Deep Learning for Image Analysis: INM705

Anndischeh Mostafavi- 220051723

Course 2024-2025

Link
WP&B Project Link
Dataset Link

1 Introduction

This coursework addresses the problem of object detection, a crucial task in computer vision with applications spanning autonomous driving, robotics, and security systems. We tackle this challenge by implementing and enhancing the **YOLOv1** (You Only Look Once) object detection model [1]. YOLOv1, despite its age, provides a valuable foundation for understanding real-time object detection principles and offers opportunities for improvement with modern techniques. We utilize the COCO (Common Objects in Context) dataset , a large-scale, open-source dataset containing labeled images of various objects in complex scenes, allowing us to train and evaluate our model effectively. This project builds upon several open-source implementations of YOLOv1, particularly those focused on PyTorch, adapting and expanding them to incorporate best practices and improve performance [2].
My work is informed by the evolution of object detection models, starting with region-based methods like R-CNN [3] and Fast R-CNN [4], which, while accurate, suffered from slow inference speeds. YOLOv1 revolutionized the field by framing object detection as a regression problem, enabling significantly faster processing. We acknowledge these earlier models, noting their relevance as stepping stones to the one-stage approach adopted by YOLO and its successors. Furthermore, we've drawn inspiration from more recent advancements, such as the use of data augmentation techniques like affine transformations and the adoption of sophisticated loss functions, to optimize the performance of our YOLOv1 implementation within the constraints of this coursework.

2 Methodology

This section details the YOLOv1 model architecture and the modifications implemented for this coursework. We also describe the loss function used for training, providing the mathematical foundation for our approach.

2.1 YOLOv1 Architecture

YOLOv1 (You Only Look Once) is a single-stage object detection model that reframes object detection as a regression problem. Instead of relying on region proposals like earlier two-stage detectors (e.g.,RCNN, Fast RCNN), YOLOv1 divides the image into an S x S grid. Each grid cell predicts B bounding boxes, a confidence score for each box (representing the probability of an object being present), and a class probability distribution. This streamlined approach significantly improves inference speed, making it suitable for real-time applications.

2.1.1 Core Network Structure

The YOLOv1 architecture is inspired by GoogLeNet , but with fewer layers [7]. It consists of a series of convolutional layers followed by fully connected layers. The convolutional layers extract features from the input image, while the fully connected layers predict the bounding box coordinates, object confidence scores, and class probabilities. Our specific implementation utilizes the architecture defined in _hyperparameters.py_ within the code. This architecture, denoted as _A_ in our code (and detailed below), comprises a sequence of convolutional blocks, max-pooling layers, and repeating convolutional sequences. The network architecture _A_ can be represented as follows:

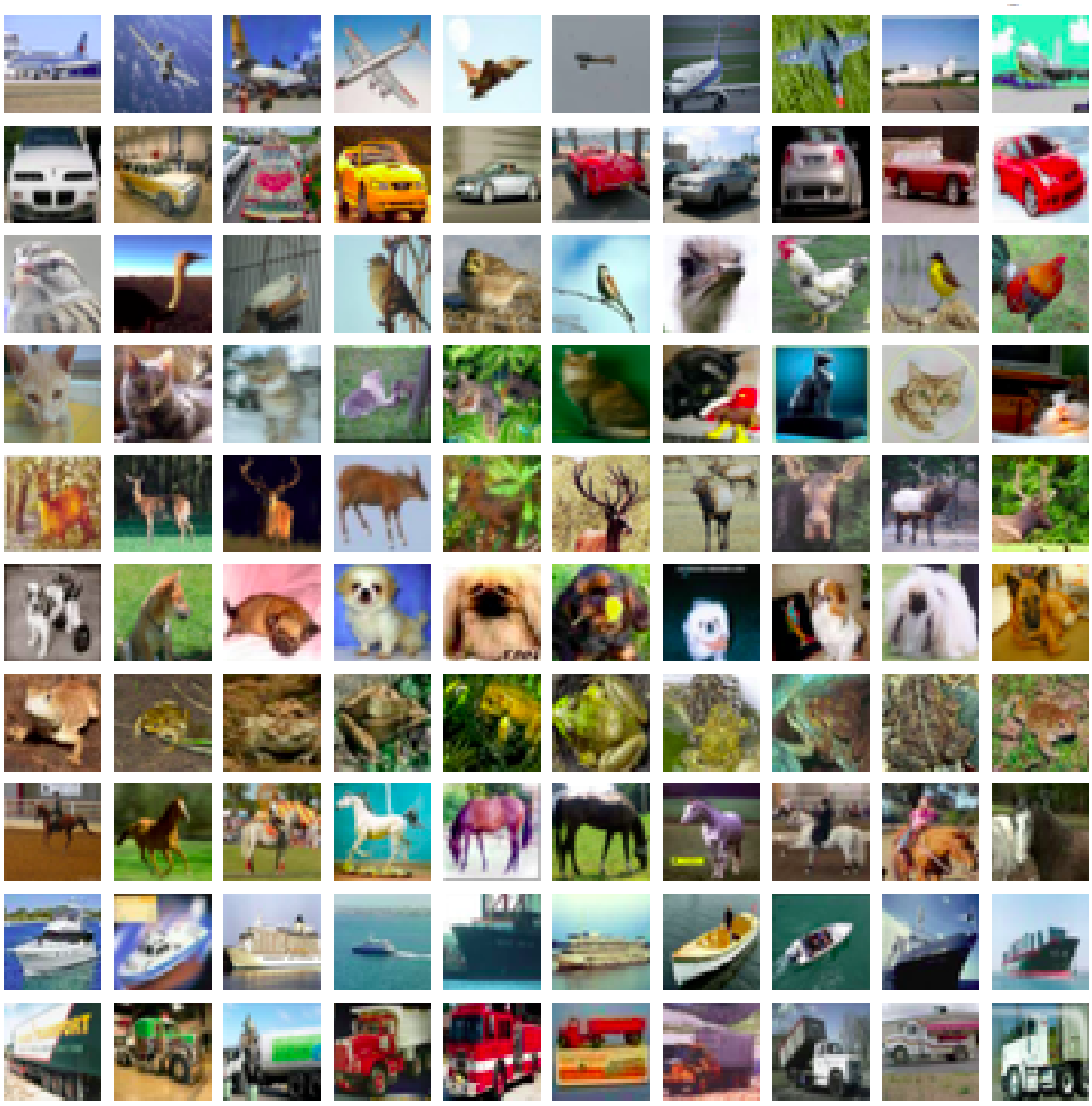


Fig. 1: The MS COCO dataset is a large-scale object detection dataset[5].

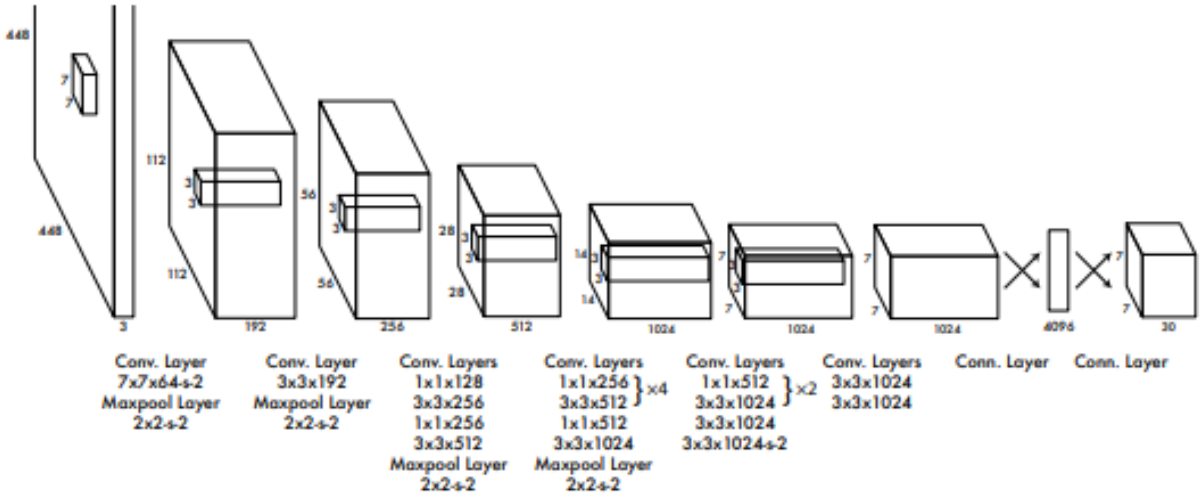


Fig. 2: Yolo network has 24 convolutional layers followed by 2 fully connected layers[6].

$$
A = \left\{ \begin{array}{l}
\text{Conv}(7 \times 7, 64, S = 2, P = 3), \\
\text{MaxPool}(S = 2), \\
\text{Conv}(3 \times 3, 192, S = 1, P = 1), \\
\text{MaxPool}(S = 2), \\
\text{Conv}(1 \times 1, 128, S = 1, P = 0), \\
\text{Conv}(3 \times 3, 256, S = 1, P = 1), \\
\text{Conv}(1 \times 1, 256, S = 1, P = 0), \\
\text{Conv}(3 \times 3, 512, S = 1, P = 1), \\
\text{MaxPool}(S = 2), \\
[\text{Conv}(1 \times 1, 256, S = 1, P = 0),\ \text{Conv}(3 \times 3, 512, S = 1, P = 1)] \times 4, \\
\text{Conv}(1 \times 1, 512, S = 1, P = 0), \\
\text{Conv}(3 \times 3, 1024, S = 1, P = 1), \\
\text{MaxPool}(S = 2), \\
[\text{Conv}(1 \times 1, 512, S = 1, P = 0),\ \text{Conv}(3 \times 3, 1024, S = 1, P = 1)] \times 2, \\
\text{Conv}(3 \times 3, 1024, S = 1, P = 1), \\
\text{Conv}(3 \times 3, 1024, S = 2, P = 1), \\
[\text{Conv}(3 \times 3, 1024, S = 1, P = 1)] \times 2
\end{array} \right\}
$$

Where:

- ▷ "Conv"$(k \times k, c, S = s, P = p)$ represents a convolutional layer with kernel size $k \times k$, c output channels, stride s, and padding p.
- ▷ "MaxPool"$(S = s)$ represents a max-pooling layer with stride s.
- ▷ [Layer Sequence] $\times N$ represents a sequence of layers repeated N times.

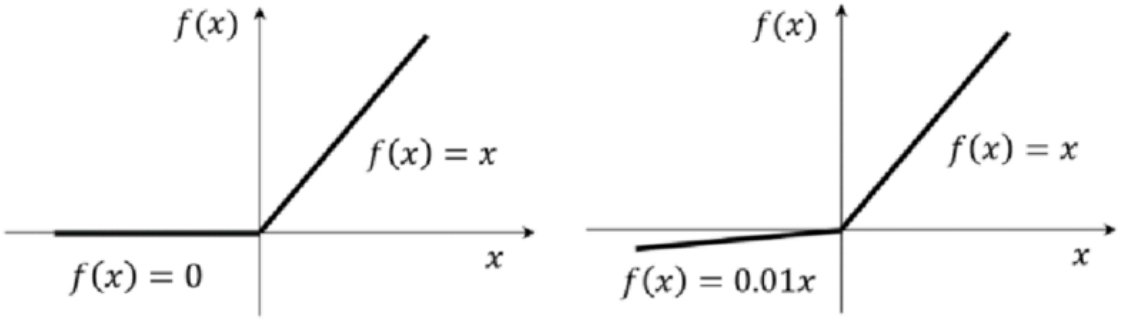
This architecture, illustrated in Figure 2 and detailed in Table 1, feeds into fully connected layers, described in detail in Model/yolo.py's self.fcs definition.

Tab. 1: YOLOv1 Core Network Architecture

Layer	Kernel Size	Output Channels	Stride	Padding
Conv	7×7	64	2	3
MaxPool	2×2	-	2	0
Conv	3×3	192	1	1
MaxPool	2×2	-	2	0
Conv	1×1	128	1	0
Conv	3×3	256	1	1
Conv	1×1	256	1	0
Conv	3×3	512	1	1
MaxPool	2×2	-	2	0
[Conv(1×1, 256), Conv(3×3, 512)] × 4				
Conv	1×1	512	1	0
Conv	3×3	1024	1	1
MaxPool	2×2	-	2	0
[Conv(1×1, 512), Conv(3×3, 1024)] × 2				
Conv	3×3	1024	1	1
Conv	3×3	1024	2	1
[Conv(3×3, 1024)] × 2				

2.1.2 CNN Block Implementation

Our implementation uses a custom CNN_block (defined in _Model/cnn_block.py_) which encapsulates a convolutional layer, batch normalization, and a LeakyReLU activation function.



(a) ReLU activation function. (b) LeakyReLU activation function.

Fig. 3: ReLU activation function vs. LeakyReLU activation function[8].

This modular block is used throughout the architecture. The operations within the CNN block can be mathematically expressed as follows:

▷ Convolution

$$z = W * x + b \qquad \text{(Convolution operation)} \tag{1}$$

Where x is the input tensor, W is the convolutional kernel, b is the bias, and z is the output of the convolution. In the CNN block the bias is set to False.

▷ Batch Normalization

$$\mu = \frac{1}{N}\sum z_i \qquad \text{(Mean of the batch)} \tag{2}$$

$$\sigma^2 = \frac{1}{N}\sum (z_i - \mu)^2 \qquad \text{(Variance of the batch)} \tag{3}$$

$$z_{\text{norm}} = \frac{z - \mu}{\sqrt{\sigma^2 + \epsilon}} \qquad \text{(Normalized output)} \tag{4}$$

$$y = \gamma \times z_{\text{norm}} + \beta \qquad \text{(Scaled and shifted output)} \tag{5}$$

LeakyReLU Activation:

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{if } x \leq 0 \end{cases} \tag{6}$$

where α is a small constant (0.1 in our implementation) that allows a small gradient to flow even when the input is negative, preventing the **dying ReLU** problem.

2.2 Output Interpretation

The output of the network is a tensor of size $(\text{Batch Size}, S \times S \times (C + B \times 5))$. This tensor represents the predictions for each grid cell:

▷ C: The number of class probabilities. Each cell predicts a probability distribution over the C classes.

▷ B: The number of bounding boxes predicted by each cell.

▷ 5: For each bounding box, the network predicts:

- x, y: The center coordinates of the bounding box relative to the cell. These values are normalized to be between 0 and 1.

- w, h: The width and height of the bounding box relative to the entire image. These values are also normalized to be between 0 and 1.
- Confidence: The confidence score of the bounding box, representing the Intersection over Union (IoU) between the predicted box and any ground truth box.

2.3 Modifications and Contributions

While our implementation builds upon existing YOLOv1 architectures, we have implemented several key modifications to enhance its training and performance:

- ▷ *AdamW (Adam with Decoupled Weight Decay):* it's an optimization algorithm that modifies the popular Adam optimizer to improve the implementation of weight decay regularization. Standard Adam implementations often couple weight decay with the adaptive learning rate mechanism, which can lead to suboptimal regularization effects, especially for parameters with large gradients. AdamW decouples the weight decay step from the gradient update, applying it directly to the weights after the adaptive gradient step, akin to how L2 regularization is typically applied in standard SGD. This decoupling often results in better model generalization and can lead to lower validation loss compared to Adam with naive L2 regularization, as it provides a more effective and predictable form of weight decay [9].

- ▷ *Cosine Annealing Learning Rate Scheduler:* Cosine annealing is a learning rate scheduling strategy designed to improve model convergence by smoothly varying the learning rate over epochs. As popularized in the context of stochastic gradient descent, the core idea is to anneal the learning rate following the shape of a cosine curve, typically starting from an initial high value and decreasing towards a minimum value (often near zero) over a predefined number of epochs ($T_{\max}$). This smooth, gradual reduction allows the model to explore the loss landscape more broadly initially and then settle into potentially wider, flatter minima as the learning rate decreases, which are often associated with better generalization performance and lower validation loss compared to minima found using step-decay or constant learning rates. While often used with warm restarts (periodically resetting the LR), the fundamental cosine decay itself is a powerful technique for stabilizing and enhancing the final stages of training[10].

- ▷ *Gradient Clipping:* Gradient clipping is a technique employed during training to mitigate the problem of exploding gradients, which can lead to numerical instability and hinder model convergence. As discussed in foundational work on training difficulties, particularly in recurrent networks, gradients can sometimes become excessively large, causing drastic updates to model parameters and potentially leading to NaN values or significant spikes in the training loss. Gradient clipping addresses this by imposing a threshold on the magnitude (typically the L2 norm) of the gradients computed during backpropagation. If the norm exceeds this predefined threshold, the gradient vector is rescaled downwards to match the threshold value before being used by the optimizer to update the model weights. This ensures that the parameter updates remain bounded, promoting more stable training dynamics and preventing divergence, which indirectly helps in achieving consistent reduction in both training and validation loss[11].

- ▷ *Data Augmentation:* We integrated data augmentation techniques, specifically horizontal flipping and random affine transformations (translation, scaling, rotation), to improve the model's robustness and generalization capabilities. These transformations are implemented in the `getitem` method of the `COCODataset` class (`DataPreparation/coco_dataset.py`). The transformations are applied using the `torchvision.transforms` module.

2.4 Loss Function

The YOLOv1 loss function is a crucial component of the model, guiding the training process by penalizing incorrect predictions. It is a multi-part loss function that combines localization

loss (bounding box coordinate errors), confidence loss (object presence/absence errors), and classification loss (class prediction errors). We use the YOLOv1_loss class in Model/losses.py to implement the loss function. The total loss function can be represented as follows 7:

$$L = \lambda_{\text{coord}} \cdot L_{\text{loc}} + L_{\text{conf}} + \lambda_{\text{noobj}} \cdot L_{\text{noobj}} + L_{\text{class}} \tag{7}$$

Where:

- ▷ L_{loc}: Localization loss (bounding box coordinate errors)
- ▷ L_{conf}: Confidence loss (for cells where an object is present)
- ▷ L_{noobj}: Confidence loss (for cells where no object is present)
- ▷ L_{class}: Classification loss
- ▷ λ_{coord}: Weight for the localization loss (set to 5 in our implementation)
- ▷ λ_{noobj}: Weight for the no-object confidence loss (set to 0.5 in our implementation)

Each component is described in detail below.

2.4.1 Localization Loss ($\mathcal{L}_{\text{loc}}$)

The localization loss measures the error in predicting the bounding box coordinates (x, y, w, h). It is calculated only for the bounding box predictor responsible for detecting the object (the one with the highest IoU with the ground truth box). The boxes1 and boxes2 used below refer to the predicted and true bounding boxes.

$$L_{\text{loc}} = \sum_{i=0}^{S^2} \sum_{j=0}^{B} I_{i,j}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + \left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \tag{8}$$

Where:

- ▷ S: The number of grid cells along one dimension of the image (e.g., 7 in a 7×7 grid).
- ▷ B: The number of bounding boxes predicted by each grid cell (2 in our implementation).
- ▷ $I_{i,j}^{\text{obj}}$: An indicator function that is 1 if the j-th bounding box in the i-th cell is "responsible" for predicting the object, and 0 otherwise. Responsibility is determined by highest IoU with the GT box.
- ▷ (x_i, y_i): The predicted center coordinates of the bounding box in cell i, relative to the cell.
- ▷ $(\hat{x}_i, \hat{y}_i)$: The ground truth center coordinates of the bounding box in cell i, relative to the cell.
- ▷ (w_i, h_i): The predicted width and height of the bounding box in cell i, relative to the entire image.
- ▷ $(\hat{w}_i, \hat{h}_i)$: The ground truth width and height of the bounding box in cell i, relative to the entire image.

The square root of the width and height is used to reduce the sensitivity to small errors in large boxes and large errors in small boxes.

2.4.2 Confidence Loss ($\mathcal{L}_{\text{conf}}$ and $\mathcal{L}_{\text{noobj}}$)

The confidence loss measures the error in predicting the confidence score of the bounding boxes. It has two components: one for cells where an object is present (L_{conf}) and one for cells where no object is present (L_{noobj}).

$$L_{\text{conf}} = \sum_{i=0}^{S^2} \sum_{j=0}^{B} I_{i,j}^{\text{obj}} (C_i - \hat{C}_i)^2 \tag{9}$$

Where:

- ▷ C_i: The predicted confidence score of the j-th bounding box in cell i.
- ▷ $\hat{C}_i$: The ground truth confidence score of the j-th bounding box in cell i. This is 1 if an object is present and the j-th bounding box is responsible for predicting it, and 0 otherwise.

$$L_{\text{noobj}} = \sum_{i=0}^{S^2} \sum_{j=0}^{B} I_{i,j}^{\text{noobj}} (C_i - \hat{C}_i)^2 \tag{10}$$

Where:

- ▷ $I_{i,j}^{\text{noobj}}$: An indicator function that is 1 if the j-th bounding box in the i-th cell is not responsible for predicting any object, and 0 otherwise. Note that $I_{i,j}^{\text{obj}}$ and $I_{i,j}^{\text{noobj}}$ are mutually exclusive.
- ▷ $\hat{C}_i$: The ground truth confidence score of the j-th bounding box in cell i. This is 0 when no object is present.

2.4.3 Classification Loss ($\mathcal{L}_{\text{class}}$)

The classification loss measures the error in predicting the class probabilities for each cell.

$$L_{\text{class}} = \sum_{i=0}^{S^2} I_i^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2 \tag{11}$$

Where:

- ▷ I_i^{obj}: An indicator function that is 1 if any object is present in cell i, and 0 otherwise. Note that this is different from $I_{i,j}^{\text{obj}}$ used in the localization and confidence losses.
- ▷ $p_i(c)$: The predicted probability of class c in cell i.
- ▷ $\hat{p}_i(c)$: The ground truth probability of class c in cell i. This is 1 for the correct class and 0 for all other classes.

We also introduce class-specific weights to the classification loss to mitigate the effects of class imbalance:

$$L_{\text{class}} = \sum_{i=0}^{S^2} I_i^{\text{obj}} \sum_{c \in \text{classes}} w_c \, (p_i(c) - \hat{p}_i(c))^2 \tag{12}$$

Where:

- ▷ w_c: The weight for class c, calculated as the inverse of the class frequency in the training dataset.

3 Results

In this section, we discuss the results we obtained while training and evaluating a streamlined version of YOLOv1. To speed things up and work within the practical limitations we faced, we adapted the model and trained it using a smaller, 5-class subset of the COCO dataset. Key deviations from the original YOLOv1 paper[1] include the following changes: we trained the model for just 35 epochs instead of 135, worked with much smaller input images (124×124 rather than 448×448), and also reduced the size of the final fully connected layers — reducing it to 1024 units instead of 4096.

Training spanned approximately 8 hours on a Colab Pro+ environment equipped with an NVIDIA A100 GPU , with sufficient system memory available throughout (16GB RAM).

Performance was evaluated using mean Average Precision (mAP@0.5).

When we trained the model, GPU usage was all over the place — you can see it in Figure 4 and Figure 5. Most of the time it hovered between 10% and 60%, but there were a few spikes, probably when saving checkpoints or loading big chunks of data. Memory, though, was super steady. The fluctuations were so small — around 0.1% — that we never had to worry about running out of it. Honestly, it just confirmed for us that gradient checkpointing really works when you're trying to keep memory usage low without throwing a ton of hardware at the problem.

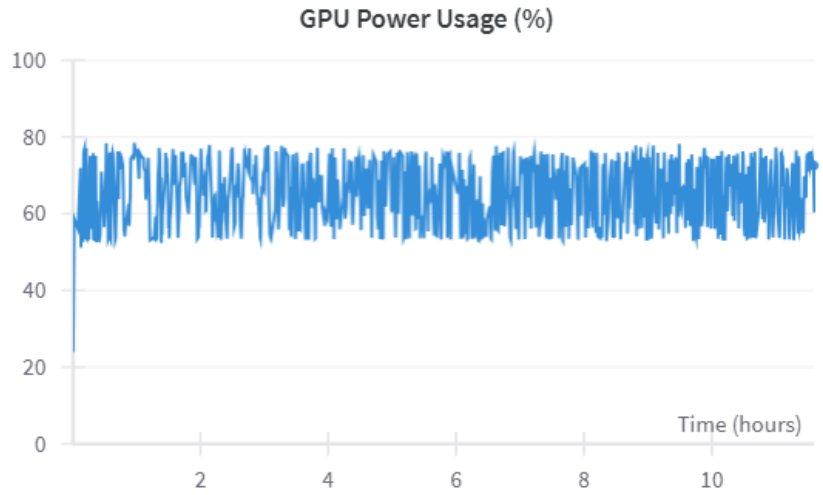


Fig. 4: GPU Power Usage Percentage[12].

Mean Average Precision: The Mean Average Precision (mAP) with an IoU threshold of 0.5 (mAP@0.5) is a standard metric used to evaluate object detection models by summarizing both classification and localization accuracy across all classes.

To calculate mAP, the Average Precision (AP) is first computed for each class by integrating the precision-recall curve over all recall values, from 0 to 1. The AP is the area under this curve, which represents the precision at different levels of recall for the given class. The final mAP is then calculated as the mean of the AP scores across all classes.

In our case, we ended up with a validation (mAP@0.5) of around 30%, as you can see in Figure 6. That's lower than the 60% mAP reported for the original YOLOv1, but that didn't really surprise us. We made some trade-offs to keep the model lightweight and fast — like using lower-resolution inputs and cutting down the number of training epochs — so we expected some drop in performance.

Fig. 5: Memory Usage Percentage[12].

The relatively low mAP also suggests that the model didn't get a full grip on all the patterns in the data — classic underfitting. If you look at the training curve, most of the big improvements happened in the first 10k steps, and then the gains slowed down. Toward the end, the curve gets a bit shaky, which probably comes from not training long enough or maybe some imbalance in the dataset that the model struggled with.

$$\text{Average Precision (AP)} = \int_0^1 P(r)\,dr \tag{13}$$

$$\text{Mean Average Precision (mAP)} = \frac{1}{C}\sum_{C=1}^{C} \text{AP}_c \tag{14}$$

If you look at Figure 6, you'll notice that the mAP shoots up pretty quickly during the first 3000 steps — looks like the model catches on fast in the beginning. But right after hitting its peak, things take a bit of a dive, and from there, the curve starts bouncing around. These ups and downs probably come from a mix of stuff: we didn't train for that many epochs, the validation data isn't perfectly consistent, and the batch size we used was fairly small. Put together, it makes the mAP curve a bit shaky and less stable than we'd like during validation.

Training Dynamics: Even though training time was pretty limited, the model actually started picking things up quite fast. When I looked at the batch loss curve (Figure 7), I noticed it bounced around a little bit at first, but then settled down nicely — mostly staying between 5 and 6 across the 35 epochs.
Validation loss (Figure 8) followed a similar pattern: a bit of a jump early on, then a steady drop, and by the end it more or less flattened out around 5.7 or 5.8.
What stood out to me was that the validation loss didn't really start creeping up again — usually a good sign that overfitting isn't happening yet. Honestly, it felt like if I had trained for a few more epochs, the model might have gotten even better without much risk.
Another thing I picked up on: after about 4000 training steps, the validation loss curve pretty much leveled off. It gave me a sense that the training was under control, despite how short the total time was.
Also, just looking at the batch loss (Figure 7), you can see quite a bit of noise — those little sharp ups and downs — which isn't surprising at all in object detection tasks, especially when the batch size is small and the data's a bit messy.

All figures were generated using Weights and Biases (wandb) logging, ensuring reproducibility and traceability of the training process.

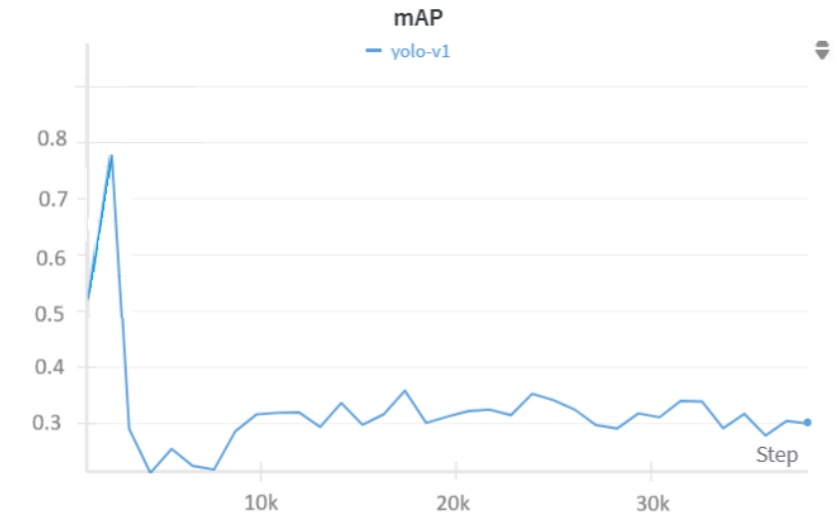


Fig. 6: Mean Average Precision Plot[12].

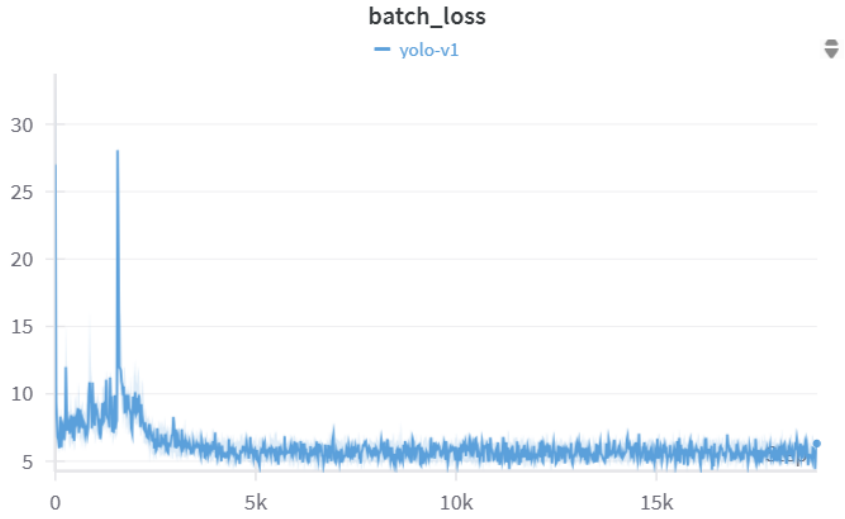


Fig. 7: Batch Training Loss[12].

Visualization of Detection Results: To get a clearer sense of how the model is performing, you can see a set of validation images with the predicted bounding boxes overlaid in Figure 9(b). For each detected object, there is a predicted class label and a confidence score. We used red boxes to indicate the model's predictions, while green boxes show the ground truth annotations when available.
For comparison, we've also included the original images without any annotations in Figure 9(a). Pictures have been selected randomly from the dataset. Seeing the inputs alongside the model's outputs makes it easier to spot where the model gets it right — and where it doesn't. This side-by-side view offers a more intuitive feel for the model's strengths and its occasional missteps.

The corresponding images can be accessed from the project's GitHub repository.

4 Conclusions

In this coursework, we tackled the challenge of efficient real-time object detection by developing a lightweight adaptation of the YOLOv1 architecture. Main key deviations from the main paper

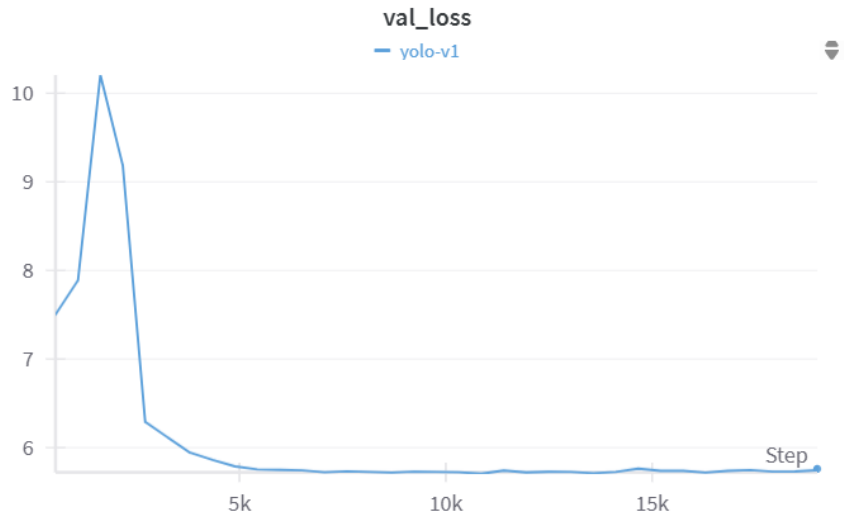


Fig. 8: Validation Loss[12].

included simplifying the fully connected layers and limiting the training schedule to 35 epochs. We applied these changes to enable practical deployment under constrained computational resources.

Our adapted YOLOv1 model achieved a validation mAP@0.5 of approximately 30%. It represents that effective object detection is still feasible under tight resource constraints. This performance highlights the enduring strength of YOLOv1's regression-based formulation.

This project shows how traditional object detection models can be refreshed and enhanced by blending core ideas with newer techniques. It also highlights how important it is to strike the right balance between model size, speed, and accuracy—especially when working with limited computing resources.

Future improvements could include training the model for more epochs, using higher-resolution inputs, and exploring more efficient lightweight backbone architectures. Another promising direction is to further refine the contrastive learning objectives to better align learned feature representations with the detection task. Comparing different contrastive loss functions—such as NT-Xent, triplet loss, or supervised contrastive loss—could also help identify which formulation best supports object-level discrimination in this context.

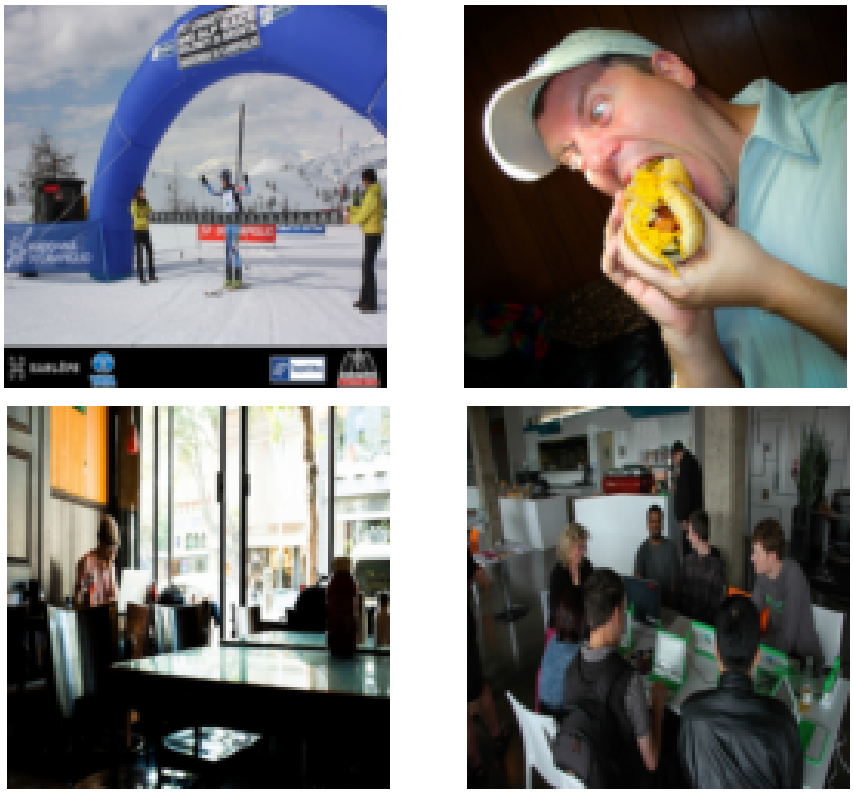
To better understand how the model performs under resource constraints, we also visualized its predictions by overlaying bounding boxes on validation images, offering a qualitative view of its detection capabilities (see Figure 9).

Finally, this report was fully written and typeset using LaTeX, ensuring professional structure, reproducibility, and adherence to academic standards.

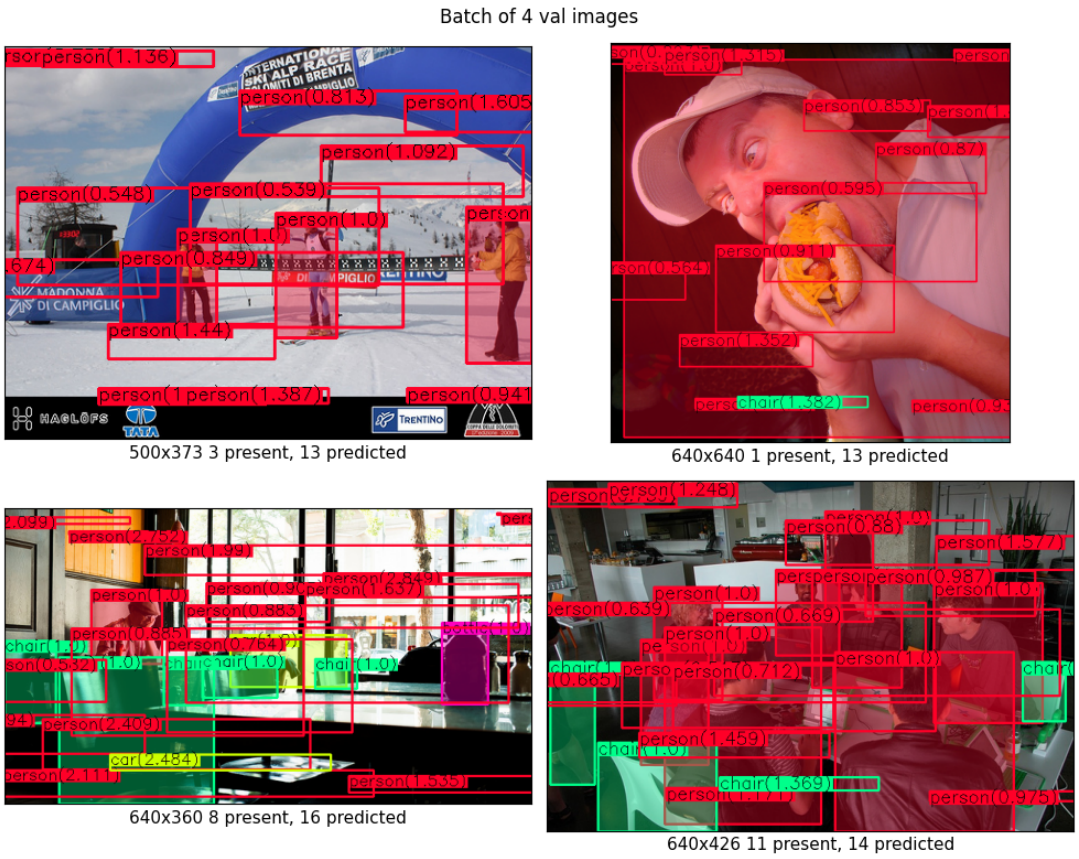
5 Reflections

This coursework provided a valuable opportunity to deepen my understanding of lightweight object detection, system troubleshooting, and project management under real-world constraints. While the technical aspects of developing a simplified YOLOv1 model were challenging, the infrastructure and operational issues I encountered were equally educational.

One of the primary challenges arose during the setup phase on the Hyperion cluster. The available OpenSSL version was severely outdated (OpenSSL 1.0.2k-fips from 2017), which was incompatible with the Python 3.11.9 environment required by libraries such as `ray[default]` and `gymnasium`. Attempting to import the `ssl` module or install related packages led to persistent errors (e.g., `ImportError: libssl.so.1.1: cannot open shared object file`). Without sudo privileges to upgrade OpenSSL, and facing network restrictions that prevented using



(a) Original input images: Randomly selected validation images without any annotations, used for comparison against the model's predicted outputs [12].



(b) Predicted bounding boxes: Model predictions on a batch of validation images, with each bounding box annotated by the predicted class label and confidence score. Red boxes indicate model predictions, and green boxes represent ground truth annotations. The full set of images is available in the GitHub repository under the subdirectory working/plots[13].

Fig. 9: Visualization of model predictions on a batch of validation images. (9(a)) Original validation images without annotations. These visualizations provide qualitative insights into the model's detection performance (9(b)) Predicted bounding boxes with class labels and confidence scores. [13].

`pyenv` to install an older Python version, I had to invest significant time troubleshooting. Despite proposing potential solutions, a fully satisfactory fix was not feasible within the coursework timeline, delaying progress on model development and experimentation.

Another significant hurdle involved managing a large dataset. I initially worked with a COCO dataset subset of approximately 35GB, which, although manageable relative to the allocated personal storage, proved challenging for practical training and file handling under the computational and network constraints of the Hyperion cluster. To address this, I sampled a smaller, representative portion of the dataset. Although this workaround allowed the project to proceed, it likely impacted the final model's generalization ability compared to training on the full dataset. Furthermore, due to the large file transfers involved and the prolonged process times (sometimes extending to 7–8 hours), unstable university internet connections occasionally interrupted the workflow, further delaying progress. To mitigate these disruptions and ensure uninterrupted training, I ultimately subscribed to Google Colab Pro+, which provided the necessary computational resources and stability to complete model training and experimentation.

Prior to this project, I had not worked with datasets of this magnitude. Consequently, I faced difficulties when attempting to push generated project files to GitHub, as several exceeded the platform's 100MB file size limit. As a workaround, I uploaded large assets to Google Drive and shared download links instead. I also explored using Git Large File Storage (Git LFS), which underscored the importance of planning for data management and version control when dealing with large-scale machine learning projects.

Despite these technical obstacles, the project was a success. Key insights gained include the critical importance of verifying system compatibility early, allocating contingency time for environmental setup, maintaining flexibility with training plans in the face of hardware or software limitations, and having backup computational solutions when primary resources fail.

If I were to repeat this coursework, I would explore alternative methods for transferring data to Hyperion instead of relying solely on FileZilla. I would experiment with smaller datasets to reduce data management complexity and training time. Additionally, I would consider using alternative packages to avoid compatibility issues. Finally, I would aim to implement a model even closer to the original YOLOv1 architecture to maximize performance and alignment with the project's objectives.

References

[1] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[2] M. Shenoda, "Real-time Object Detection: YOLOv1 Re-Implementation in PyTorch," *arXiv preprint arXiv:2305.17786*, 2023.

[3] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 580–587, 2014.

[4] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, pp. 1440–1448, 2015.

[5] COCO Consortium, "Coco - common objects in context," 2014. Accessed 2024.

[6] R. Jiang, Q. Lin, and S. Qu, "Let blind people see: real-time visual recognition with results converted to 3d audio," *no. January*, 2016.

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[8] Z. Li, W. Nash, S. O'Brien, Y. Qiu, R. Gupta, and N. Birbilis, "cardigan: A generative adversarial network model for design and discovery of multi principal element alloys," *Journal of Materials Science & Technology*, vol. 125, pp. 81–96, 2022.

[9] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[10] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," *arXiv preprint arXiv:1608.03983*, 2016.

[11] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *International conference on machine learning*, pp. 1310–1318, Pmlr, 2013.

[12] A. Mostafavi, "https://wandb.ai/anndischeh-univ-/Deep Learning for image Analysis," 2025.

[13] A. Mostafavi, "https://github.com/Anndischeh/Deep-Learning-for-Image-Analysis-Coursework," 2025.