

# Aprendizagem de Máquina

[Início da aula 6]

**Resumo:** Na fase 5, desenvolveremos um programa que reconhece dígito manuscrito dentro das placas em vídeos, usando aprendizado de máquina. Na fase 6, desenvolveremos um sistema que dirige carrinho autonomamente, de acordo com os dígitos que escritos nas placas. Para isso, utilizaremos o conceito de máquina de estados finita.

## 1 Introdução

A parte teórica de aprendizagem de máquina já foi vista na disciplina obrigatória PSI3471 e complementada na disciplina optativa PSI3472 ([www.lps.usp.br/hae/apostila](http://www.lps.usp.br/hae/apostila)).

Em particular, o reconhecimento de dígitos manuscritos usando banco de dados MNIST, linguagem C++ e algoritmos de aprendizagem convencionais (não deep learning) foi abordado na apostila [classif-ead.odt](#). As apostilas [densakeras-ead.odt](#) e [convkeras-ead](#) explicam como reconhecer os dígitos manuscritos em Python e Tensorflow/Keras/PyTorch usando rede neural densa e rede neural convolucional.

## 2 Quinta fase do projeto

[Lição de casa única da aula 6] Modifique o programa fase3.cpp para que, além de detectar a placa, leia o dígito manuscrito inscrito dentro da placa, quando a placa estiver suficientemente próxima da câmera, escrevendo o dígito lido no vídeo de saída (figura 1). Use OpenMP e OpenCV v3 para acelerar o processamento (compila com o comando “compila fase5 -c -v3 -omp”). Executando:

```
$ fase5 capturado.avi quadrado.png locarec.avi
```

o seu programa deve ler os arquivos capturado.avi e quadrado.png e gerar o vídeo locarec.avi. Um exemplo de saída está em:

[locarec.mp3](#) [locarec.avi](#)

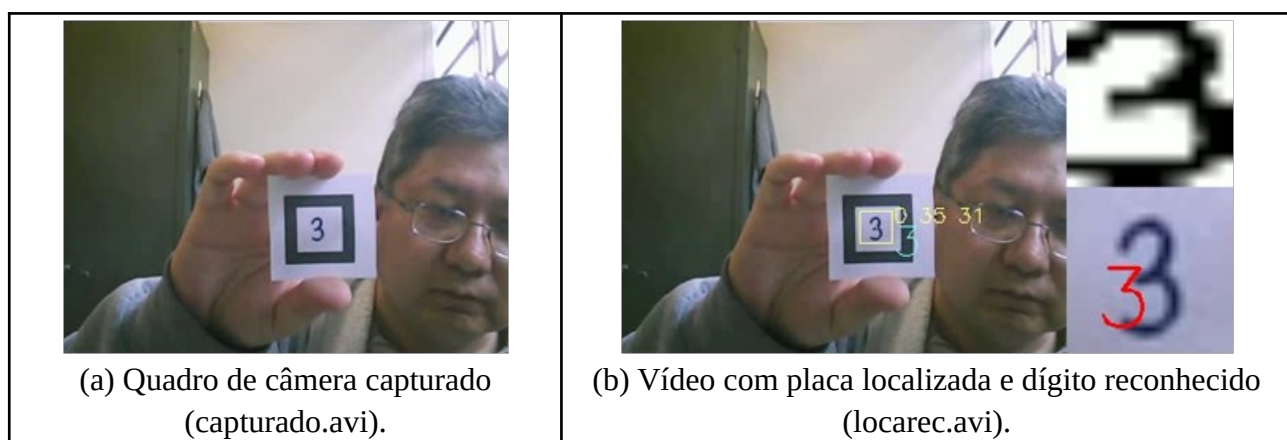


Figura 1: Localização da placa com reconhecimento do dígito manuscrito.

**Nota:** Estou inserindo a subimagem do dígito extraída na parte direita inferior da imagem (figura 1b). Também estou inserindo essa imagem após o pré-processamento para acertar brilho/contraste e eliminar linhas/colunas brancas na parte direita superior. Isto facilita debugar o programa e recomendo que vocês façam algo parecido. Mas você não é obrigado a fazer isso: basta escrever em algum lugar do vídeo de saída o valor do dígito reconhecido.

**Nota:** Se quiser testar apenas o reconhecimento dos dígitos (sem a localização da placa), pode usar o vídeo abaixo, com os dígitos já segmentados:

<http://www.lps.usp.br/hae/apostilaraspi/digitos.avi>

**Nota:** Pode usar a base de dados de dígitos manuscritos MNIST para treinar o seu algoritmo de classificação de dígitos manuscritos:

<http://www.lps.usp.br/hae/apostilaraspi/mnist.zip>

Para quem instalou Cekeikon, uma cópia está em: (...) /cekeikon5/tiny\_dnn/data

Site original: <http://yann.lecun.com/exdb/mnist/>

**Nota:** Para facilitar, deixe todos os arquivos de entrada necessários no seu diretório default. Os arquivos necessários são:

capturado.avi (<http://www.lps.usp.br/hae/apostilaraspi/capturado.avi> )

quadrado.png (<http://www.lps.usp.br/hae/apostilaraspi/quadrado.png> )

t10k-images.idx3-ubyte

t10k-labels.idx1-ubyte

train-images.idx3-ubyte

train-labels.idx1-ubyte

Os quatro últimos arquivos são obtidos descompactando mnist.zip. Use obrigatoriamente os nomes originais dos arquivos acima.

**Nota:** Dentro do Cekeikon, há a classe MNIST que faz a leitura da base de dados MNIST. Para isso, execute os comandos:

```
MNIST mnist(14, true, true);  
// Reduz o tamanho das imagens para 14x14. inverte_cores=true, ajustaBbox=true  
mnist.le("/home/hae/haebase/mnist");  
// Faz a leitura do MNIST. O parametro e' o diretorio onde estao os arquivos.
```

O primeiro comando define que, quando ler o BD:

- As imagens devem ser redimensionadas para 14×14 (o original é 28×28);
- Preto e branco devem ser invertidos, ficando fundo=branco e caracter=preto. O original é o contrário;
- As linhas e colunas completamente brancas em torno do caracter devem ser eliminadas (ajustaBbox=true).

O segundo comando (mnist.le) realmente faz a leitura do BD MNIST, que deve estar disponível no diretório especificado. Após a leitura, ficam disponíveis as seguintes estruturas dentro do objeto *mnist*:

```
int na; // Numero de amostras de treinamento (60000)
vector< Mat_<GRY> > AX; // Vetor de 60000 imagens de treinamento
                        // redimensionadas, com cores invertidas, e com BBox.
vector<int> AY; // Classificacoes das imagens de treinamento (0 a 9)
Mat_<FLT> ax; // Imagens de treinamento convertidos para FLT, cada imagem por linha.
Mat_<FLT> ay; // Classificacoes das imagens de treinamento convertidas para FLT (0 a 9)

int nq; // Numero de imagens testes (10000)
vector< Mat_<GRY> > QX; // Vetor de 10000 imagens de teste
                        // redimensionadas, com cores invertidas, e com BBox.
vector<int> QY; // Classificacoes das imagens de teste (0 a 9)
Mat_<FLT> qx; // Imagens de teste convertidos para FLT, cada imagem numa linha.
Mat_<FLT> qy; // Classificacoes das imagens de teste (0 a 9)
Mat_<FLT> qp; // Matriz ja alocado para colocar as classificacoes dos algoritmos de aprendizagem.
                // Matriz tem uma coluna e 10000 linhas.
```

- 1) *AX* e *QX* são vetores de imagens de treino e teste respectivamente (dependendo das opções de leitura, já com cores invertidas e ajustadas por bounding box (linhas/colunas brancas eliminadas).
- 2) *AY* e *QY* são vetor de inteiros de rótulos (números de 0 a 9, as classificações corretas) de treino e teste respectivamente.
- 3) *na* é o número de elementos de treino (*AX* e *AY*, 60000). *nq* é o número de elementos de teste (*QX* e *QY*, 10000).
- 4) *ax* e *qx* são matrizes tipo float (respectivamente com *na* e *nq* linhas) que tem que em cada linha o vetor de todos os pixels de uma imagem *AX* ou *QX* (convertidos para 0=preto e 1=branco). Estas são estruturas adequadas para alimentar as rotinas de aprendizagem do OpenCV.
- 5) *ay* e *qy* são matrizes tipo float (respectivamente com *na* e *nq* linhas) com uma única coluna. São cópias dos vetores *AY* e *QY*, convertidos de *int* para *float*. Contêm rótulos de 0 a 9.

Neste projeto, vamos usar somente os dados de treino *AX*, *AY*, *ax*, *ay* e *na*. Não vamos usar os dados de teste *QX*, *QY*, *qx*, *qy* e *np*.

Além disso, os seguintes métodos da classe *MNIST* fazem redimensionamento de imagem para o tamanho especificado na construção da classe (14×14) e ajustam bounding box eliminando as linhas e colunas brancas da borda da imagem. Estes métodos não invertem as cores.

```
Mat_<GRY> MNIST::bbox(Mat_<GRY> a); // procura primeiro pixel diferente de 255
Mat_<FLT> MNIST::bbox(Mat_<FLT> a); // procura primeiro pixel menor que 0.5
```

Você pode usar esses métodos para pré-processar a imagem extraída do quadro do vídeo. O método *bbox* para *GRY* considera que o pixel é branco somente se o seu valor for 255. O método *bbox* para *FLT* considera que o pixel é branco se for  $\geq 0.5$ .

Entre os diferentes métodos de aprendizagem descritos na apostila *classif-ead*, *FlaNN* é ao mesmo tempo muito rápido (0,5s para treinar o método de aprendizagem a partir de 60000 imagens e 0,3s para classificar 10000 imagens) e com boa acuracidade (3% de taxa de erro). A minha sugestão é que usem esse método. A taxa de erro de 3% não causa preocupação, pois os erros ocorrem geralmente ao tentar classificar dígitos escritos de forma ilegível. Se você escrever legivelmente os dígitos nas placas, a ocorrência de erros deve ser muito baixa.

Rede neural convolucional (CNN) simples “tipo LeNet” comete por volta de 0,7% de erro e demora 62s para treinar e 0,6s para fazer predição das 10000 imagens, num computador com GPU. Num computador sem GPU, o mesmo programa demora 940s para treinar e 2s para fazer predição. Portanto, poderia usar uma CNN nesta aplicação, bastando treinar previamente a rede e salvar o modelo. A única dificuldade é na comunicação de C++ com Python, que dá certo trabalho (mas é perfeitamente possível). Outra solução seria usar um pacote de CNN em C++, por exemplo, a biblioteca “TinyDnn” que está incluída dentro do Cekeikon e que implementa uma pequena biblioteca de rede neural convolucional.

O que o programa fase 5 deve fazer:

- 1) Lê o BD MNIST e faz treino do FlaNN.
- 2) Pega um quadro do vídeo *capturado.avi*. Se chegou ao fim do vídeo, termina o programa.
- 3) Localiza a placa no quadro. Se há placa no quadro, marca a sua localização no quadro.
- 4) Se a placa estiver suficientemente próxima da câmera:
  - 4a) Copia a imagem do dígito dentro da placa para uma nova imagem *a* (figura 1b, canto inferior direito).
  - 4b) Faz ajuste de brilho/contraste em *a*, de forma que os pixels de fundo fiquem brancos e os pixels do dígito fiquem pretos. Note que a imagem original extraída do vídeo é cinza clara no fundo e cinza escura no dígito.
  - 4c) Chama o método `b=mnist.bbox(a)` para eliminar linhas/colunas brancas da borda e redimensionar a imagem para 14×14 (figura 1b, canto superior direito).
  - 4d) Chama a função de predição de FlaNN, para que classifique a imagem de dígito *b*.
  - 4e) Insere a classificação do dígito no quadro do vídeo (figura 1b canto inferior direito).
- 5) Grava o quadro do vídeo no vídeo de saída *locarec.avi*.
- 6) Vai para linha (2).

[fim da aula 6]

### 3 Sexta fase do projeto

[Início da aula 7]

[Lição de casa única da aula 7] Faça o sistema cliente-servidor cliente6.cpp e servidor6.cpp, onde o carrinho será controlado pelo conteúdo das placas. O carrinho deve seguir automaticamente o caminho indicado pelos dígitos manuscritos nas placas. A forma de chamar os programas deve ser:

```
raspberrypi$ servidor6  
computador$ cliente6 192.168.0.110 [videosaida.avi]
```

Como neste ano (2022) a apresentação final do projeto será através do vídeo, pensando em quem não tem espaço em casa/apartamento, você só precisa implementar obrigatoriamente 3 comandos (convenção A): pare o carrinho, vire 180 graus à esquerda e vire 180 graus à direita. Pode mostrar no vídeo o carrinho indo e voltando num corredor, semelhante a:

[https://drive.google.com/file/d/1MGivOxp4-SA68Oxg8gN7NErh\\_Ezbxmcw/view?usp=sharing](https://drive.google.com/file/d/1MGivOxp4-SA68Oxg8gN7NErh_Ezbxmcw/view?usp=sharing)

Quando quiser parar o carrinho, pode apertar ESC no computador ou colocar à frente do carrinho uma placa com um dígito indicando “pare”.

Convenção A:

- 0, 1, 2 ou 3: Pare o carrinho.
- 4, 5 ou 6: Vire 180 graus à esquerda imediatamente.
- 7, 8 ou 9: Vire 180 graus à direita imediatamente.

Opcionalmente, pode implementar todos os comandos dos anos passados (convenção B). Veja exemplos em “vídeos de projetos dos anos 2018 e 2019”:

<http://www.lps.usp.br/hae/apostilaraspi/index.html>

Convenção B:

- 0 ou 1: Pare o carrinho.
- 2: Vire 180 graus à esquerda imediatamente.
- 3: Vire 180 graus à direita imediatamente.
- 4 ou 5: Passe por baixo da placa e continue em frente.
- 6 ou 7: Vire 90 graus à esquerda imediatamente.
- 8 ou 9: Vire 90 graus à direita imediatamente.

Nesta fase, pode ser que o seu programa (em desenvolvimento) termine sem que tenha desligado corretamente os dois motores. Neste caso, os motores vão continuar girando mesmo após o fim do seu programa. Para se prevenir contra este imprevisto, deixe compilado o programa abaixo no Raspberry, cujo único objetivo é desligar os motores:

```
//paramotor.cpp
//compila paramotor -w
#include <wiringPi.h>
int main () {
    wiringPiSetup () ;
    pinMode (0, OUTPUT) ;
    pinMode (1, OUTPUT) ;
    pinMode (2, OUTPUT) ;
    pinMode (3, OUTPUT) ;
    digitalWrite (0, LOW) ;
    digitalWrite (1, LOW) ;
    digitalWrite (2, LOW) ;
    digitalWrite (3, LOW) ;
}
```

## Máquina de estados finita:

Existem problemas que não podem ser resolvidos adequadamente usando a programação estruturada que usa estruturas: “sequência de comandos”, “while”, “for” e “if ... else ...”. A trajetória automática do nosso carrinho é um deles. O nosso problema pode ser descrita mais adequadamente usando máquina de estados finita.

Uma *máquina de estados finita* (do inglês Finite State Machine) ou *autômato finito* é um modelo matemático usado para representar programas de computadores ou circuitos lógicos. É uma máquina abstrata que deve estar num estado de um conjunto finito de estados. A máquina pode estar num único estado por vez e este estado é chamado de estado atual. Um estado armazena informações sobre o passado, isto é, ele reflete as mudanças desde o início do sistema até o momento presente. Uma transição indica uma mudança de estado e é descrita por uma condição que precisa ser realizada para que a transição ocorra. Uma ação é a descrição de uma atividade que deve ser realizada num determinado momento. Máquinas de estado finitas podem modelar um grande número de problemas, entre os quais a automação de design eletrônico, projeto de protocolo de comunicação, análise e outras aplicações de engenharia [[Wikipedia](#)].

Uma máquina de vender café pode servir como exemplo de máquina de estados finita. No estado inicial, a máquina fica esperando um cliente chegar e escolher a bebida apertando um botão: café curto, café longo, capuccino, chocolate quente, etc. Uma vez que o cliente escolhe a bebida, a máquina passa para o próximo estado onde informa ao cliente o preço da bebida e aguarda o cliente inserir o dinheiro. Se o cliente não inserir dinheiro suficiente durante os próximos, digamos 60 segundos, a máquina devolve o dinheiro ao cliente volta para o estado inicial. Se o cliente inseriu o dinheiro suficiente, a máquina passa para o próximo estado, onde devolve o troco se o cliente não tiver inserido a quantia exata. Depois, passa para outro estado, onde começa a preparar a bebida. E assim por diante, até que o cliente retire a bebida e a máquina volta ao estado inicial onde fica esperando a chegada do próximo cliente. Repare que a máquina passa por vários “estados” e o comportamento da máquina depende do seu estado atual.

A fase 6 deste projeto pode ser modelada como uma máquina de estados. Se você tentar resolver esta fase usando programação convencional (estruturada ou orientada a objetos), provavelmente a solução ficará “bagunçada”.

A figura 5 mostra um exemplo de máquina de estados.

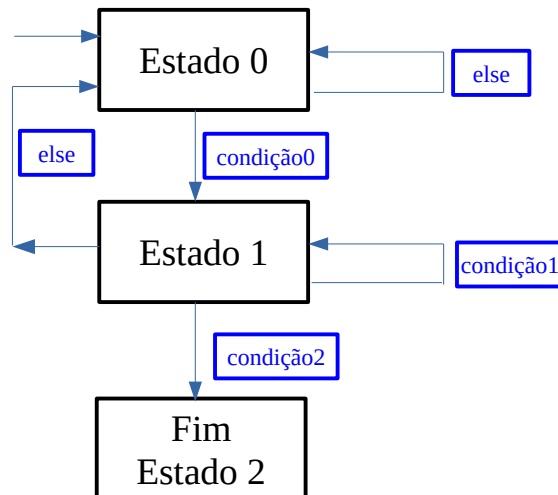


Figura 5: Um exemplo de máquina de estados.

A tradução desse modelo de máquina de estados para a linguagem C++ pode usar “labels” e comandos “goto”:

```
estado0:
    //Tarefas para executar no estado0
    if (condicao0) goto estado1;
    else goto estado0;
estado1:
    //Tarefas para executar no estado1
    if (condicao1) goto estado1;
    else if (condicao2) goto estado2;
    else goto estado0;
estado2:
    //Termina o programa no estado2
```

Provavelmente, os comandos “label” e “goto” foram considerados “proibidos” nos cursos introdutórios de programação, pois quebra o paradigma de programação estruturada. Porém, existem situações em que é a forma de deixar o seu programa mais organizado possível.

Outra possibilidade é usar uma variável para indicar o estado atual e comandos if/case dentro de um loop:

```
int estado=0;
do {
    if (estado==0) {
        //Tarefas para executar no estado0
        if (condicao0) estado=1;
    } else if (estado==1) {
        //Tarefas para executar no estado1
        if (condicao1) estado=1;
        else if (condicao2) estado=2;
        else estado=0;
    }
} while (estado!=2); //Termina o programa no estado2
```



O mais importante é modelar corretamente a máquina de estados, desenhando o diagrama de estados. A tradução para C/C++ é relativamente fácil. A fase 6 do projeto pode ser modelado como autômato finito desenhado na figura 6. Possivelmente, o diagrama deve ser adaptado para o tipo de motor disponível (plástico amarelo de alta rotação ou metálico de baixa rotação).

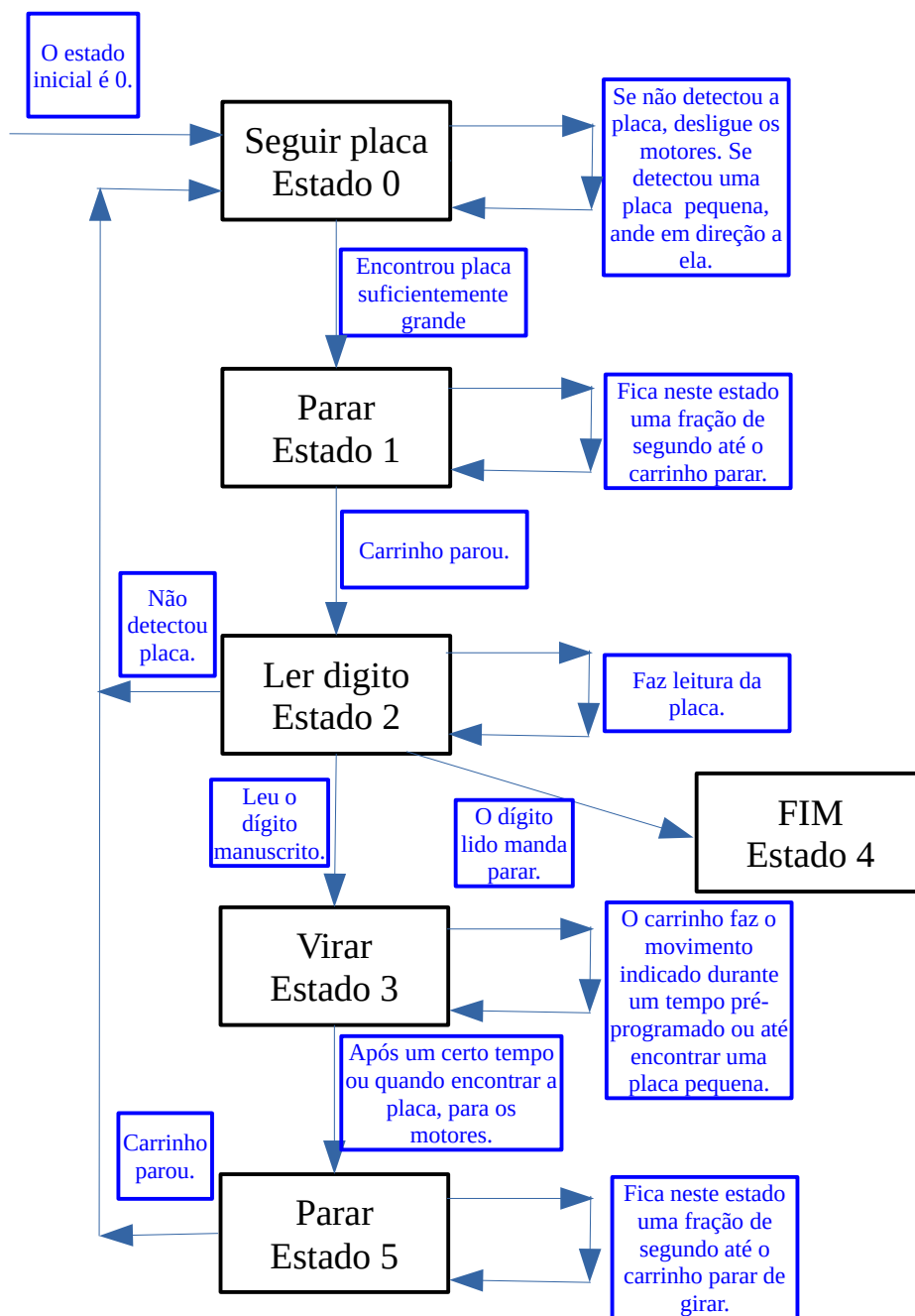


Figura 6: Um possível diagrama de estados (simplificado) da fase 6, para motor de alta rotação (motor amarelo de plástico).

**Estado 0.** No início do programa, o sistema está no estado 0. Este estado é semelhante ao projeto da fase 4. O sistema procura a placa. Se encontrar, o sistema faz o carrinho se movimentar em direção à placa, onde a direção do carrinho é controlada pela potência dos dois motores. Se não encontrar, o sistema desliga os motores do carrinho. O sistema fica neste estado até que enxergue uma placa suficientemente grande, onde seja possível ler o dígito.



Figura 7: Estado 0 é o estado inicial do sistema. O carrinho procura placa e vai em sua direção.

**Estado 1:** O sistema entra neste estado quando detectar uma placa suficientemente grande. Neste estado, o sistema manda parar os motores, para evitar uma colisão com a placa ou parede. Na figura 8, o sistema tentou ler o dígito manuscrito, mas errou a leitura, pois a placa estava borrada pelo movimento do carrinho (o carrinho ainda não tinha freado completamente). O sistema fica neste estado o tempo suficiente (por volta de um segundo) até que o carrinho pare.



Figura 8: O sistema entra no estado 1 quando encontra uma placa suficientemente grande. O carrinho começa a parar. Na figura, não é possível ler o dígito devido ao borrão de movimento.

**Estado 2:** O sistema entra no estado 2 um certo tempo após ter entrado no estado 1. Neste estado, o carrinho está completamente parado. A placa está bem visível com tamanho grande, sem borrão de movimento. Se não conseguir encontrar placa, o sistema volta para o estado 0. No estado 2, faz a leitura do dígito e entra no estado 3 carregando a informação lida. Se o dígito lido indicar “pare”, o programa termina (estado 4).

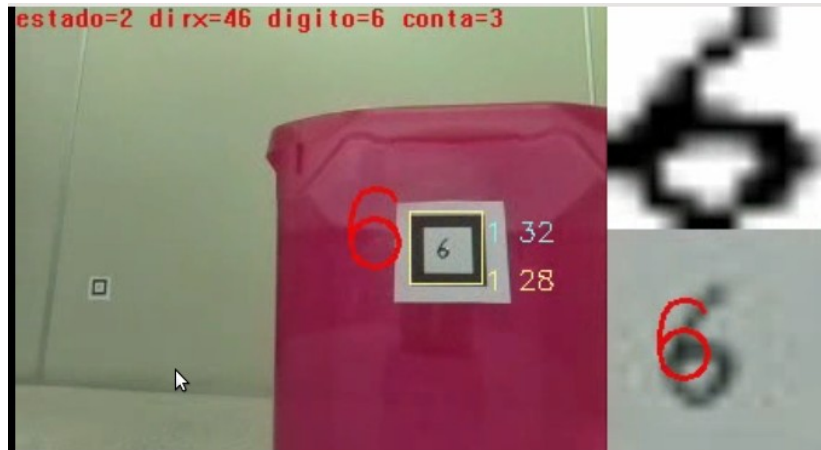


Figura 9: O sistema entra no estado 2 quando o carrinho parar completamente e enxergar a placa bem grande. Neste estado, faz a leitura do dígito manuscrito, com o carrinho completamente parado.

**Estado 3:** No estado 3, o carrinho faz o movimento indicado pelo dígito lido na fase 2 (por exemplo, virar 90 graus para esquerda). O carrinho sai deste estado após um tempo pré-programado ou quando enxergar a próxima placa a seguir. Vai para o estado 5.



Figura 10: No estado 3, o carrinho faz o movimento e volta para o estado 5.

**Estado 4:** Termina o programa.

**Estado 5:** Neste estado, espera um instante (uma fração de segundo) até que o movimento de virar do carrinho cesse completamente. Vai para o estado 0.

**Nota:** Aparentemente, o comando

```
cam >> image;
```

pode retornar a imagem antiga que estava armazenada na memória do Raspberry ou da câmera. O seguinte truque esvaziou a memória buffer e fez retornar a imagem vista no instante presente pela câmera:

```
cam.grab();
```

```
cam.grab();
```

```
cam >> image;
```