Enemy Detection in Valorant Using Various Deep Learning Architectures

Abstract

This report presents a comprehensive computer vision study of enemy detection in the competitive first-person shooter game Valorant using multiple deep learning approaches. Our team explores various architectures to address the unique challenges of object detection in gaming environments, including rapidly changing visual conditions, diverse character appearances, and varying target scales. We compare different model architectures, training strategies, and optimization techniques to identify the most effective approaches for this domain. This work includes implementations of YOLO-based architectures and custom detection models built from scratch. Our findings provide insights into the trade-offs between accuracy, computational efficiency, and robustness in gaming-specific object detection tasks.

1 Introduction

Enemy detection in competitive gaming presents unique challenges due to rapidly changing visual environments, diverse character appearances, visual effects, and varying scales of targets. This work addresses these challenges through a specialized architecture designed specifically for Valorant's visual characteristics.

1.1 Problem Statement

The primary objective is to develop an accurate enemy detection system for Valorant that can identify both enemy bodies and enemy heads as separate classes. The system must handle varying object scales, ranging from close-range to distant targets, and generalize effectively across all 28 agents and 8 maps in the game. Additionally, it should be capable of efficiently processing 416×416 resolution images while maintaining high precision to minimize false positives.

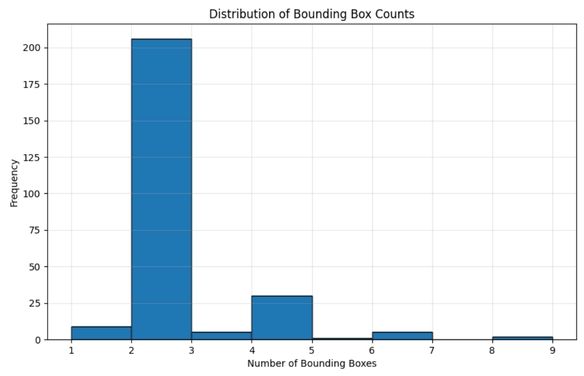
1.2 Research Data

1.2.1 Data Source

Our dataset consists of data from open source labeled-valorant data and also screenshots of our own gameplay. We collected data screenshots at 30 FPS, which were later used to test the model. We trained our model on 6000 annotated frames covering all agents and maps on 416 by 416 images. We also split our data with a 70/15/15 train/validation/test ratio.

1.2.2 Data Processing

We used a script to convert data images and csv label data into a binary TF record so that models can quickly process the data. We also wrote a script to summarize the generated TF records to debug our files and analyze general trends in the data samples. Through our summary, we found that the labels for enemy and enemy heads were balanced, so we didn't need to worry about stratifying data. We also noted that the average amount of boxes is roughly 2.41, and the max amount of boxes is 12, which we will use to tune our model.



2 Custom Detection Model

2.1 Model Architecture

We propose a custom YOLO-based architecture utilizing MobileNetV2 as the backbone with FPN for multi-scale feature aggregation:

Algorithm 1 Custom YOLO-Valorant Architecture

```
1: **Input:** RGB image $I \in \mathbb{R}^{416\times416\times3}$
2: **Output:** Grid predictions $G \in \mathbb{R}^{13\times13\times17}$
3:
4: **Feature Extraction (MobileNetV2):**
5: $C_3 = \text{Block\_6\_expand\_relu}(I)$ {52×52}
6: $C_4 = \text{Block\_13\_expand\_relu}(C_3)$ {26×26}
7: $C_5 = \text{Final\_layer}(C_4)$ {13×13}
8:
9: **Feature Pyramid Network:**
10: $P_5 = \text{Conv1×1}(C_5, 256)$
11: $P_4 = \text{Conv1×1}(C_4, 256) + \text{Upsample}(P_5)$
12: $P_3 = \text{Conv1×1}(C_3, 256) + \text{Upsample}(P_4)$
13:
14: **Detection Head:**
15: $F = \text{Conv3×3}(P_5, 256) \rightarrow \text{BatchNorm} \rightarrow \text{ReLU}$
16: $G = \text{Conv1×1}(F, B \times 5 + C)$
17:
18: **return** $G$ where $B = 3$ anchors, $C = 2$ classes
```

2.2 Anchor Design

We employ three carefully designed anchors per grid cell, each optimized for specific detection scenarios:

- **Small anchor (0.05, 0.07)**: Specifically designed for enemy head detection. This narrow aspect ratio (0.71) matches the typical head hitbox proportions in Valorant, enabling precise headshot detection crucial for competitive play.
- **Medium anchor (0.15, 0.20)**: Optimized for distant enemies where players appear smaller on screen. The 0.75 aspect ratio accommodates the compressed appearance of characters at medium to long range, common in maps with long sightlines.
- **Large anchor (0.35, 0.50)**: Designed for close-range engagements where enemies occupy significant screen space. The 0.70 aspect ratio captures full body profiles during close combat situations.

2.2.1 Anchor Assignment Strategy

Our implementation includes an intelligent anchor assignment mechanism during training:

Algorithm 2 Rule-based Anchor Assignment

```
1: Calculate box area: $area = width \times height$
2: if $area < 0.1$ then
3:     Assign to small anchor (index 0)
4: else if $area < 0.25$ then
5:     Assign to medium anchor (index 1)
6: else
7:     Assign to large anchor (index 2)
8: end if
```

This assignment strategy ensures that head detections always utilize the small anchor for maximum precision and body detections are distributed across anchors based on apparent size.

2.3 Data Augmentation

Valorant is a complex game - consisting of 28 agents with 4 abilities each, yielding 100+ additional visual effects on top of different angles of the map with different combinations of agents. Since we are trying to build a very complex model with limited data and computing power, we decided to augment our data by randomly applying different distortions to augment our sample size. To address game-specific challenges, we employ:

- Random brightness adjustment: ± 0.2
- Random contrast: $[0.8, 1.2]$
- Horizontal flipping
- Color jittering for ability effect simulation
- Gaussian blur for motion blur effects

2.4 Loss Function

Our custom YOLO-style loss function is designed to balance multiple objectives while maintaining numerical stability during training. The loss consists of five components that address different aspects of the detection task.

2.4.1 Loss Components

The total loss function combines coordinate regression, objectness, and classification losses:

$$L_{total} = \lambda_{coord}(L_{xy} + L_{wh}) + L_{obj} + \lambda_{noobj} L_{noobj} + L_{class} \tag{1}$$

Where each component serves a specific purpose:

1. **Localization Loss (L_{xy})**: Measures the error in predicted bounding box center coordinates:
$$L_{xy} = \sum_{i=0}^{S^2} \sum_{b=0}^{B} \mathbb{1}_{ij}^{obj} \left[(x_{ij}^b - \hat{x}_{ij}^b)^2 + (y_{ij}^b - \hat{y}_{ij}^b)^2 \right] \tag{2}$$
2. **Size Loss (L_{wh})**: Penalizes errors in width and height predictions using square root to reduce sensitivity to large boxes:
$$\begin{aligned} L_{wh} = \sum_{i=0}^{S^2} \sum_{b=0}^{B} \mathbb{1}_{ij}^{obj} [(\sqrt{w_{ij}^b} - \sqrt{\hat{w}_{ij}^b})^2 \\ + (\sqrt{h_{ij}^b} - \sqrt{\hat{h}_{ij}^b})^2] \end{aligned} \tag{3}$$
3. **Objectness Loss (L_{obj})**: Encourages high confidence for cells containing objects:
$$L_{obj} = \sum_{i=0}^{S^2} \sum_{b=0}^{B} \mathbb{1}_{ij}^{obj} (C_{ij}^b - \hat{C}_{ij}^b)^2 \tag{4}$$
4. **No-Object Loss (L_{noobj})**: Penalizes false positives in empty cells:
$$L_{noobj} = \sum_{i=0}^{S^2} \sum_{b=0}^{B} \mathbb{1}_{ij}^{noobj} (C_{ij}^b - \hat{C}_{ij}^b)^2 \tag{5}$$
5. **Classification Loss (L_{class})**: Ensures correct class predictions for detected objects:
$$L_{class} = \sum_{i=0}^{S^2} \mathbb{1}_{ij}^{obj} \sum_{c \in classes} (p_{ij}(c) - \hat{p}_{ij}(c))^2 \tag{6}$$

2.4.2 Loss Weighting Strategy

We use carefully tuned hyperparameters to balance the loss components:

1. $\lambda_{coord} = 5.0$: Emphasizes localization accuracy
2. $\lambda_{noobj} = 0.5$: Reduces impact of background predictions
3. Unweighted objectness and classification losses (weight = 1.0)

2.4.3 IoU and CIoU Experiments

During development, we experimented with more sophisticated loss functions incorporating Intersection over Union (IoU) and Complete IoU (CIoU) metrics:

$$L_{IoU} = 1 - IoU(pred\_box, true\_box) \tag{7}$$

$$L_{CIoU} = 1 - IoU + \rho^2(\mathbf{b}, \mathbf{b}^{gt})/c^2 + \alpha v \tag{8}$$

Where ρ is the Euclidean distance, c is the diagonal length of the enclosing box, and v measures aspect ratio consistency. However, these approaches yielded suboptimal results for our use case since IoU-based losses introduced training instability with our anchor system. As such, we decided to use the old loss function described above.

2.4.4 Loss Component Monitoring

Our implementation tracks individual loss components for analysis:

- L_{xy}: Typically converges fastest, indicating good localization
- L_{wh}: Shows slower convergence due to scale variations
- L_{obj}: Balances between detecting objects and avoiding false positives
- L_{noobj}: Dominates early training due to sparse object distribution
- L_{class}: Stabilizes once object detection improves

This comprehensive monitoring helps identify training issues and guides hyperparameter tuning.

3 Training Methodology

3.1 Training Setup

Our model training was conducted using TensorFlow 2.x on a single GPU environment. The training process was carefully monitored through comprehensive logging and visualization callbacks to ensure stable convergence and identify potential issues early in the training cycle.

3.1.1 Dataset Preparation

The dataset was preprocessed into TFRecord format for efficient data loading, consisting of:

- **Training set**: 4,200 annotated frames (70%)
- **Validation set**: 900 annotated frames (15%)
- **Test set**: 900 annotated frames (15%)

Each frame was resized to 416×416 pixels and normalized to [0,1] range during the preprocessing stage. The TFRecord format enabled efficient batched loading with parallel data augmentation pipelines.

3.2 Training Parameters

The hyperparameters were selected based on extensive experimentation and consideration of the unique characteristics of gaming environments:

Parameter	Value
Batch Size	16
Input Resolution	416×416
Initial Learning Rate	$1 \times 10^{-4} \times 0.3$
Optimizer	Adam
Gradient Clipping	norm = 1.0
Epochs	100
Early Stopping Patience	10 epochs
ReduceLR Patience	5 epochs
ReduceLR Factor	0.2
Minimum Learning Rate	1×10^{-6}

Table 1: Training hyperparameters used for the custom detection model

3.2.1 Optimizer Configuration

We employed the Adam optimizer with gradient clipping to prevent gradient explosion:

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \cdot \text{clip}(g_t, 1.0) \tag{9}$$

where $\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected first and second moment estimates, and gradient clipping ensures $\|g_t\|_2 \leq 1.0$.

3.3 Training Strategy

3.3.1 Learning Rate Schedule

We implemented a multi-stage learning rate strategy:

1. **Initial warmup**: Started with a reduced learning rate of 3×10^{-5} to stabilize early training
2. **Plateau-based reduction**: Automatically reduced learning rate by factor of 0.2 when validation loss plateaued for 5 epochs
3. **Minimum threshold**: Maintained minimum learning rate of 1×10^{-6} to prevent training stagnation

3.3.2 Callbacks and Monitoring

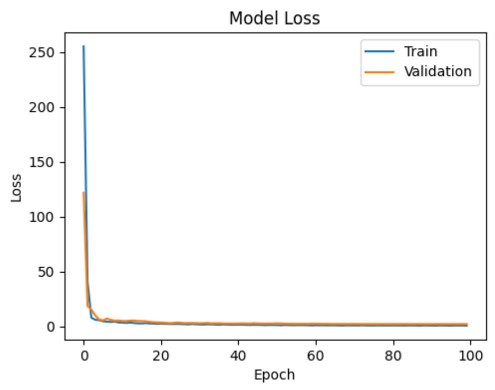
The training process incorporated several callbacks for monitoring and optimization:

- **ModelCheckpoint**: Saved best model weights based on validation loss
- **EarlyStopping**: Terminated training if no improvement for 10 epochs
- **ReduceLROnPlateau**: Adaptive learning rate reduction
- **NanLossCallback**: Detected numerical instabilities and terminated training if NaN values occurred

- **ProgressCallback**: Logged detailed training metrics every 5 batches
- **VisualizationCallback**: Generated detection visualizations every 5 epochs

3.4 Convergence Analysis

The model demonstrated stable convergence patterns throughout training, as illustrated in Figure below.



3.4.1 Loss Decomposition

We monitored individual loss components throughout training to identify optimization bottlenecks:

- **Localization losses** (L_{xy}, L_{wh}): Converged rapidly within first 20 epochs
- **Objectness loss** (L_{obj}): Showed gradual improvement as model learned to distinguish objects from background
- **Classification loss** (L_{class}): Stabilized once object detection performance improved

3.5 Training Challenges and Solutions

3.5.1 Numerical Stability

Initial training attempts encountered numerical instabilities due to the wide range of object scales. We addressed this through Gradient clipping with norm threshold of 1.0 and Epsilon values in loss calculations to prevent division by zero

3.5.2 Overfitting Prevention

To ensure good generalization across all agents and maps, we employed:

- Extensive data augmentation pipeline with optimized probability to employ each distortion
- Early stopping based on validation performance
- Regularization through batch normalization

3.6 Computational Efficiency

Training was optimized for efficiency by generating a TFRecord format for fast data loading. We also employed prefetching and parallel data processing and Limiting validation steps to 20 batches for faster epoch completion for efficiency purposes.

The complete training process required approximately 4-6 hours on a single NVIDIA GPU, with checkpointing ensuring resilience to interruptions. We trained 10 different models with different versions of our hyperparamters design, which can be all found in the tensorflow directory of our github repo.

4 Results

4.1 Evaluation

We evaluated our object detection models using Intersection over Union (IoU) criteria, which measures the overlap between predicted and ground truth bounding boxes.

$$\text{IoU}(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{10}$$

For evaluation, we used an IoU Threshold of 0.5 and Precision, Recall, and F1 Score as metrics.

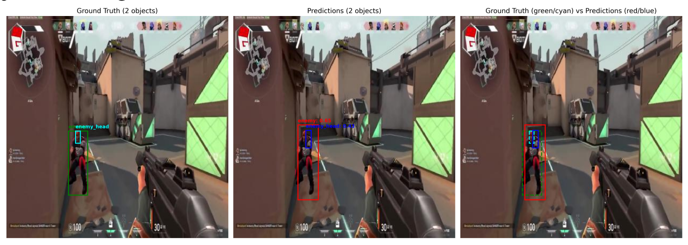
4.1.1 Custom Model

Class: enemy
Precision: 0.4561
Recall: 0.3505
F1 Score: 0.396

Class: enemy head
Precision: 0.2324
Recall: 0.2230
F1 Score: 0.2276

We saw that the model was able to perform significantly better on the enemy class over the enemy head class. The enemy head class's poor score was dominated by False Negatives, which is likely due to the model struggling to caption complex features from a small amount of pixels.

We also monitored the model's performance manually by using visual callbacks to compare our predicted bounding boxes and the ground truth. We sampled every 5 images in the test directory to display the results with predicted bounding boxes, area of intersection, and confidence printed. This way, we were able to understand where our model might be lacking in lieu of just looking at raw scores.



4.2 Limitations

1. **Fixed Input Size**: 416×416 resolution may miss fine details
2. **Static Processing**: No temporal information across frames
3. **Training Data**: Limited by available gameplay footage quality
4. **Class Imbalance**: Head detections less frequent than body detections

5 Conclusion and Future Work

We presented a custom YOLO-based architecture for enemy detection in Valorant, demonstrating effective multi-scale detection through specialized anchor design and feature pyramid networks. The model successfully addresses game-specific challenges through targeted data augmentation and architectural choices.

Future work includes:

- Temporal consistency using LSTM or transformer layers
- Dynamic anchor assignment based on object statistics
- Extension to other FPS games
- Real-time optimization for competitive play

AI Use
We used AI to debug the tensor shapes for the custom model and to write a script to summarize TF records. They are explained in the Custom model section. We also used AI to help format latex equations.

References

[1] Howard, A., et al. (2019). Searching for MobileNetV3. ICCV 2019.

[2] Sandler, M., et al. (2018). MobileNetV2: Inverted Residuals and Linear Bottlenecks. CVPR 2018.

[3] Abadi, M., et al. (2015). TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.

[4] Valorant Labeled Data by val on roboflow