimensão, que é o valor de temperatura;
- A segunda camada tem 3 neurônios e utiliza a função de ativação *sigmoid*, que é uma função logística. Neste caso, ela é utilizada para mapear as saídas da camada anterior em probabilidades de pertencer a cada uma das três classes possíveis (frio, agradável ou quente).

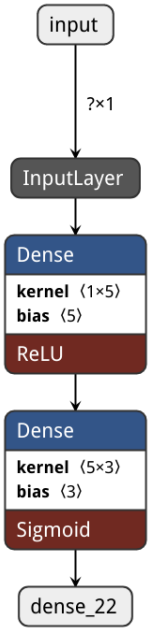

Figura 4 – Arquitetura da rede neural criada para classificação dos valores de temperatura

Em seguida, o modelo foi treinado usando a função de perda *categorical_crossentropy*, que é uma medida utilizada em problemas de classificação multiclasse. A métrica utilizada para avaliar o desempenho do modelo é a acurácia do mesmo durante o treinamento.

Durante o processo de treinamento, o modelo é alimentado com o conjunto de dados de treinamento várias vezes, em um número de ciclos chamado de época (*epochs*). Cada época é composta por uma passagem completa pelos dados de treinamento, onde o modelo atualiza seus pesos. O uso de várias épocas permite que o modelo melhore sua capacidade de generalização para o problema. Neste caso, o processo de treinamento foi realizado em 10 épocas, que foi suficiente para atingir uma boa acurácia. O código final do modelo é mostrado a seguir.

Código 3.5 – Modelagem da rede neural

```
model = Sequential()
model.add(Input(shape=(1,)))
model.add(Dense(5, activation='relu'))
model.add(Dense(3, activation='sigmoid'))

model.compile(loss='categorical_crossentropy', optimizer='adam',
    metrics=['accuracy'])

model.fit(X_train, to_categorical(y_train), epochs=10, verbose=1)
```

A acurácia do modelo atingiu um valor de 99,86% durante o processo de treinamento, e 99,58% no conjunto de teste, mostrando uma boa precisão na classificação dos dados de temperatura. A Figura 5 mostra a evolução da precisão e perda durante o processo de treinamento do modelo.

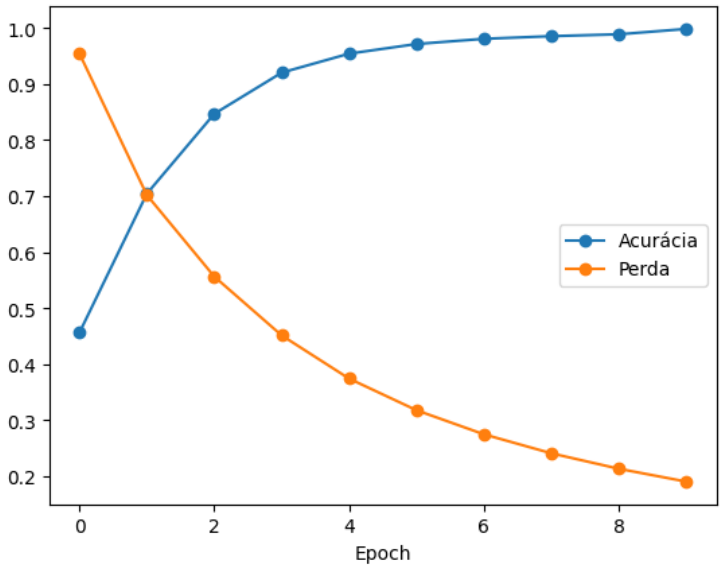

Figura 5 – Acurácia e perda durante o processo de treinamento do modelo criado

Por fim, o modelo foi salvo para uso nas próximas etapas do trabalho.

3.2.3 Conversão para Tensorflow Lite

Após criar o modelo Tensorflow, o próximo passo é convertê-lo para um modelo *Lite*, que pode ser executado em um microcontrolador com recursos limitados.

Para realizar a conversão, utilizamos o pacote Tensorflow Lite disponível na biblioteca do Tensorflow. A documentação oficial do Tensorflow Lite (disponível em [5]), fornece exemplos e informações sobre como realizar a conversão, e foi utilizada como base nesse processo.

O trecho de código abaixo foi usado para converter o modelo criado para a versão Tensorflow Lite. Com objetivo de facilitar o processo, não foi aplicado nenhum método de quantização no processo de conversão do modelo. A quantização é uma otimização que permite melhorar o desempenho e adicionar suporte para dispositivos sem unidade ponto flutuante, em troca de uma redução na precisão do modelo.

Código 3.6 – Conversão do modelo para Tensorflow Lite

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)

converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.inference_input_type = tf.float32
converter.inference_output_type = tf.float32

tflite_model = converter.convert()
with open(f'temp_model.tflite', 'wb') as f:
    f.write(tflite_model)
```

Desta maneira, o modelo está pronto para ser carregado no *firmware* do ESP32.

3.3 Implementação do modelo no ESP32

3.3.1 Importação do modelo Tensorflow Lite

Para executar o modelo em Tensorflow Lite no ESP32, foi utilizado como guia o exemplo testado e mencionado na seção 3.1, pois toda a estrutura de compilação da biblioteca do Tensorflow Lite já está configurada.

O arquivo gerado em *.tflite* foi então convertido para C++ usando a ferramenta "xdd" no Linux através do comando a seguir.

Código 3.7 – Comando para converter arquivo .tflite em hexdump para C++

```
xxd -i ./model/temp_model.tflite > ./main/model.cc
```

Este comando gera um arquivo com as instruções do modelo em hexadecimal associadas a uma variável, e que poderá ser importado pelo *runtime* do Tensorflow Lite. O arquivo gerado pode ser visto na Figura 6.

```
unsigned char __model_temp_model_tflite[] = {
  0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33,
  ....
};
unsigned int __model_temp_model_tflite_len = 1800;
```

Figura 6 – Arquivo gerado pelo comando "xdd" com instruções do modelo Tensorflow Lite em C++

3.3.2 Ambiente de execução do Tensorflow Lite

Em seguida foi desenvolvida uma classe chamada $TemperatureClassifier$ para realizar a configuração do ambiente do Tensorflow Lite e execução do modelo. O processo de configuração de um modelo em Tensorflow Lite é realizado nas seguintes etapas:

1. **Carregamento do modelo:** nesta etapa, o modelo salvo é anteriormente carregado, é mapeado em uma estrutura que pode ser interpretada pelo Tensorflow Lite;
2. **Definição das operações do interpretador:** é necessário definir quais operações serão carregadas para interpretar o modelo. É importante notar que quanto mais operações forem definidas, maior será o consumo de memória para suportar o modelo de aprendizagem de máquina;
3. **Criação do interpretador:** nesta etapa, o interpretador é criado, passando o modelo, as operações e uma área de memória para alocar os tensores de entrada e saída;
4. **Alocação dos tensores:** com o interpretador criado com sucesso, é possível alocar a memória para os tensores de entrada e saída, e então criar ponteiros para manipulá-los.

Na etapa 2 do processo de configuração, não foi possível identificar de maneira automática quais as operações necessárias para o modelo gerado. No exemplo apresentado na seção seção 3.1, o autor utilizou a instrução $statictflite :: AllOpsResolverresolver$, que carrega todas as operações disponíveis.

Entretanto, para otimizar o tamanho do ambiente de execução, foram realizados testes com as operações disponíveis. Com base nestes testes, e na arquitetura do modelo mostrado na Figura 4, foram definidas as duas operações necessárias: *FullyConnected* e *Logistic*.

Para configurar as operações do interpretador, o código em C++ a seguir foi utilizado:

Código 3.8 – Configuração das operações para interpretação do modelo

```
1 tflite::MicroMutableOpResolver<2>*resolver = new tflite::
     MicroMutableOpResolver<2>();
2
3 resolver->AddFullyConnected();
4 resolver->AddLogistic();
```

Assim, foi possível definir manualmente as operações necessárias para a interpretação do modelo. O tamanho final da alocação de memória utilizado pelo modelo ficou em 600 bytes (pode ser visualizado através do comando $interpreter->arena\_used\_bytes()$).

Nesta mesma classe, também foi desenvolvido um método de *predict*, que realiza a inferência do modelo com base no valor de temperatura passado como parâmetro. Esse método carrega o valor de temperatura no tensor de entrada do interpretador, invoca a inferência e, em seguida, lê os tensores de saída do modelo, sendo que, neste caso, há três tensores de saída correspondendo a cada uma das três classes.

Um dos problemas enfrentados nesta etapa foi relacionado à normalização dos dados durante o treinamento do modelo, pois valores normais de temperatura estavam gerando resultados errados de classificação.

Para isso, foi verificado que a biblioteca *Scikit Learn* utiliza, no pacote *StandardScaler*, a função Z-Score (Equação 3.1). Esta função é calculada pela subtração da média μ e pela divisão do desvio padrão σ do conjunto de dados [6].

$$z_{score} = \frac{x - \mu}{\sigma} \tag{3.1}$$

Desta maneira, foi necessário armazenar a média e o desvio padrão do conjunto de dados na classe desenvolvida, e aplicar a normalização nos valores de temperatura antes de passar para o tensor de entrada do modelo através da equação do Z-Score. Não foi encontrado como obter estes valores a partir do modelo que é gerado.

O código completo da implementação da classe $TemperatureClassifier$ pode ser visto no Apêndice A.

3.3.3 Integração com o sensor DHT11

Para testar o sistema, foi integrado ao microcontrolador ESP32 um sensor de temperatura DHT11, por meio da biblioteca disponível em <https://github.com/UncleRus/esp-idf-lib>.

O sensor DHT11 possui quatro pinos, sendo o terceiro sem uso. A conexão do ESP32 com o sensor foi realizada da seguinte maneira:

- Alimentação: conectado o pino 01 do sensor na saída de $5V$ do ESP32;
- Comunicação: conectado o pino 02 do sensor, na GPIO 13 do ESP32. Conforme é descrito no *datasheet* do sensor [1], também foi adicionado um resistor de pull-up de 1 kΩ entre o barramento de alimentação e a interface de comunicação.
- Referência de tensão: conectado o pino 04 do sensor ao GND do ESP32.

A Figura 7 ilustra a conexão entre os dois dispositivos na prática.

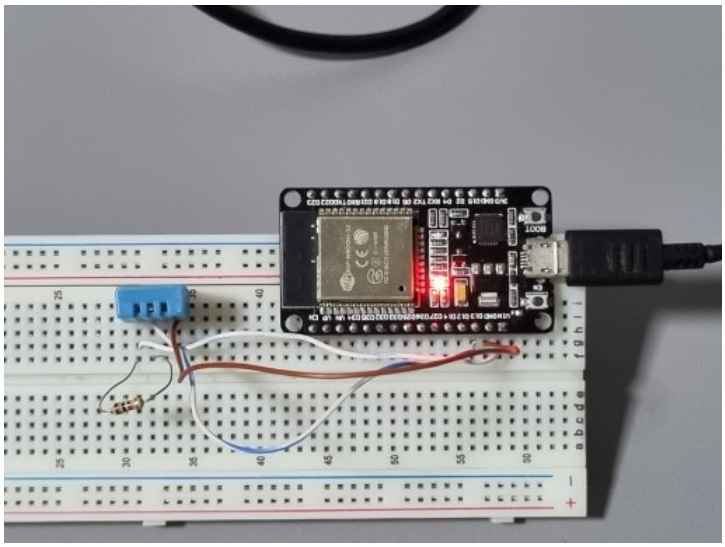

Figura 7 – Conexão entre o ESP32 e o sensor de temperatura DHT11 para o trabalho

3.3.4 Execução do algoritmo

Para realizar a leitura do sensor e classificação do valor de temperatura, foi desenvolvido o código apresentado abaixo. Primeiramente, a classe responsável por manipular o modelo do Tensorflow Lite é inicializada. Em seguida, é feita a chamada da biblioteca responsável pela leitura do sensor DHT11. Caso a comunicação com o sensor seja bem-sucedida e o valor de temperatura seja lido corretamente, o mesmo é submetido ao método *predict* do modelo, que retorna a qual classe a temperatura pertence.

Código 3.9 – Leitura do sensor DHT11 e classificação do valor de temperatura usando o modelo gerado

```
1
2 TemperatureClassifier* temp_model = new TemperatureClassifier();
3 float temperature, humidity;
4 int prediction = -1;
```

```
5  if (dht_read_float_data(SENSOR_TYPE, GPIO_NUM, &humidity, &
       temperature) == ESP_OK) {
6      init_time = esp_timer_get_time();
7      prediction = temp_model->predict(&temperature);
8      total_time = esp_timer_get_time() - init_time;
9      printf("-> Temperatura: %f, Classificacao: ", temperature);
10     print_temperature_class(&prediction);
11     printf(", Tempo de execucao: %lld us\n", total_time);
12 }
```

A execução deste código no ESP32, reproduz a saída mostrada na Figura 8, informando a temperatura lida, a classe que pertence e o tempo computacional para realizar a inferência.

```
Iniciando leitura do sensor DHT11
-> Temperatura: 27.000000, Classificação: 2 (Quente), Tempo de execução: 64 us
-> Temperatura: 27.000000, Classificação: 2 (Quente), Tempo de execução: 49 us
-> Temperatura: 27.000000, Classificação: 2 (Quente), Tempo de execução: 45 us
```

Figura 8 – Saída do ESP32 informando a temperatura lida e a classe

É possível verificar que o tempo de execução é na ordem de dezenas microssegundos, mostrando um bom desempenho para o modelo criado.

4 Resultados

Com objetivo de medir o tempo de execução e a precisão do modelo convertido para Tensorflow Lite, foi desenvolvida uma função que testa valores de temperatura entre 10 à 30 °C, ao passo de 0.5°C. A função utilizada no ESP32 é mostrada no Apêndice B. Os resultados podem ser vistos na Tabela 2.

Métrica	Valor
Tempo médio de execução	39,78 μs
Acurácia	87,80%

Tabela 2 – Resultado da execução do módelo Tensorflow Lite no ESP32

A Tabela 2 apresenta o desempenho da inferência do modelo executado no ESP32, que é da ordem de microssegundos, um tempo pequeno considerando que se trata de uma rede neural executada em um dispositivo com recursos limitados.

No entanto, a acurácia do modelo sofreu uma redução em relação ao modelo original, atingindo 87,80%, enquanto anteriormente chegava a quase 100.00% de precisão. Apesar da queda, esse valor ainda é satisfatório. Ao analisar os *logs* da função, percebeu-se que o modelo Lite está tendo dificuldades em classificar a classe "agradável", na qual começa a predizer como "quente".

O motivo desse erro precisa ser avaliado com maior cautela, verificando se está relacionado à precisão do ponto flutuante do dispositivo em relação ao modelo original ou se está relacionado à biblioteca do Tensorflow Lite.

Outra avaliação realizada foi da alteração do número de operações carregadas na inicialização do ambiente do Tensorflow Lite.

Operações	Tempo médio de execução [μs]	Acurácia [%]
AllOpsResolverresolver	64.98	87.80
AddFullyConnected e *AddLogistic*	42.44	87.80

Tabela 3 – Comparativo entre o carregamento diferentes operações carregadas no modelo

Podemos observar na Tabela 3 que há uma diferença significativa no tempo de execução entre carregar todas as operações no ambiente e carregar apenas as operações necessárias. Ao utilizar apenas as operações necessárias, houve uma redução de 34,68% no tempo de execução, sem prejudicar a acurácia do modelo. Esse resultado demonstra a importância de se escolher cuidadosamente quais operações serão carregadas no ambiente do Tensorflow Lite.

Com esses resultados, pode-se concluir que é possível realizar a classificação de valores de temperatura lidos em tempo real, de forma eficiente, com tempos de execução na ordem de microssegundos, mesmo em dispositivos embarcados com limitações de processamento e memória, como o caso do ESP32. Essa possibilidade permite a criação de aplicações inteligentes que podem ser executadas localmente no dispositivo, sem a dependência de serviços em nuvem. Apesar da redução na acurácia do modelo em relação à versão original, ainda é possível atingir resultados satisfatórios.

Referências

1 DHT11 Technical Data Sheet Translated Version. Disponível em: <https://www.mouser.com/datasheet/2/758/DHT11-Technical-Data-Sheet-Translated-Version-1143054.pdf>. Acesso em: 30 de Abril 2023. Citado 2 vezes nas páginas 7 e 17.

2 GOOGLE. *TensorFlow Lite for Microcontrollers*. 2022. <https://www.tensorflow.org/lite/microcontrollers>. Acesso em: 24 de Abril 2023. Citado na página 8.

3 MÜLLER, A. C.; GUIDO, S. *Introduction to machine learning with Python: a guide for data scientists*. [S.l.]: "O'Reilly Media, Inc.", 2016. Citado na página 11.

4 BROWNLEE, J. Rectified linear activation function for deep learning neural networks. *Machine Learning Mastery*, 2019. Acesso em: 24 de Abril 2023. Disponível em: <https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/>. Citado na página 12.

5 TensorFlow. *Conversão de modelos para TensorFlow Lite*. <https://www.tensorflow.org/lite/convert?hl=pt-br#python_api>. Acesso em: 24 de Abril 2023. Citado na página 14.

6 SCIKIT Learn StandardScaler. <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler>. Acesso em: 01 de Maio 2023. Citado na página 16.

A Classe TemperatureClassifier

Código A.1 – neural_network.h

```
// Modelo que sera carregado
#include "model.h"

// Classe para classificacao do modelo de temperatura
class TemperatureClassifier {
   public:
      TemperatureClassifier();
      int predict(float* temperature);

   private:
      const tflite::Model* model_;
      tflite::MicroInterpreter* interpreter_;
      TfLiteTensor* input_;
      TfLiteTensor* output_;

      // Numero de classes na saida do modelo
      const int number_of_classes_ = 3;

      // Valores de media e desvio padrao utilizados na normalizacao
      const float norm_mean_ = 20.07228311;
      const float norm_std_ = 4.49116319;

      uint8_t tensor_arena_[TENSOR_ARENA_SIZE];
};
```

Código A.2 – neural_network.c

```
\#include "neural_network.h"

TemperatureClassifier::TemperatureClassifier() {
    // Carrega o modelo na memoria (model.h)
    model_ = tflite::GetModel(__model_temp_model_tflite);
    if (model_->version() != TFLITE_SCHEMA_VERSION) {
        printf("Versao do modelo (%d) != (%d).\n", model_->version()
            , TFLITE_SCHEMA_VERSION);
        return;
    }
    //Operacoes necessarias no modelo.
    tflite::MicroMutableOpResolver<2>*resolver = new tflite::
       MicroMutableOpResolver<2>();
    resolver->AddFullyConnected();
    resolver->AddLogistic();
```

```
14      // Define o interpretador para execucao do modelo
15      interpreter_ = new tflite::MicroInterpreter(
16          model_,
17          *resolver,
18          tensor_arena_,
19          TENSOR_ARENA_SIZE
20      );
21      // Alocam memoria para os tensores do modelo
22      TfLiteStatus allocate_status = interpreter_->AllocateTensors();
23      if (allocate_status != kTfLiteOk) {
24          MicroPrintf("AllocateTensors() falhou\n");
25          return;
26      }
27      // Computa o tamanho da memoria utilizada pelo modelo
28      size_t used_bytes = interpreter_->arena_used_bytes();
29      printf("Tensorflowlite - Total de memoria usada: %d bytes\n",
            used_bytes);
30      // Ponteiros para os tensores de entrada e saida
31      input_ = interpreter_->input(0);
32      output_ = interpreter_->output(0);
33  }
34
35  int TemperatureClassifier::predict(float* temperature){
36      // Normalizacao atraves da equacao Z-Score
37      float norm_temp = (*temperature - norm_mean_) / norm_std_;
38      // Transfere o valor normalizado para o tensor de entrada
39      input_->data.f[0] = norm_temp;
40      // Executa a inferencia do modelo
41      TfLiteStatus invoke_status = interpreter_->Invoke();
42      if (invoke_status != kTfLiteOk) {
43          printf("Invoke() falhou.\n");
44          return -1;
45      }
46      // Iteracao pelos valores de saida para encontrar o maior valor
47      int max_index = 0;
48      float value = output_->data.f[0];
49      for (int i = 1; i < number_of_classes_; i++) {
50          if (output_->data.f[i] > value) {
51              max_index = i;
52          }
53      }
54      // Retorna o maior index, que no caso e a propria classe
55      return max_index;
56  }
```

B Rotina de teste para o modelo em Tensorflow Lite

Código B.1 – Função para teste do modelo de classificação de temperatura no ESP32

```
void test_temperature_model(TemperatureClassifier* model) {
  // Testa todas as temperaturas entre 10 e 30 graus, com incremento de 0.5 graus
  float temperature = 10.0;
  float time_avg = 0;
  int correct_predictions = 0;
  printf("Testando modelo com todas as temperaturas entre 10 e 30 graus:\n");


  while (temperature <= 30.0) {
    init_time = esp_timer_get_time();
    int prediction = model->predict(&temperature);
    total_time = esp_timer_get_time() - init_time;

    printf("- Temperatura: %f, Resultado: %d (executado em %lld us)\n", temperature, prediction, total_time);
    time_avg += total_time;
    if ((temperature < 20 && prediction == 0) || (temperature >= 20 && temperature <= 25 && prediction == 1) || (temperature > 25 && prediction == 2)) {
      correct_predictions++;
    } else {
      ESP_LOGE("Teste", "Erro na predicao da temperatura %f", temperature);
    }
    temperature += 0.5;
  }
  printf("- Tempo medio de execucao: %f us\n", time_avg/((30.0-10.0)/0.5+1));
  printf("- Porcentagem de acerto: %f %%\n", (float) correct_predictions/((30.0-10.0)/0.5+1)*100);
}
```