

# Aprendizado de Máquina

## [Início da aula 5]

**Resumo:** Na fase 5, desenvolveremos um programa que reconhece dígito manuscrito dentro das placas em vídeos, usando algum algoritmo de aprendizado de máquina. Na fase 6, desenvolveremos um sistema que dirige carrinho autonomamente, de acordo com os dígitos que escritos nas placas, utilizando o conceito de máquina de estados finita. Na fase 7, vamos juntar os controles manual e automático.

## 1 Introdução

A parte teórica de aprendizado 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, vimos na apostila [classif-ead.odt](#) o reconhecimento de dígitos manuscritos usando o conjunto MNIST, linguagem C++ e algoritmos de aprendizado clássicos (não deep learning). Além disso, as apostilas [densakeras-ead.odt](#) e [convkeras-ead](#) explicam como reconhecer os dígitos manuscritos em Python/Tensorflow/Keras usando redes neurais densa e convolucional.

## 2 Fase 5 do projeto

**[Lição de casa #1/1 da aula 5. Vale 10.]** 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 para acelerar o processamento (compile com o comando “compila fase5 -ocv -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`. O exemplo de vídeo de saída está em:

[\(GoogleDrive\)locarec.avi](#) ou [\(ServidorLocal\)locarec.avi](#)

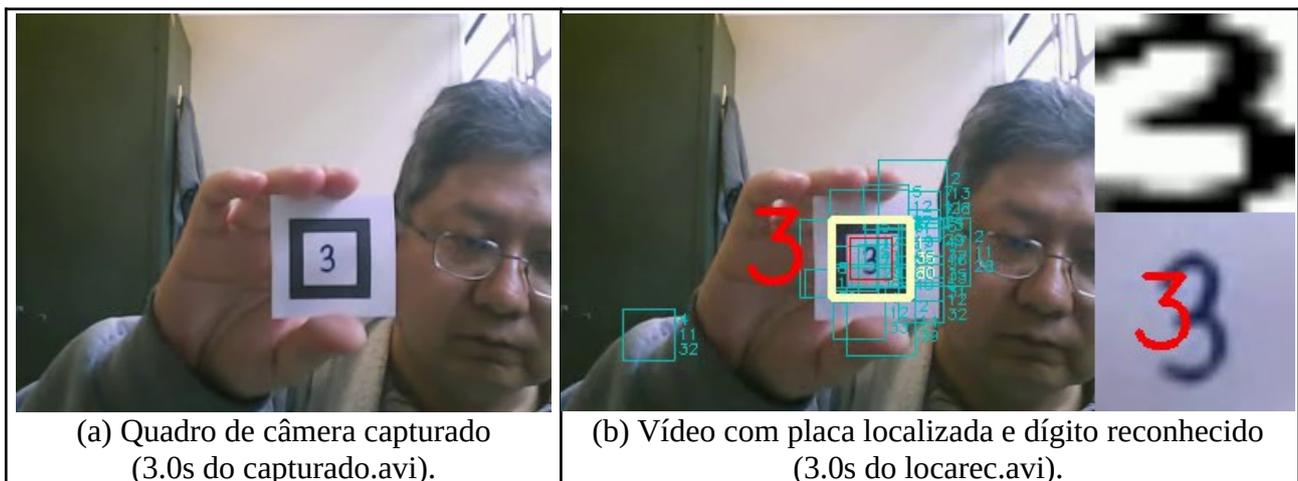


Figura 1: Localização da placa com reconhecimento do dígito manuscrito.

**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. Em projetos de Processamento de Imagens, é importante olhar as imagens intermediárias (como as imagens laterais da figura 1b). Isso permite descobrir facilmente os erros do projeto. Se você escrever um programa grande de Processamento de Imagens sem olhar as imagens intermediárias, provavelmente o seu programa vai cometer algum erro e você não vai ter como descobri-lo.

**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 o conjunto 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 *raspberry.hpp*, 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 MNIST:

- 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.

Nota: 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* (por quê?)

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 aprendizado de máquina 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 – mas vocês são livres para usarem o método que quiserem. 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. Escreva os dígitos com traços grossos – pode dar erro se os traços forem muito finos.

*Nota:* 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. Vocês precisam pesquisar como usar uma rede treinada em Tensorflow/Keras dentro de um programa C++.

*Nota:* Outra possibilidade seria fazer comunicação de C++ com Python. Uma outra solução seria usar um pacote de CNN para C++, por exemplo, a biblioteca “TinyDnn” que está incluída dentro de Cekeikon e que implementa uma pequena biblioteca de rede neural convolucional. Também existe a biblioteca Dlib.

O que o programa fase 5 deve fazer:

- 1) Lê o conjunto MNIST e faz treino do FlaNN. Alternativas: Usar outro algoritmo. Pré-treinar o modelo, deixá-lo gravado no HD e ler esse modelo no início do programa.
- 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 (no meu programa, placa na escala 0 ou 1):
  - 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 5]**

## [Início da aula 6]

### 3 Fase 6 do projeto

**[Lição de casa #1/2 da aula 6, vale 5]** Faça o sistema cliente-servidor cliente6.cpp e servidor6.cpp, onde o carrinho será controlado pelos dígitos manuscritos nas placas. O carrinho deve seguir automaticamente o caminho indicado nas placas. A forma de chamar os programas deve ser:

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

Implemente os comandos abaixo. Veja exemplos em “vídeos de projetos dos anos 2018 e 2019”:

<http://www.lps.usp.br/hae/apostilaraspi/index.html> e  
[vai e volta celular.mp4](#)

Comandos (2024):

- 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.

Nota: 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 (autômato finito):**

Existem problemas que não podem ser expressos adequadamente usando a programação estruturada que usa: “sequência de comandos”, “while”, “for” e “if ... else ...”. A trajetória automática do nosso carrinho é um deles. O nosso problema pode ser descrito mais facilmente 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 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 chegar o 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 e/ou orientada a objetos), provavelmente a solução ficará “bagunçada”.

A figura 5 mostra um exemplo de máquina de estados.

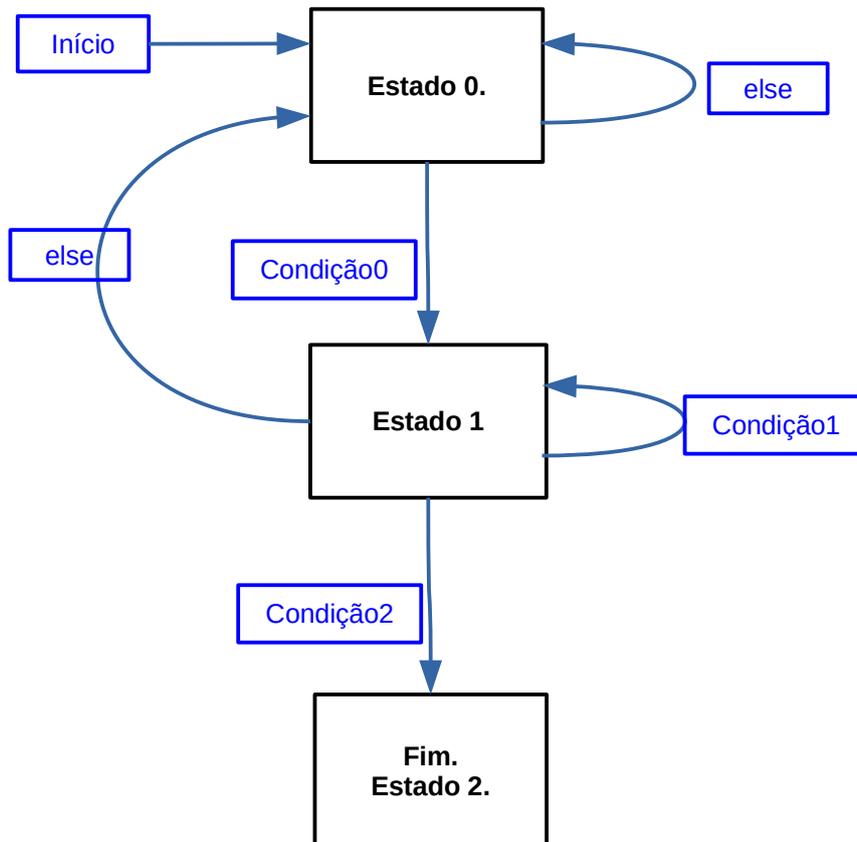


Figura 5: Um exemplo de diagrama de máquina de estados.

1) Para traduzir esse modelo para a linguagem C/C++ podemos usar “labels” com 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
```

*Nota:* Provavelmente, os comandos “label” e “goto” foram considerados “comandos 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.

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

```
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
```

É mais fácil raciocinar olhando o diagrama de estados do que o código C/C++. Assim, é importante desenhar correta e detalhadamente o diagrama de estados antes de traduzi-la para código C/C++. A tradução do diagrama para C/C++ é fácil.

*Nota:* É importante que um projeto de software esteja organizado. Se o projeto estiver bagunçado, é alta a probabilidade do projeto conter algum erro e será difícil encontrá-lo. Para organizar o projeto, existem paradigmas como programação estruturada e programação orientada a objetos, e práticas como indentar o código, comentar o código, etc. Não despreze essas práticas. A máquina de estados finita é uma das formas de organizar o seu projeto.

A fase 6 do projeto pode ser modelado como autômato finito desenhado na figura 6, que deve ser adaptado por cada grupo.

Coloquei toda a lógica da figura 6 no computador (cliente). O Raspberry recebe comandos simples do computador (como “siga a placa que está 30% à direita do centro da câmera”, “comece a virar para esquerda”, ou “pare de virar”, etc.) e os executa.

Para conseguir um controle fino dos movimentos de virar (90°, 180°, etc.), fiz o carrinho se movimentar durante um tempo pré-definido em frações de segundos e não durante uma certa quantidade de quadros. Por exemplo, digo “vire à esquerda durante 2,5s” e não vire “vire à esquerda durante 13 quadros”. Isto é importante, pois a quantidade de quadros por segundo varia de acordo com a velocidade da internet.

**Ajuda:** Como calcular intervalo de tempo em *Raspberry.hpp*:

```
double t1=timeSinceEpoch();
(...)
double t2=timeSinceEpoch();
double t=t2-t1; // t contem segundos decorridos entre t1 e t2
```

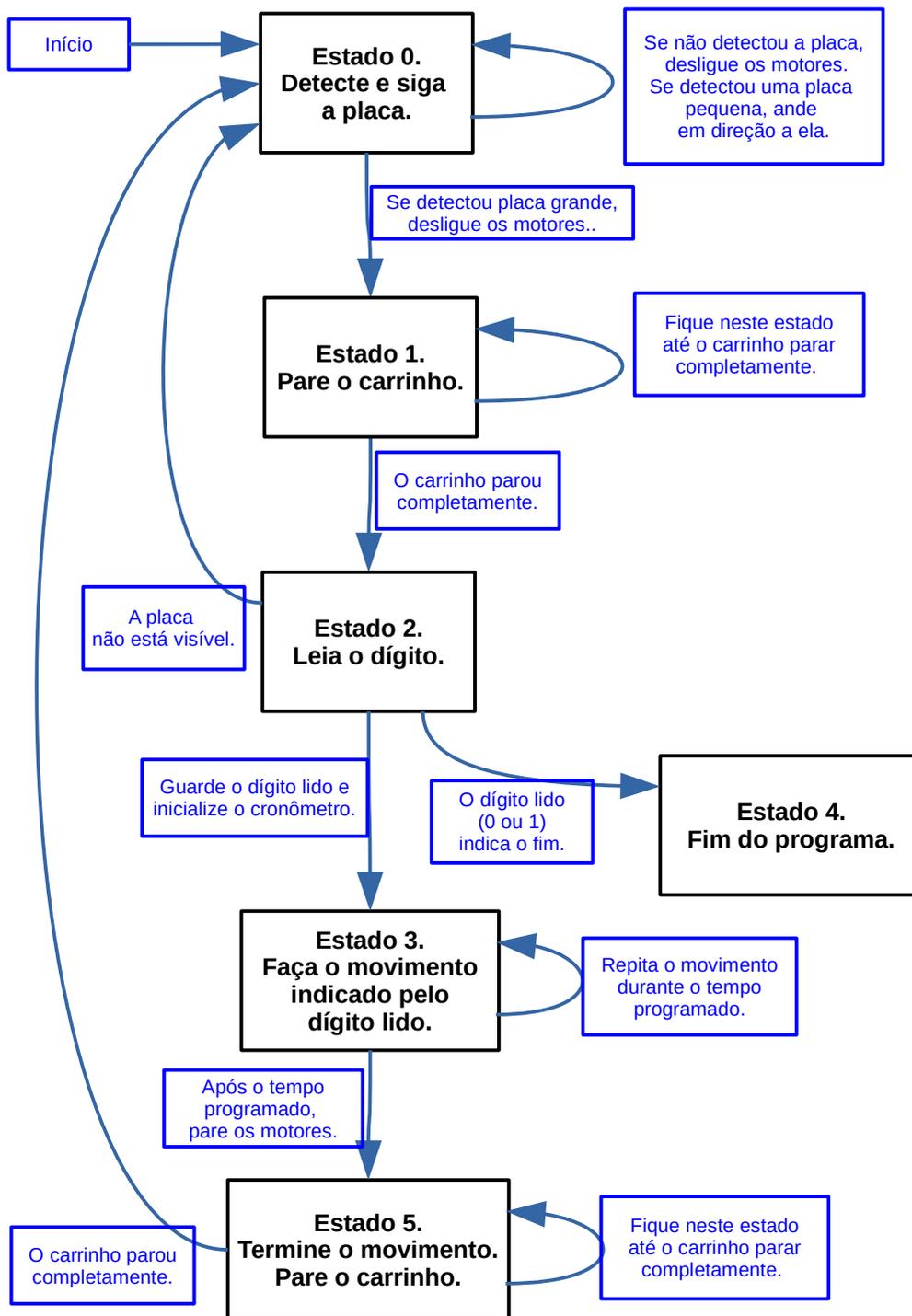


Figura 6: Um possível diagrama de estados (simplificado) da fase 6.

**Estado 0.** No início do programa, o sistema está no estado 0. Este estado é semelhante à fase 4: o sistema procura a placa. Se encontrar, o sistema faz o carrinho se movimentar em sua direção. Se não encontrar, o sistema desliga os motores do carrinho e espera. O sistema fica neste estado até enxergar uma placa suficientemente grande – quando encontrar, para os motores para evitar uma colisão.

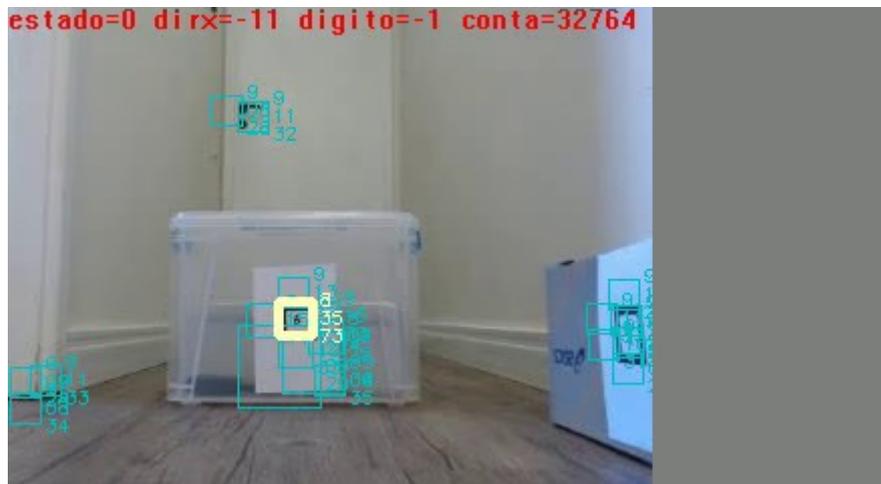


Figura 7: Estado 0 é o estado inicial do sistema. O carrinho procura placa e vai em sua direção. Dirx=-11 indica que a placa está à esquerda e que o carrinho deve virar ligeiramente à esquerda.

**Estado 1:** O sistema entra neste estado quando detectar uma placa suficientemente grande (escala 1 ou 0). O sistema para os motores para evitar uma colisão com a placa e para poder ler o dígito numa imagem nítida, sem borrão de movimento. O sistema fica neste estado durante um pequeno intervalo de tempo. Lembre-se de que, devido à inércia, é impossível o carrinho parar instantaneamente.

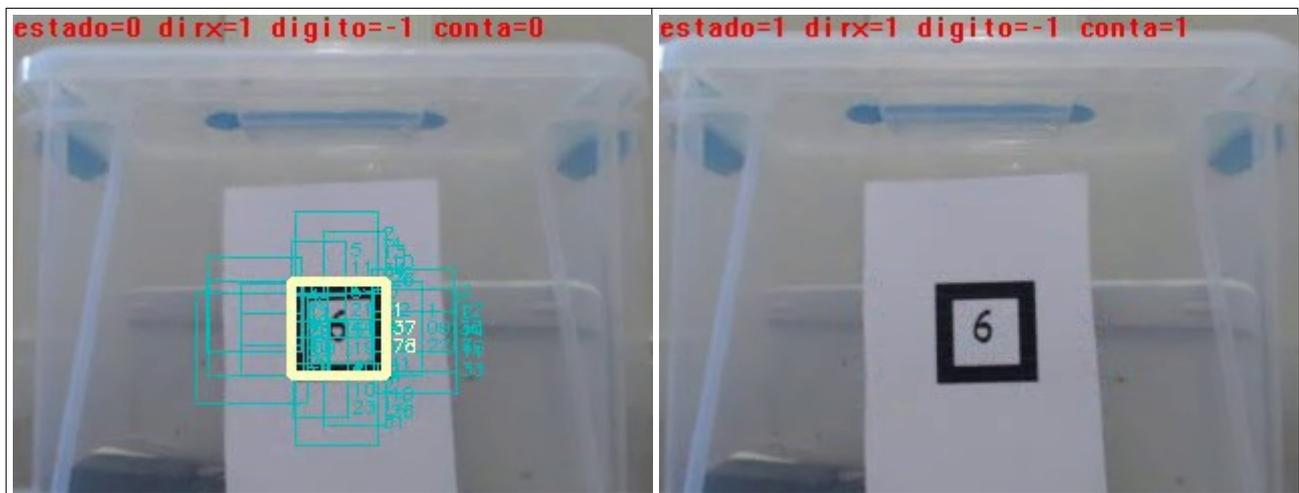


Figura 8: O sistema passa do estado 0 para 1 quando encontra uma placa suficientemente grande (escala 1 ou 0). O sistema desliga os motores durante um pequeno intervalo de tempo para o carrinho parar completamente e poder ler o dígito nítido sem borrão de movimento.

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

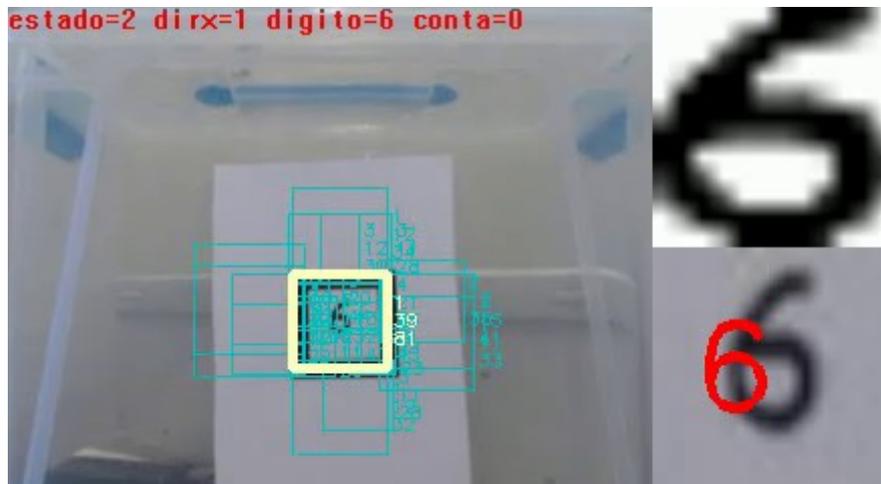


Figura 9: O sistema entra no estado 2 quando o carrinho parar completamente e enxergar a placa bem grande. Neste estado, faz a leitura do dígito manuscrito, com o carrinho completamente parado.

**Estado 3:** No estado 3, o carrinho faz o movimento indicado pelo dígito lido na fase 2 (por exemplo, virar 90 graus para esquerda). Enquanto faz o movimento, o Raspberry deve continuar enviando os quadros capturados ao computador. O carrinho sai deste estado após um tempo pré-programado e vai para o estado 5.



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

**Estado 4:** Entra neste estado quando ler o dígito que manda terminar o programa (dígito 0 ou 1).

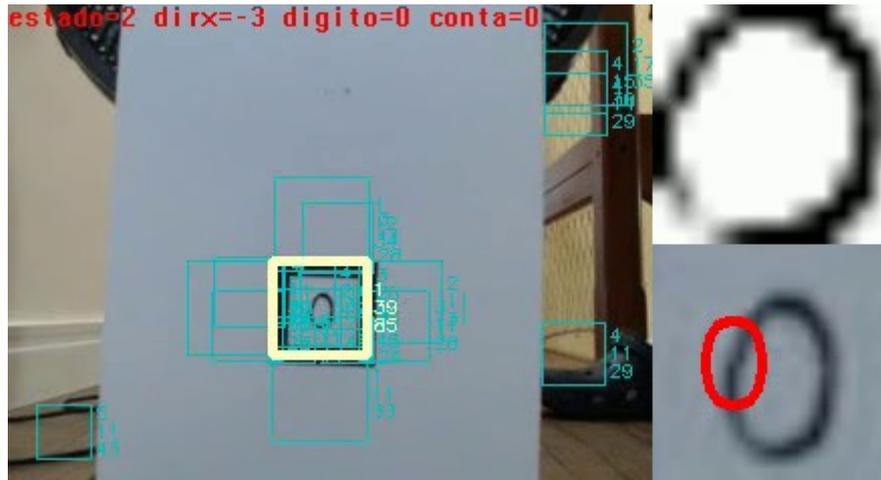
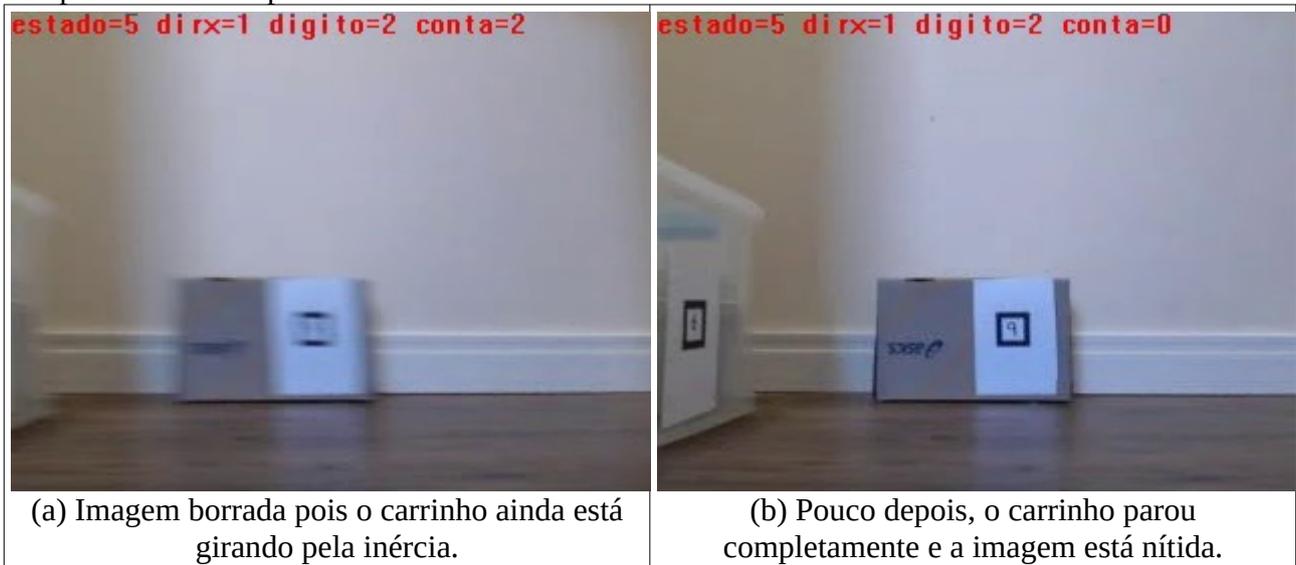


Figura 11: Estado 4 que termina o programa.

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



(a) Imagem borrada pois o carrinho ainda está girando pela inércia.

(b) Pouco depois, o carrinho parou completamente e a imagem está nítida.

Figura 12: Estado 5 que para a movimentação do estado 3. A figura (a) está borrada, pois o carrinho ainda está girando pela inércia, apesar dos motores terem parado. A figura (b) está nítida, pois o movimento de virar terminou.

O vídeo acima está em:

[fase6\\_raspberry.avi](#)

[fase6\\_celular.mp4](#)

**Nota:** Aparentemente, o comando

```
cam >> image;
```

pode retornar uma 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;
```

**Nota 2024:** Para poder usar aprendizado profundo, é necessário dispor de uma quantidade grande de imagens de treino. Assim, peço que vocês gravem vídeos que possam ser usados para treinar rede neural convolucional, a ser usado nos próximos anos. Cada grupo deve gravar vídeo da visão da câmera (sem teclado virtual e sem nenhuma anotação), de aproximadamente 1 minuto na fase 6 (controle automática). Coloquem os vídeos no formato AVI no Google Drive:

<https://drive.google.com/drive/folders/1Sc8kmIPNH4HIiz-6PfyhXVcr3RkaXqyug>

com nome, por exemplo:

Fase6\_FulanoDeTal\_CiclanoDeQual.avi

## 4 Fase 7 do projeto

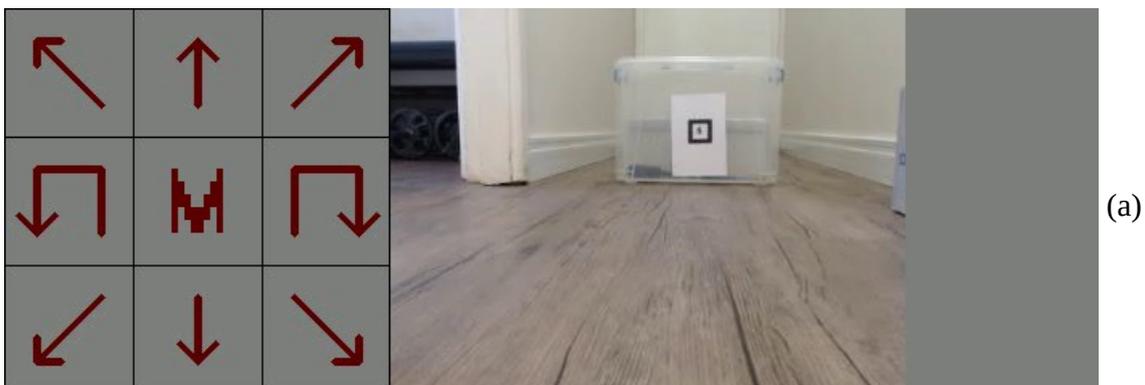
Na fase anterior (6), fizemos com que o carrinho seja controlado automaticamente pelos dígitos manuscritos no centro das placas. O problema é que, desta forma, não é possível controlar completamente o carrinho à distância. É necessário que o usuário coloque manualmente o carrinho na posição inicial, vá até o computador e digite os comandos no computador e Raspberry para que o carrinho comece executar a trajetória automática. Se houver algum problema no meio da trajetória, é necessário que o usuário mexa manualmente o carrinho novamente.

**[Lição de casa #2/2 da aula 6, vale 5]** Para que o usuário possa controlar o carrinho inteiramente à distância, nesta fase vamos misturar o controle manual (fase 2, apostila *rede*) com o controle automático (fase 6 desta apostila). O comando para esta fase deve ser:

```
raspberrypi$ servidor7  
computador$ cliente7 192.168.0.110 [videosaida.avi] [t/c]
```

O sistema deve começar no modo “manual”, mostrando uma tela semelhante à figura 13a. O usuário poderá controlar o carrinho manualmente para colocar o carrinho na posição inicial onde se inicia a trajetória automática. Neste momento, o usuário aperta o botão virtual “M/A” e o carrinho passa para o controle automático, seguindo as placas e obedecendo aos comandos escritos nas placas.

Quando chegar no final da trajetória automática, o controle volta para o modo manual. Além disso, a qualquer momento o usuário pode reassumir o controle manual do carrinho apertando a tecla virtual “M/A”. Este funcionamento pode ser resumido pelo diagrama de estados da figura 14.



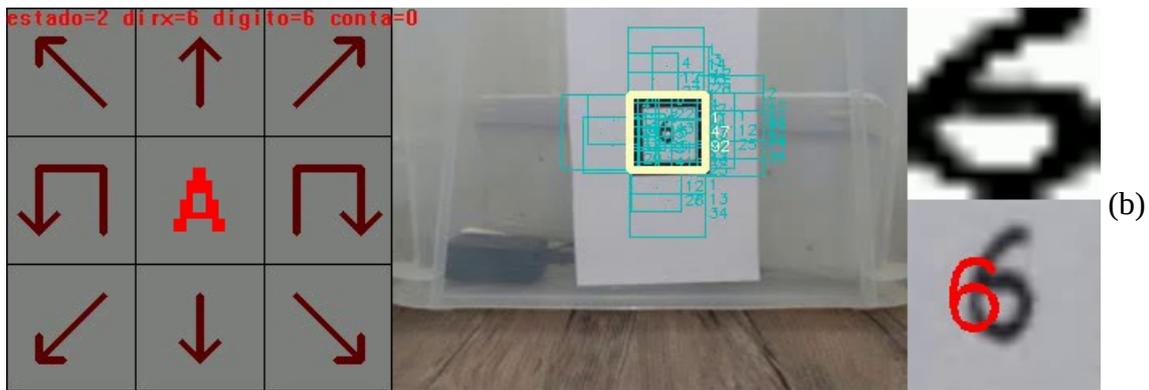


Figura 13: Modos manual e automático da fase 7.

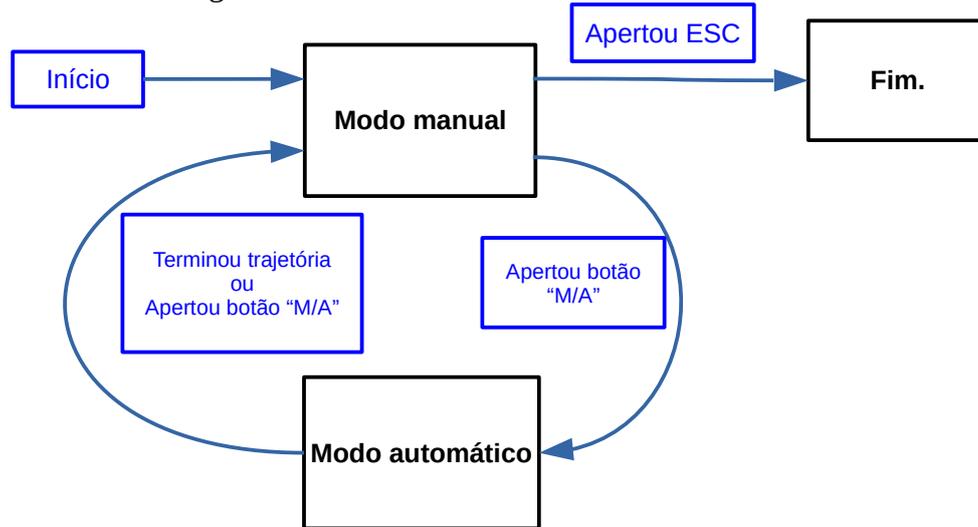


Figura 14: Diagrama de estados da fase 7.

O vídeo abaixo ilustra essa fase, passando do modo manual para automático e vice-versa:  
[fase7\\_raspberry.avi](#)

**Nota 2024:** Para poder usar aprendizado profundo, é necessário dispor de uma quantidade grande de imagens de treino. Assim, peço que vocês gravem vídeos que possam ser usados para treinar rede neural convolucional, a ser usado nos próximos anos. Cada grupo deve gravar vídeo da visão da câmera (sem teclado virtual, sem nenhuma anotação e sem imagens intermediárias), de aproximadamente 1 minuto nesta fase 7 (controle automática/manual). Coloquem os vídeos no formato AVI no Google Drive:

<https://drive.google.com/drive/folders/1Sc8kmIPNH4HIiz-6PfyhXVcr3RkaXqyug>

com nome, por exemplo:

Fase7\_FulanoDeTal\_CiclanoDeQual.avi

**[Fim da aula 6]**