Detecção de Objetos Super Mario 64

João Pedro Lopes Bazotti, Lucas dos Santos Bonorino

I. Introdução

Dentro do contexto de jogos virtuais uma coisa que é comum de vermos é a ideia de treinar uma inteligência artificial para jogar um jogo de complexidade alta. Porém, como podemos fazer uma dessas que seja genérica o suficiente?

Enquanto abordagens iniciais podem usar informações extraídas diretamente dos dados do jogo em tempo de execução(como posições exatas, estados de ação dos personagens, etc), uma outra abordagem possível é usar técnicas de visão computacional para identificar objetos de interesse em jogos, tais quais: inimigos, itens valiosos, o próprio player e assim por diante.

A. Descrição do Problema

Considerando essa abordagem, nosso problema passa a ser o seguinte: como conseguir captar informação de localização de objetos de interesse a partir de uma imagem. Além disso, como, podemos usar essa informação para conseguir definir informação de localização espacial no plano tridimensional?

II. Nossa solução

A primeira parte de nosso problema é comum em visão computacional, sendo comumente chamada de detecção de objetos. A segunda parte por outro lado pode ser vista como uma versão muito mais simplificada de um problema de estimar a distância da câmera para um objeto a partir de uma imagem e informações geométricas conhecidas do objeto.

A. Detecção de objetos

A primeira abordagem cogitada para realizar a detecção de objetos foi a de treinar um modelo de classificador cascata [5], porém o algoritmo obteve desempenho abaixo das expectativas para as imagens de nosso dataset, independente de número de estágios, profundidade máxima das árvores de decisão, tamanho do *ensemble* ou *features* utilizadas para descrição dos objetos. Acredita-se que isso se deva as características extremamente simplistas das cores do modelo do jogo, que, majoritariamente, não possuem uma textura, mas sim vértices com cores. Por conta disso, tantos as *features* de Haar, quanto as *features* LBP não captaram a essência dos modelos.

Tendo isso em vista, optamos por usar uma rede convolucinal para realizar a detecção de objetos, uma vez que ela prova-se mais fácil de treinar(classificador cascata requer um treino por classe, enquanto rede treina para todas as classes em um mesmo treino). Além disso, a própria rede aprende as *features* que devem ser procuradas, o quê pode reduzir o número tanto de detecções com falsos negativos quanto detecções com falsos positivos.

Para conseguir nossos dados anotados, foi utilizado um processo de automação. Primeiro, foi utilizado um emulador para gravar uma *playthrough* do jogo. Esse arquivo de gravação da emulação gravou todos os dados da emulação, capturando completamente o estado do jogo(movimentos e posições do jogador, movimentos e posições dos inimigos, etc). Após isso, modificamos uma versão descompilada do jogo para que todas as classes de interesse(no nosso caso, os diferentes inimigos) tivessem apenas uma cor e classes que não fossem de interesse ficassem com a cor preta. Tendo essas duas vesões diferentes do jogo em mãos, usamos a gravação do emulador para executar o jogo, tanto em sua versão normal, quanto em sua versão modificada, e gravamos um vídeo para cada uma.

Após a gravação, em uma etapa de pós processamento, nós sincronizamos o video ao máximo(ainda existem pequenas diferenças de frames), e o alimentamos para script que dividem o video em seus frames. Para o video do jogo inalterado, nós armazenamos suas imagens, criando um *dataset* com as imagens de cor. Já para o vídeo do jogo modificado, nós usamos as imagens para criar máscaras de segmentação com a maior precisão possível(sistema de iluminação do jogo ainda afetou os resultados, gerando alguns *outliers* no *dataset*).

A partir dessas máscaras de segemtação, nós fizemos múltiplos processos de binarização em cada imagem(um por classe), em que cada uma delas gerava uma imagem contendo apenas as máscaras de segmentação do objeto da classe de interesse.A partir das imagens binarizadas, nós aplicamos segmentação de contornos para extrair informação de posição e dimensão de cada objeto, e, a partir dos contornos encontrados, calculamos aproximações de *bounding boxes* dos objetos aos quais eles pertencem. Isso nos permitiu automatizar a anotação tanto das *bounding boxes* quanto de seus rótulos.

Nós então definimos uma rede com a arquitetura da *YOLOv8*[4], uma vez que a rede em si é consolidada, além de ser um detctor *one-stage*, o quê é substancialmente mais rápido que um detector *two-stage*. Como *backbone* da nossa rede, nós usamos uma versão reduzida da *MobileNetv3*[3], uma vez que precisavamos de uma rede rápida e eficiente. Ambas as escolhas foram feitas partindo do pressuposto que queremos plugar essa saída em uma IA para jogar o jogo, e, portanto, a velocidade de detecção deve ser em tempo real(ou minimamente próxima disso). O treinamento durou 156 épocas por questões de tempo, teve uma taxa de aprendizado de 0.005

Não houve tempo o suficiente para testar outras arquiteturas de redes, e, logo, não temos certeza se essa é uma configuração ótima.

Fig. 1. Imagem colorida

Fig. 2. Máscara de segmentação

B. Estimativa de distância

Considerando que estamos na situação de um jogo, a estimativa de distância torna-se um problema relativamente fácil. Uma das informações que precisamos para fazer essas estimativas é a razão entre a altura do personagem pivô(o Mario) com a altura dos outros personagens alvos(inimigos) presentes no jogo supondo que ambos estejam a uma mesma distância da câmera. Para obter esse números, fizemos uma inspeção visual do jogo e encontramos valores aproximados baseados nos movimentos do Mario, e em como ele se parece quando está próximo dos inimigos.

Para fazer esse cálculo, assumimos os seguintes pressupostos:

1) Nosso interesse é na distância relativa entre os inimigos e o Mario, portanto, a escala da dimensionalidade não importa(uma vez que pode ser ajustada)
2) Assumimos que o Mario está a sempre uma distância fixa da câmera
3) Assumimos que a perspectiva da visão é tal que a obliquidade da bounding box não será afetada, o quê significa que podemos alinhar as *bounding boxes* dos inimigos com a *bounding box* do Mario através dos centróides.
4) a alteração da distância entre um personagem possui uma relação de proporcionalidade com a alteração da altura de sua *bounding box*

Nesse caso, poderiamos fazer um corte transversal no plano

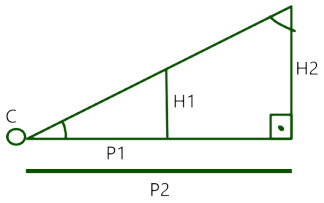

Fig. 3. Configuração das boudding boxes

de visão da câmera de forma a ficarmos na vista em 3

Nesse caso, as alturas H1 e H2 serão as alturas da *bounding box* esperada caso o inimigo estivesse a mesma distância da câmera que o Mario(calculada usando a altura da *bounding box do Mario* e as razões anotadas) e a altura real da *bounding box* encontrada ordenadas por altura(H1 sempre será menor que H2).

Aqui, C pode ser visto como o ponto antípoda da câmera, isso é, a câmera transladada e rotacionada de forma tal que o objeto mais distante fique como o objeto mais próximo. Para fazer isso, basta que definamos uma distância fixa da câmera que servirá como P1. Como queremos um triângulo retângulo para fazer essa projeção, podemos definir essa distância usando a lei dos senos e a altura da *Bounding box* como:

$$P1 = seno(A) * \frac{H1}{seno(B)} \tag{1}$$

Para o nosso caso, usamos os ângulos A=60º e B=30º.

Como queremos uma medida de distância entre dois objetos com relação um ao outro, a orientação a partir da qual fazemos esse cálculo não é de grande importância.

A partir disso, definir P2 torna-se um problema de semelhança de triângulos:

$$\frac{P1}{P2} = \frac{H1}{H2} \tag{2}$$

Resolvendo para P2:

$$P2 = \frac{P1}{H1} * H2 \tag{3}$$

Sendo assim, nós conseguimos calcular a distância de P2. A partir disso, basta calcular a distância entre ambos como sendo:

$$P = P2 - P1 \tag{4}$$

Nosso algoritmo pode ser sumarizado nos seguintes passos:

1) Usar a altura do Mario e a razão entre a altura dele e da classe do inimigo para calcular qual a altura esperada do inimigo supondo que eles estivessem a uma mesma distância da câmera.
2) Aplicar (1) para encontrar P1
3) Aplicar (3) para encontrar P2
4) Calcular P=P2-P1
5) Após isso, calculamos a distância utilizando as diferenças entre as coordenadas *X* esquerda e *Y* além da profundidade encontrada *P* para estimar a distância entre os objetos.

A partir desses dados nossa fórmula fica assim:

$$d = \sqrt{(X_{MarioEsq} - X_{InimigoEsq})^2 + (Y_{Marioinf} - Y_{Inimigoinf})^2 + P^2} \quad (5)$$

III. Resultados

Infelizmente, a rede não foi capaz de treinar o suficiente para conseguir realizar qualquer detecção de imagens no Dataset. Sua loss ficou muito alta, em aproximadamente 3. Não existem hipóteses vigentes sobre quais motivos levaram a rede a não conseguir desemepenho. Vamos mostrar, porém como ficaram as estimativas de distância de algumas imagens através de nosso *ground truth*.

Fig. 4. Exemplo 1

Em 4 podemos notar que a *Piranha Plant* está muito mais próxima do que sua distância indica, porém note que sua *bounding box* também está muito distorcida, não refletindo realmente suas dimensões. Espera-se que, supondo que sua bounding box estivesse mais póxima de suas dimensões reais, ela teria uma distância mais próxima.

Fig. 5. Exemplo 2

Em 5, podemos ver de maneira interessante como o cálculo funciona. Nesse caso o *King Bob-omb* possui duas *bounding boxes*: uma que representa adequadamente sua figura, e outra que representa um *outlier* na detecção. A *bounding box* do *outlier* é muito menor que o inimigo real, portanto, nosso método entende que é como se ele estivesse bem longe, considerando suas relações de proximidade originais.

Fig. 6. Exemplo 3

Em 6, podemos ver um dos erros de notação do Dataset. A pequena dissincronização entre os vídeos já foi o suficiente para gerar grandes diferenças na posição da *bounding box* do Mario.

Fig. 7. Enter Caption

Em 7, vemos um claro erro de cálculo. Um dos motivos prováveis para isso é que as esferas vistas, apesar de marcadas como *Bob-ombs* não são realmente *Bob-ombs* tratando-se de *outliers*,possívelmente provocados pelo fato de os inimigos terem partes do modelo com uso compartilhado com as bolas. Além disso, os erros de detecção(partes do objeto que ficaram de fora da *bounding box* também devem ter contribuído para isso.

IV. Conclusion

Apesar do nosso método parecer funcionar, ele ainda parece ser muito sensível a *outliers* nas *bounding boxes*, o quê significa que, para que o façamos ter bons resultados, seria necessário uma confiablidade extremamente alta para as *bounding boxes* estimadas.

Além disso, o treinamento para a identificação das *bounding boxes* provou-se mais desafiador do quê o esperado. E mesmo um jogo com gráficos e iluminação relativamente simples não pôde ser aprendido de maneira satsifatória por uma arquitetura já consolidada como a do YOLOv8[4].

Um ponto digno de nota na implementação desse trabalho foi a criação do *dataset*. Conseguimos, a partir de técnicas visão, processamento de imagens e simples modificações no jogo, criar um *dataset* com mais de trinta mil imagens em menos de meia hora. Isso mostra-se de grande utilidade para tarefas de anotação de máscaras de segmentação e de detecção de objetos.

Vê-se um grande potencial nessa técnica, se aplicada a jogos mais modernos com gráficos mais complexos para que possamos criar *datasets* volumosos de maneira simples e rápida, de maneira que eles possam ser generalizáveis para situações reais.

Appendix A
Razões encontradas

Tabela 1: Razões de alturas encontradas

Inimigo	**Razão de Altura**
Bob-Omb	0.85
Goomba	0.75
Chain Chhomp	3.2
King Bob-Omb	2.2
Piranha Plant	1.5
Twhomp	2.85
Whomp	3.5

REFERENCES

[1] Super mario 64 decompilation pc port, 2021.
[2] Andrew Houts, Bram Hagens, and Matt Kempster. Super mario 64 decompilation project, 2023.
[3] A. Howard et al. Searching for mobilenetv3. 2019.
[4] D. Reis, J. Kupec, J. Hong, and A. Daoudi. Real-time flying object detection with yolov8. 2023.
[5] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proc. CVPR'01*, pages 511–518, 2001.