Preprints.org

Article

Not peer-reviewed version

Toward a Deeper Understanding of YOLO26: Block-Level Architectural Analysis and Ablation Studies

Marc Tornero-Soria *, Antonio-José Sánchez-Salmerón, Eduardo Vendrell-Vidal

Posted Date: 1 April 2026

doi: 10.20944/preprints202603.2518.v1

Keywords: YOLO; YOLO26; object detection; deep learning; computer vision; ablation study; latency benchmarking

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 April 2026 doi:10.20944/preprints202603.2518.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

Toward a Deeper Understanding of YOLO26: Block-Level Architectural Analysis and Ablation Studies

Marc Tornero-Soria *, Antonio-José Sánchez-Salmerón and Eduardo Vendrell-Vidal

Department of Systems Engineering and Automation, Universitat Politècnica de València (UPV), Camí de Vera s/n, 46022 València, Spain

* Correspondence: mtornero@doctor.upv.es

Abstract

Public YOLO model releases typically provide high-level architectural descriptions and headline benchmark results, but offer limited empirical attribution of performance to individual blocks under controlled training conditions. This paper presents a modular, block-level analysis of YOLO26's object detection architecture, detailing the design, function, and contribution of each component. We systematically examine YOLO26's convolutional modules, bottleneck-based refinement blocks, spatial pyramid pooling, and position-sensitive attention mechanisms. Each block is analyzed in terms of objective and internal flow. In parallel, we conduct targeted ablation studies to quantify the effect of key design choices on accuracy (mAP50–95) and inference latency under a fixed, fully specified training and benchmarking protocol. Experiments use the MS COCO [1] dataset with the standard train2017 split (≈118k images) for training and the full val2017 split (5k images) for evaluation. The result is a self-contained technical reference that supports interpretability, reproducibility, and evidence-based architectural decision-making for real-time detection models.

Keywords: YOLO; YOLO26; object detection; deep learning; computer vision; ablation study; latency benchmarking

---

1. Introduction

Object detection has advanced rapidly over the past decade, with one-stage detectors such as the YOLO (You Only Look Once) family [2–13] offering a practical balance of detection accuracy and inference speed for real-time computer vision applications. In this context, the question of which YOLO version is "best" is rarely absolute. It is typically framed as a trade-off between detection accuracy and inference speed. Prior studies and surveys emphasize that no single detector universally dominates across datasets [14–16], tasks, and evaluation settings, and that reported improvements can be sensitive to training recipes and benchmarking assumptions. MS COCO has nevertheless remained a common reference benchmark for comparing YOLO variants, making it a useful baseline for controlled analysis.

YOLO26, released by Ultralytics on January 14, 2026, represents a recent step in this lineage improving the accuracy–efficiency frontier on COCO relative to earlier YOLO releases (Figure 1). However, understanding why a given YOLO variant performs well remains challenging without component-level attribution that links architectural choices to measurable outcomes. Public descriptions and early analyses [17]often summarize model changes and report aggregate benchmark numbers, but they rarely quantify the marginal value of each architectural component under controlled conditions.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

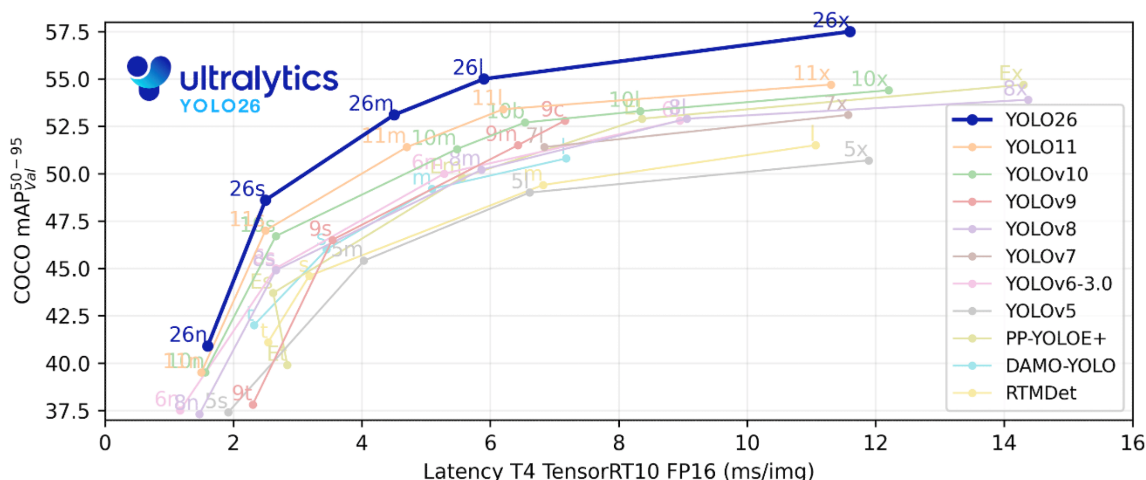


Figure 1. Ultralytics-reported COCO accuracy vs latency on NVIDIA T4.

This gap is compounded by the fact that modern detector performance on COCO is frequently influenced by pretraining on large external datasets (e.g., Objects365) prior to COCO fine-tuning. While such pretraining can improve absolute metrics, it complicates interpretation when the goal is to understand architectural effects or to reproduce results under a training-from-scratch setting. As a result, researchers and practitioners may know what modules exist in the network, but not which ones provide the most "bang for the buck" in the accuracy–latency trade-off under a consistent recipe.

This paper addresses these challenges through two complementary efforts focused on YOLO26n (nano), chosen as a representative high-efficiency variant where architectural choices most directly impact latency and deployment practicality. First, we provide a structured, block-by-block dissection of YOLO26n. Using a consistent example input, we trace tensor shapes and feature transformations across the backbone, and neck, and we interpret the functional role of each unique module. Second, we conduct a controlled ablation suite that modifies one factor at a time (e.g., activation functions, C3k2 variants, SPPF settings, attention configurations) while holding the training and evaluation protocol constant. We report absolute metrics enabling evidence-based conclusions about which architectural elements materially affect COCO mAP50–95 and inference latency.

Scope and non-goals. The objective of this work is not to maximize absolute COCO mAP50–95 or to reproduce Ultralytics-reported headline benchmarks under large-scale pretraining and extensive recipe tuning. Instead, our goal is to compare YOLO26n architectural variants under identical training conditions to isolate the effect of specific module choices. Accordingly, we do not perform hyperparameter searches, multi-dataset pretraining (e.g., Objects365), or other optimizations aimed primarily at leaderboard performance. All ablations use a fixed base configuration and differ only in the component under study, enabling meaningful and reproducible comparisons.

Experimental rationale. We hypothesize that Ultralytics' default YOLO26n configuration represents a strong optimum in the accuracy–efficiency space, but that the marginal value of individual architectural choices is not well quantified publicly. To test this hypothesis, we hold the training and benchmarking protocol constant and perform targeted ablations of individual modules and settings. By measuring accuracy and latency jointly, we identify components that are consistently beneficial, components whose gains are marginal, and components whose cost may outweigh their benefit under the tested constraints.

This paper makes the following contributions:

1. Block-level architectural analysis: A module-by-module dissection of YOLO26n detailing operations, tensor dimensions, and transformations across backbone, and neck.
2. Functional interpretation: Explanations of what each component does and why it is included

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 April 2026 doi:10.20944/preprints202603.2518.v1

3. Controlled ablation suite: A set of targeted ablations evaluated on MS COCO train2017/val2017 that quantifies the impact of individual design choices on mAP50–95 and latency under a fixed compute and training recipe.
4. Reproducibility protocol: A fully specified configuration and benchmarking methodology to support replication and future extension.

2. Related Work

The YOLO family has been widely studied through surveys and comparative reviews [18–20] that document architectural evolution and performance trends across versions, often at the level of backbone–neck design patterns, training strategies, and speed–accuracy trade-offs. Several architecture-centric reviews further attempt to reconstruct detailed block definitions by cross-referencing documentation with released code [15,21], motivated by the fact that official architectural schematics and scholarly write-ups are not always available for fast-moving YOLO releases. These works provide valuable context and help standardize terminology, but they typically remain descriptive and do not quantify the marginal contribution of individual modules under controlled experimental conditions.

For YOLO26 specifically, the current literature is limited to a small number of early analyses. Sapkota et al. [17] summarize YOLO26's main design themes and report performance benchmarks across tasks and deployment targets, emphasizing changes such as end-to-end (NMS-free) inference, the removal of Distribution Focal Loss (DFL) [22], and training-time mechanisms including ProgLoss, STAL, and the MuSGD optimizer. Chakrabarty [23] focuses on YOLO26's end-to-end/NMS-free paradigm and its training and deployment implications, including deterministic latency and export-related motivations. Hidayatullah and Tubagus (preprint) [24] provide a code-derived overview of Ultralytics YOLO26 and discuss several reported architectural and training changes.

Despite these contributions, prior YOLO26 work does not provide a systematic, block-level attribution analysis of YOLO26 under controlled conditions. In particular, the available YOLO26-specific studies do not jointly provide (i) block-by-block tensor-shape and information-flow tracing, (ii) detailed functional interpretations, and (iii) marginal accuracy and latency attribution under a fixed, specified training and benchmarking protocol. Our work complements these initial YOLO26 analyses by offering a block-by-block dissection of YOLO26n and a controlled ablation suite that modifies one architectural factor at a time and measures its effect on COCO mAP50–95 and inference latency under identical training conditions.

All architectural schematics and tensor-shape traces reported in this paper were produced by the paper authors derived from the official Ultralytics YOLO26 configuration and module definitions.

3. Approach

This section describes (i) the methodology used to analyze YOLO26 at the block level and (ii) the experimental protocol used to run controlled ablations and latency benchmarks. To reduce ambiguity and improve reproducibility, we explicitly separate the architecture analysis procedure from the training and evaluation procedure.

3.1. Architecture Analysis Methodology (Objective → Flow)

Our architecture analysis proceeds in a structured, repeatable manner for each unique module used in YOLO26:
1. Objective. We describe the goal of the block in the context of detection
2. Flow. We document the internal sequence of operations and how information is routed through the block.

We begin with a generic YOLO26 diagram that is applicable across model scales. We then instantiate the YOLO26n configuration with a fixed example input size and trace tensor shape evolution stage-by-stage through the backbone, and neck. This progression from general to specific

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 April 2026 doi:10.20944/preprints202603.2518.v1

enables readers to understand both the configurable macro-architecture and the concrete micro-level transformations induced by each module.

Although YOLO26 is available in multiple model sizes, we focus on YOLO26n because it is the most compute-constrained variant and therefore the most sensitive to architectural trade-offs—making it a practical testbed for "bang-for-buck" design decisions. Where relevant, we comment on whether a given observation is likely to generalize to larger variants, while noting that validation across all sizes is outside the scope of this study.

3.2. Experimental Setup and Controlled Ablation Protocol

In parallel with the block-level analysis, we perform controlled ablation studies to quantify the marginal effect of specific architectural choices on both accuracy and latency. Our experiments are designed around a single principle: change one factor at a time while holding the rest of the pipeline constant.

Dataset and evaluation. All models are trained on MS COCO using the standard train2017 split (≈118k images) and evaluated on the full val2017 split (5k images). Accuracy is reported using COCO mAP50–95. Latency is measured using a fixed benchmarking procedure described in Appendix B, including the exact runtime settings and methodology.

Training from scratch vs pretraining (interpretation note). Ultralytics-reported COCO results for some YOLO26 variants rely on pretraining on external data (e.g., Objects365) prior to COCO fine-tuning. Consequently, COCO-only training from scratch is not expected to match those headline metrics. In this work, we primarily train from scratch to ensure that observed differences arise from architectural changes rather than inherited representations from pretraining.

Training duration and convergence control. In preliminary experiments, we observed that YOLO26n trained from scratch may require extended training to reach peak mAP50–95, with continued gains beyond typical short schedules. To reduce the risk of comparing under-trained variants, we set a maximum of 3000 epochs and enable early stopping with patience = 300, terminating training if validation mAP50–95 fails to improve for 300 epochs. This schedule is applied uniformly across the baseline and all ablations.

Configuration transparency and reproducibility. Unless stated otherwise, all runs use the same base training configuration (training arguments and values) provided in the paper's Appendix A. This includes optimizer and learning-rate schedule settings, augmentation parameters, image size, batch size, and any other training-time options that materially affect outcomes. The latency benchmarking arguments used to obtain inference times for each ablation are also documented to enable direct replication and described in Appendix B.

Ablation design. We focus on ablations that reflect meaningful architectural and implementation choices commonly considered by practitioners, including activation functions, C3k2 variants, SPPF settings, and attention module configurations. Each ablation is evaluated against the current Ultralytics baseline module or configuration under identical training and evaluation conditions. We report absolute performance in terms of both accuracy and inference speed.

Outputs. The combined outcome of this methodology is (i) an interpretable, block-level explanation of YOLO26n and (ii) an evidence-backed set of ablation results that highlight which modules drive improvements and which offer limited benefit relative to their cost, guiding future architectural iterations and practical deployment decisions.

4. Network Architecture Overview

The YOLO26 model is organized into a sequence of stages and blocks that transform an input image through a hierarchical pipeline, ultimately producing multi-scale feature representations for detection.

The backbone initiates feature extraction. Early convolutional layers downsample the input, reducing spatial resolution while capturing low-level patterns such as edges and textures. These are followed by repeated C3k2 modules, which refine features efficiently by capturing local context and

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

progressively strengthening representation capacity. As depth increases, shallow layers retain fine-grained spatial details, while deeper layers generate more abstract, high-level semantic features. This process yields feature maps at multiple scales (e.g., P3, P4, and P5), supporting detection of objects with varying sizes.

The SPPF module then aggregates information across multiple receptive-field scales, enriching the feature representation with broader contextual cues. Following this, the C2PSA module combines convolutional processing with position-sensitive attention to capture both local detail and longer-range dependencies while preserving spatial structure. This enhances the discriminative power of features at the backbone–neck interface and improves overall detection performance.

The neck further refines extracted features through multi-stage fusion. Upsampling operations increase spatial resolution, after which concatenation layers merge these features with corresponding features from earlier stages. C3k2 modules, together with upsampling and concatenation layers, combine low- and high-level features along channels, progressively upsampling or downsampling fused features to integrate information across layers [25–27]. Additional C3k2 modules then process the fused representations to enhance feature consistency and predictive performance across scales.

The overall architecture presented in Figure 2 is consistent across all YOLO26 variants: Nano, Small, Medium, Large, and Extra Large. The exact tensor dimensions at each stage are determined by two scaling parameters: w (width multiplier), and mc (maximum number of channels).

Additionally, the number of blocks in the C3k2 and C2PSA modules depend on the result of multiplying 2 with another parameter: d (depth multiplier).

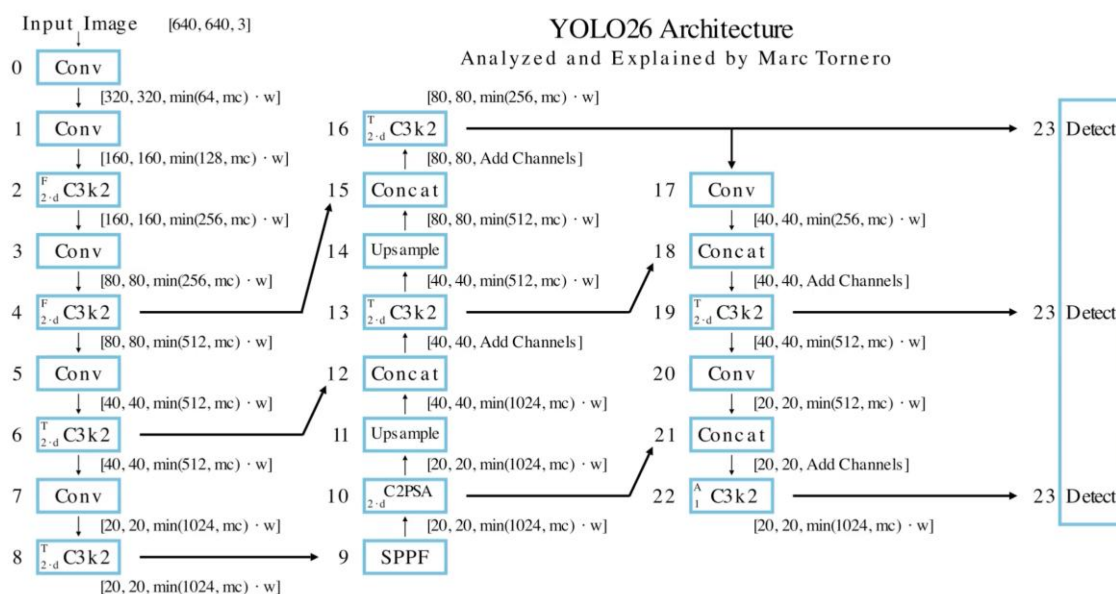


Figure 2.

For example, in the nano configuration, the scaling parameters are set to d or depth of 0.5, a w or with of 0.25, and an mc or maximum number of channels of 1024.

Model	Depth (d)	Width (w)	Max Channels (mc)
yolo26n	0.50	0.25	1024

Figure 3.

Substituting these values into the architecture formula yields the corresponding number of block repetitions and the tensor shapes at each stage.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

doi:10.20944/preprints202603.2518.v1

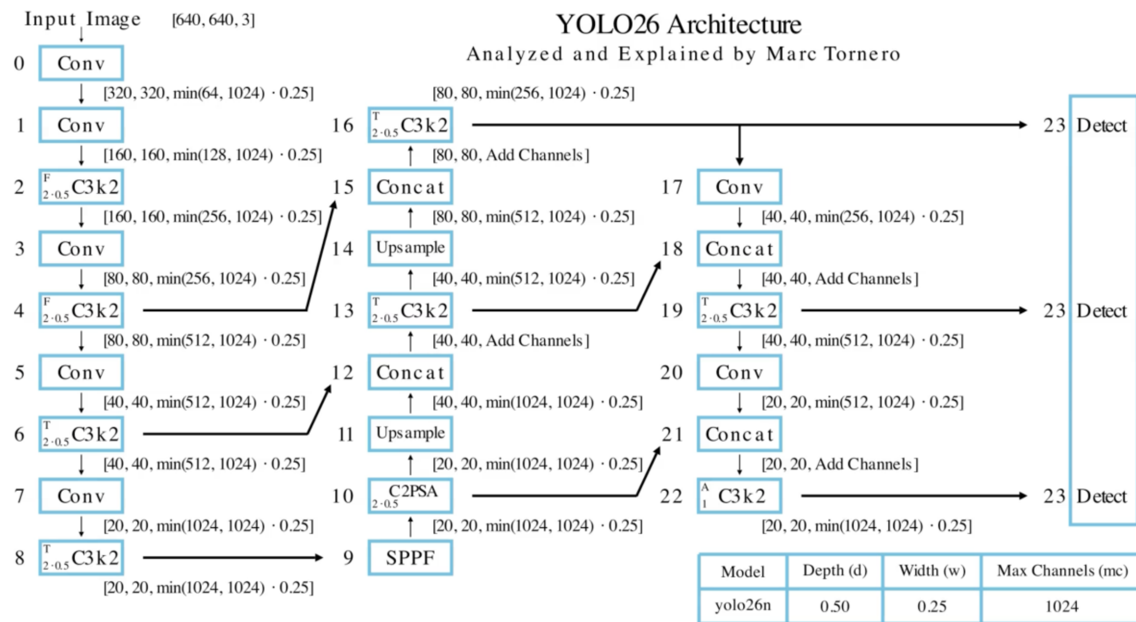


Figure 4.

We also explicitly note the shapes for the cross-stage tensor to further enhance clarity.

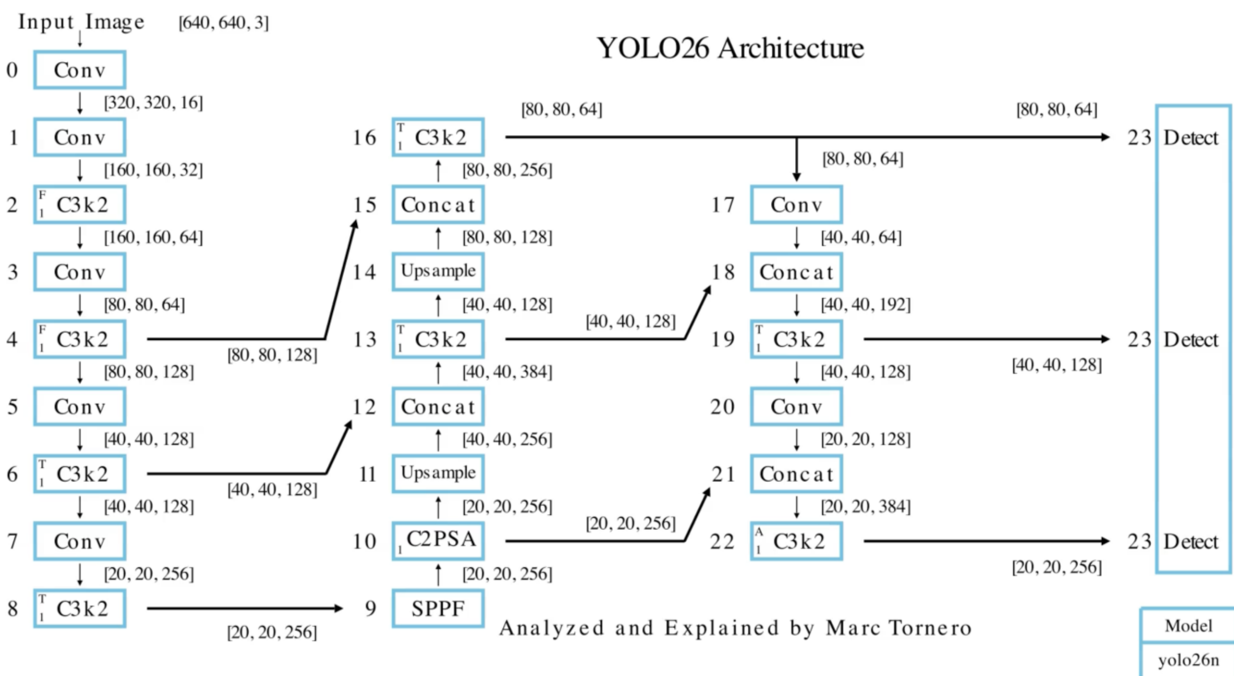


Figure 5.

To illustrate how tensor shapes evolve through the network for a concrete input, we consider the example image shown in Figure 6, with original dimensions 768×1024 (height × width). Tensor shapes throughout the network depend on the spatial resolution of the input. Therefore, feature-map heights and widths vary with the input height and width and will differ for other input resolutions.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 April 2026 doi:10.20944/preprints202603.2518.v1

Figure 6.

The network input is not required to be square (e.g., 640×640). In our preprocessing pipeline, images are resized while preserving aspect ratio such that the longer side is 640 pixels. The shorter side is then adjusted to be divisible by the network's largest stride (32) to ensure that all downsampling stages produce integer-valued feature-map sizes. If necessary, letterboxing pads the image to the nearest multiple of 32.

Using this procedure, the example image in Figure 6 is resized (and, if necessary, letterboxed) to produce the network input of 480×640×3, shown in Figure 7.

Figure 7.

The corresponding tensor shapes throughout the network (including intermediate cross-stage tensors) are reported in Figure 8. For consistency with PyTorch conventions, all tensor shapes are expressed as [batch, channels, height, width].

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

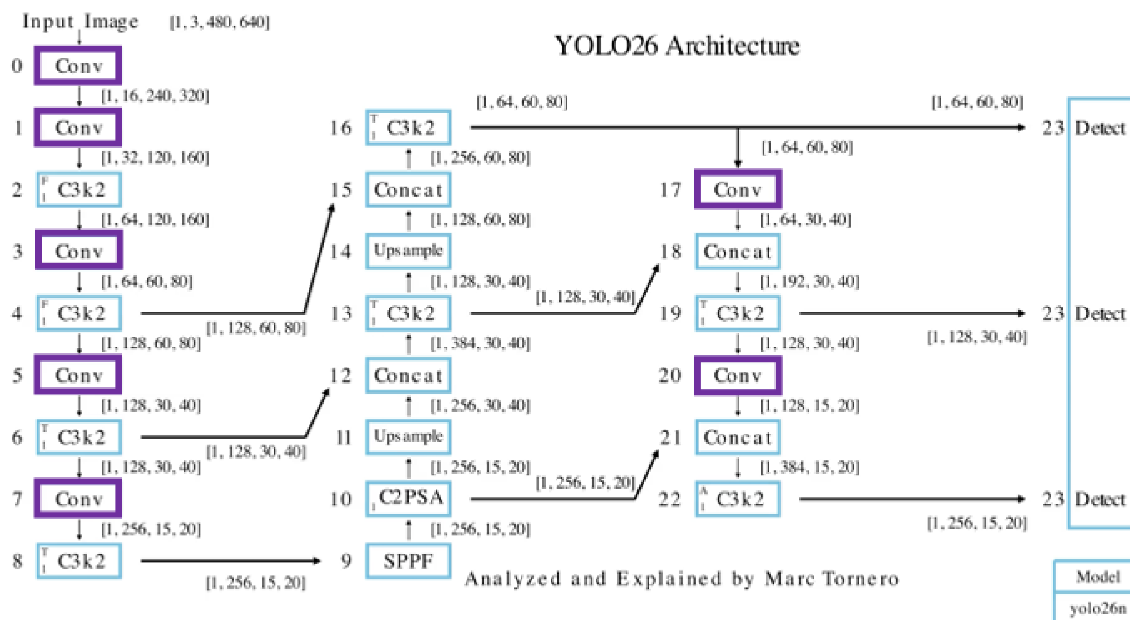


Figure 8.

This notation and these dimensions are used throughout the paper as we examine individual modules and their internal steps.

5. Convolutional Block (Conv)

5.1. Objective

Convolutional layers are the backbone of neural networks for object detection, playing a vital role in:

- Feature extraction: By applying filters (kernels) to input data, they learn spatial hierarchies of patterns, such as edges, textures, shapes, and objects.
- Hierarchical learning: Stacking multiple layers allows for capturing increasingly abstract and complex features, from low-level edges in early layers to high-level representations in deeper layers.
- Progressive downsampling: These layers reduce the spatial dimensions of the input image (e.g., P1 → P2 → P3) while increasing channel depth. This process enables efficient feature compression, reducing computational cost while preserving critical information. By maintaining spatial relationships, the network retains the structural arrangement of the data and preserves meaningful features.
- Parameter sharing: Convolutional filters are reused across the input, reducing the number of learnable parameters and enabling translational invariance. Common variants include standard convolution (3×3), pointwise convolution (1×1), and depthwise convolution (DWConv) [28].

5.2. Flow

Each convolutional block consists of three sequential operations:

1. Convolution: Applies a learned kernel (technically implemented as cross-correlation) to extract local spatial features.
2. Batch normalization (BN) [29]: Normalizes intermediate activations to stabilize training and accelerate convergence.
3. Activation: Introduces non-linearity, enabling the network to model complex patterns. YOLO26 uses SiLU (also known as Swish) [30,31] as the default activation, which provides a smooth,

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 April 2026 doi:10.20944/preprints202603.2518.v1

non-linear transformation. With sufficient SiLU neurons, the network can approximate complex functions with high fidelity.

During inference warm-up, convolution and BN layers are typically fused. Because BN parameters are frozen after training, its scaling and shifting can be absorbed into the preceding convolution, producing a single convolutional layer with updated weights and biases. This pre-fusion step reduces computational overhead without altering the network's behavior.

To analyze the effect of the activation function, we train three YOLO26n variants from scratch for up to 3000 epochs on the COCO dataset, differing only in their activation functions: (1) all SiLU, (2) all ReLU, and (3) all Leaky ReLU. As shown in Figure 9, the SiLU variant consistently achieves the highest mAP@0.5:0.95 across the entire training run, and training terminates early at epoch 2591 because no further improvement is observed.

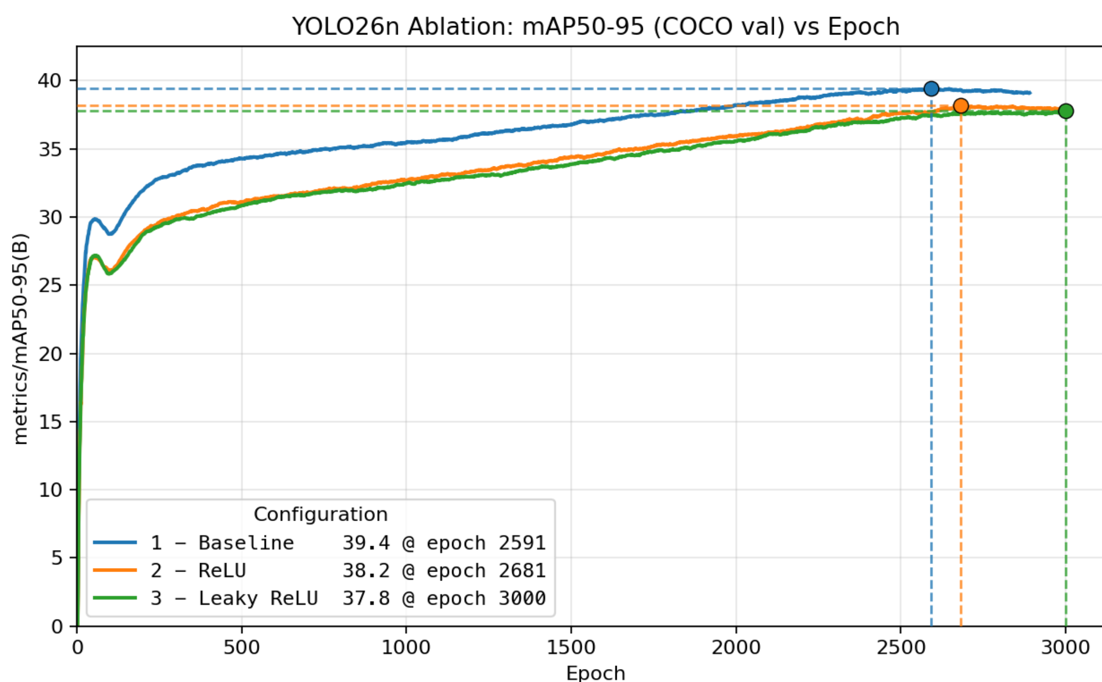


Figure 9.

Benchmarking these three activation variants on TensorRT 10 (FP16) with an H100 GPU (Table 1) confirms a clear accuracy–latency trade-off. The baseline configuration (1, all SiLU) achieves the highest mAP@0.5:0.95 (0.3933) but also exhibits the highest latency (1.05 ms). Switching to ReLU (2) produces the lowest latency (0.93 ms), but reduces accuracy to 0.3808 mAP@0.5:0.95. Using Leaky ReLU (3) provides no benefit: it yields the lowest accuracy (0.3761) while remaining slower than ReLU at 1.02 ms. Therefore, between ReLU and Leaky ReLU, ReLU is preferable on both accuracy and latency.

Ultralytics adopts SiLU because the accuracy improvement over ReLU is substantial (0.3933 vs. 0.3808 mAP@0.5:0.95), and this gain can justify the modest latency increase for applications where detection quality is prioritized. In this sense, SiLU represents the best overall accuracy-focused choice among the tested activations, while ReLU is the best speed-focused alternative.

Table 1.

Architecture	mAP@0.5:0.95	Latency (ms)
1 – Baseline	**0.3933** ☑	**0.99**
2 – ReLU	0.3808	0.93 ☑
3 – Leaky ReLU	0.3761	1.02

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 April 2026
doi:10.20944/preprints202603.2518.v1

6. Feature Refinement Module (C3k2 with Argument False)

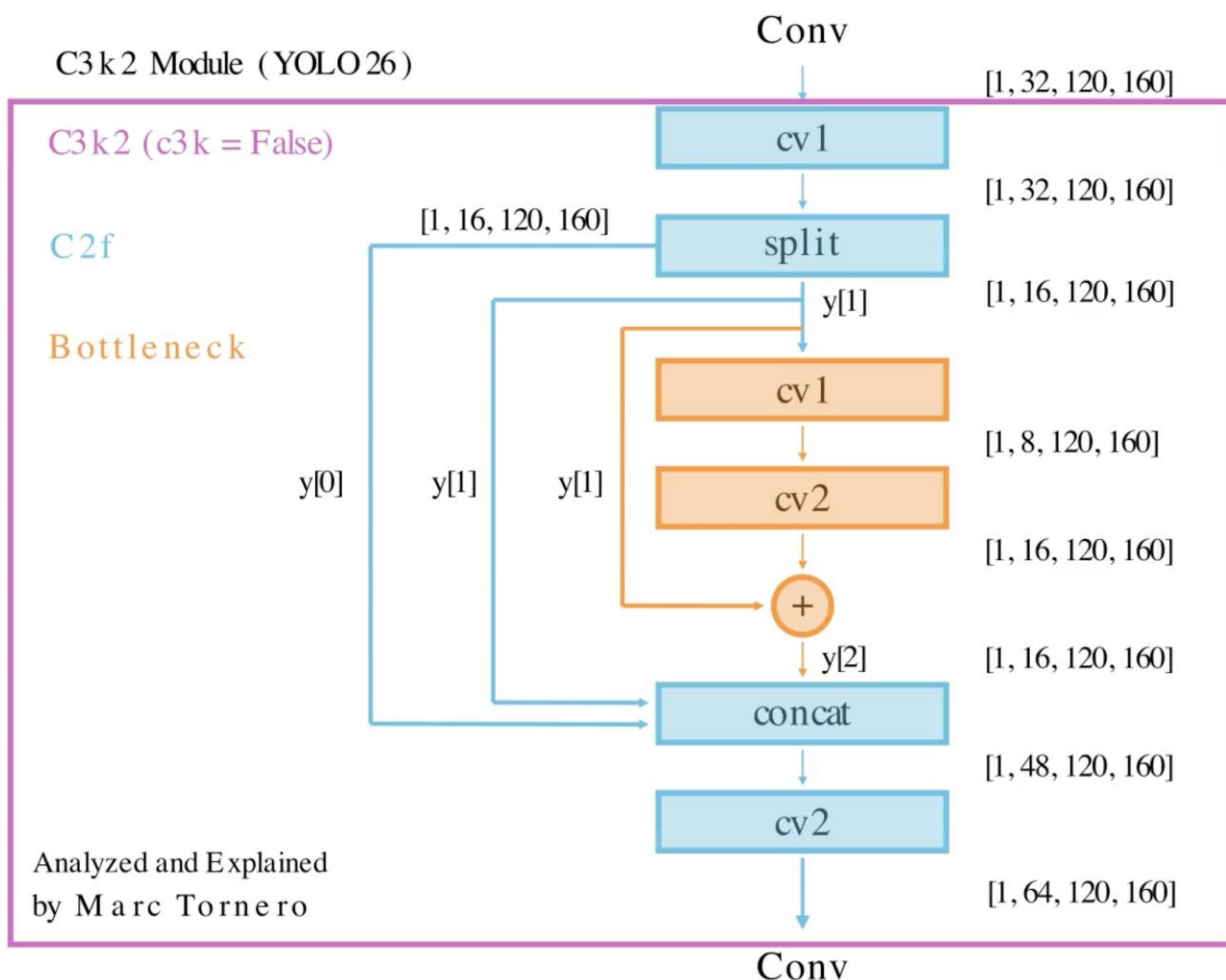


Figure 10.

6.1. Objective

The C3k2 (F) module is a lightweight feature refinement block designed to efficiently enrich and transform features within the YOLO26 architecture. Standard deep networks often suffer from computational redundancy, where the same information is processed multiple times across layers. The Cross Stage Partial (CSPNet) approach [32] mitigates this by splitting the feature map into two parts:

- Refinement path: one part passes through a compact bottleneck with residual connections [33] to extract refined features.
- Shortcut path: the other bypasses the bottleneck entirely.

The two paths are then recombined through concatenation, followed by a projection into a richer, more discriminative feature space. This design improves representational capacity while preserving computational efficiency. Inside the C3k2 (F) module, the bottleneck block inside excels in several key areas:

- Compression phase (Squeezing features): The first convolution in the bottleneck block reduces the input channels. This dimensionality reduction focuses on distilling the most critical information while discarding less important features.
- Processing in the bottleneck: The reduced feature set undergoes transformations (convolutions and activations) to refine patterns efficiently. This step emphasizes core patterns while conserving computational resources.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

- Expansion phase (Rebuilding features): The final convolution expands the channels back to ensure the network retains capacity for complex pattern modeling. This combines the critical features from compression with the structural richness needed for downstream tasks.
- Promoting a compact and informative representation: By alternating between high and low-dimensional spaces, the Bottleneck prioritizes relevant features, retaining only the most useful information.
- Scalability: In YOLO26, C3k2(F) scales its internal depth with model size: nano/small/medium use one Bottleneck, while large/extra-large use two Bottlenecks in series between split and concat.

The C3k2 module represents YOLO26's evolution of the C2f module, first introduced in YOLOv8 by Ultralytics [8]. The C2f structure subsequently became a key building block in modern YOLO architectures, such as YOLOv10 [10], and YOLO11 [11], typically incorporating a default Bottleneck module. Originally introduced in YOLOv3 [3], the Bottleneck has remained a core component of increasingly sophisticated modules across successive YOLO versions. By leveraging the C3k2 module, YOLO26 achieves high precision while using fewer parameters than its predecessors [17], making it computationally efficient without sacrificing accuracy.

6.2. Flow

The operations within the C3k2 (F) module are as follows:

- Initial convolution (cv1): A 1×1 convolution is applied to the input tensor, preserving spatial resolution while transforming features and mixing channels.
- Split: The input tensor is divided along the channel dimension into two groups (32 channels → 16 + 16). One half (y[0]) is preserved for an identity/skip connection, while the other half (y[1]) is passed through the bottleneck for transformation.
- Bottleneck: The bottleneck processes y[1] via two consecutive 3×3 convolutions. The first reduces channels (16 → 8), compressing information and enabling learning in a reduced space. The second restores channels to the original number (8 → 16), expanding the feature representation while incorporating local spatial context. This encourages the network to capture structured patterns (curves, corners, textures) that are critical for object boundaries and class distinctions.
- Concatenation: The outputs from the split step and the bottleneck result are concatenated along the channel axis (y = [y[0], y[1], bottleneck's output]). This operation merges raw and transformed features, creating a multi-view representation that enhances the network's ability to detect diverse patterns.

Because y[0] and y[1] are passed through unchanged, the network can learn the optimal balance between retaining original input features and incorporating the bottleneck-processed features. This design also facilitates better gradient flow during training.

- Final convolution (cv2): A 1×1 convolution is applied to the concatenated tensor to learn inter-channel relationships and project the features to the desired number of channels (48 → 64), enriching the feature space.

7. Feature Refinement Module (C3k2 with Argument True)

All C3k2 (T) modules are highlighted in green in the architecture diagram below.

The double-bottleneck from Figure 11 is further expanded below in Figure 12 for more clarity.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 April 2026 doi:10.20944/preprints202603.2518.v1

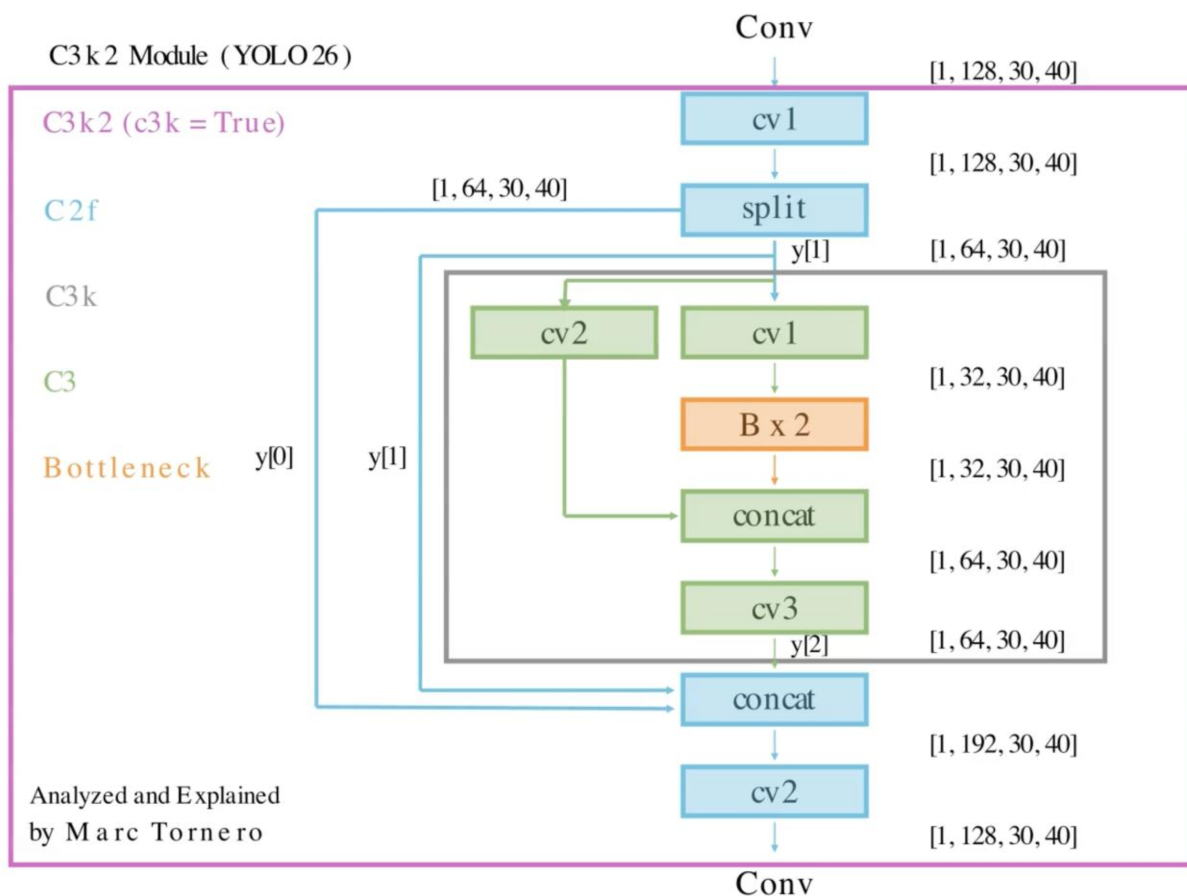


Figure 11.

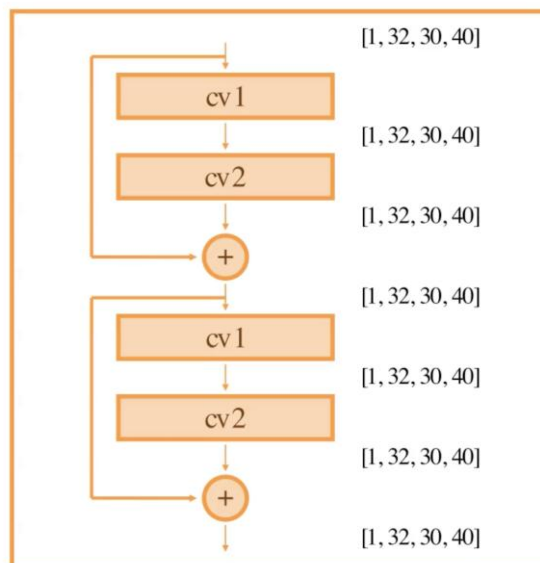


Figure 12.

7.1. Objective

The C3k2 (T) block is the heavier variant of the C3k2 module, designed to deepen feature refinement via C3k modules with nested bottlenecks. By applying multi-stage transformations within each partition of the input, it increases the richness and diversity of feature representations. This

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

module extends the C2f design with a customizable double-bottleneck structure, allowing more sophisticated hierarchical feature processing while maintaining computational efficiency.

Inside the C3k2 (T) module, the C3k block inside excels in several key areas:

- Hierarchical compression (squeezing features): The initial convolutions reduce the input channels and partition features into smaller, more manageable subsets. This hierarchical compression retains the most critical information, optimizing for both efficiency and diversity of feature representation.
- Multi-stage processing within C3k: Each subset undergoes further refinement through a series of nested Bottleneck blocks. These blocks sequentially transform the compressed features to emphasize core patterns while discarding redundancies.
- Final expansion and aggregation: The outputs of the bottleneck blocks are recombined and expanded through concatenation and the final convolution. This phase balances dimensionality and feature richness, ensuring the network is prepared for subsequent stages.
- Promoting feature diversity and refinement: By incorporating multiple convolutional paths and iterative processing, the C3k block enhances the diversity of extracted patterns. This design ensures that both fine-grained and broader structural features are effectively captured.
- Scalability: In YOLO26, C3k2(T) scales its internal depth with model size: nano/small/medium use one C3k, while large/extra-large use two C3k blocks in series between split and concat (each C3k contains two Bottleneck blocks).

C3k2(T) inherits from the C2f block (like C3k2(F)) but introduces hierarchical feature refinement via the C3k sub-block. Instead of the standard Bottleneck used in C3k2(F), it employs a C3k module, a C3-style block with three convolutions and an internal double-Bottleneck design, enabling deeper feature refinement within each split partition.

7.2. Flow

The operations within the C3k2 (T) module are as follows:

1. Initial convolution (cv1): 1×1 convolution projects the input tensor for partitioned processing.
2. Partitioning: Tensor is split along the channel dimension into two partitions: identity path and processed path.
3. C3k path (c3k=True): The processed partition is passed through a C3k block with an internal double-Bottleneck design that uses 3×3 kernels to enhance spatial feature extraction and refinement.
4. Concatenation: Merge input paths (y[0] and y[1]) and the C3k output along channels.
5. Final convolution (cv2): 1×1 convolution projects concatenated tensor to the desired number of channels for downstream modules.

8. C3k2 Module Variants: Architecture Placement and Design Trade-Offs

As we have seen, the C3k2 module in YOLO26 supports two configurations via the c3k flag. The lightweight path (c3k=False) contains a single bottleneck block, providing low computational cost and capturing shallow residual context, making it suitable for early-stage processing with large feature maps. The heavier path (c3k=True) incorporates a C3 module with nested Bottlenecks, enabling deeper, nonlinear transformations and richer feature representations for later stages. In short, c3k=False prioritizes efficiency, while c3k=True emphasizes representational capacity.

As shown in Figure 13, the placement of C3k2 variants reflects a deliberate, stage-wise allocation of computational resources, guided by feature map size, computational cost, and semantic requirements.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

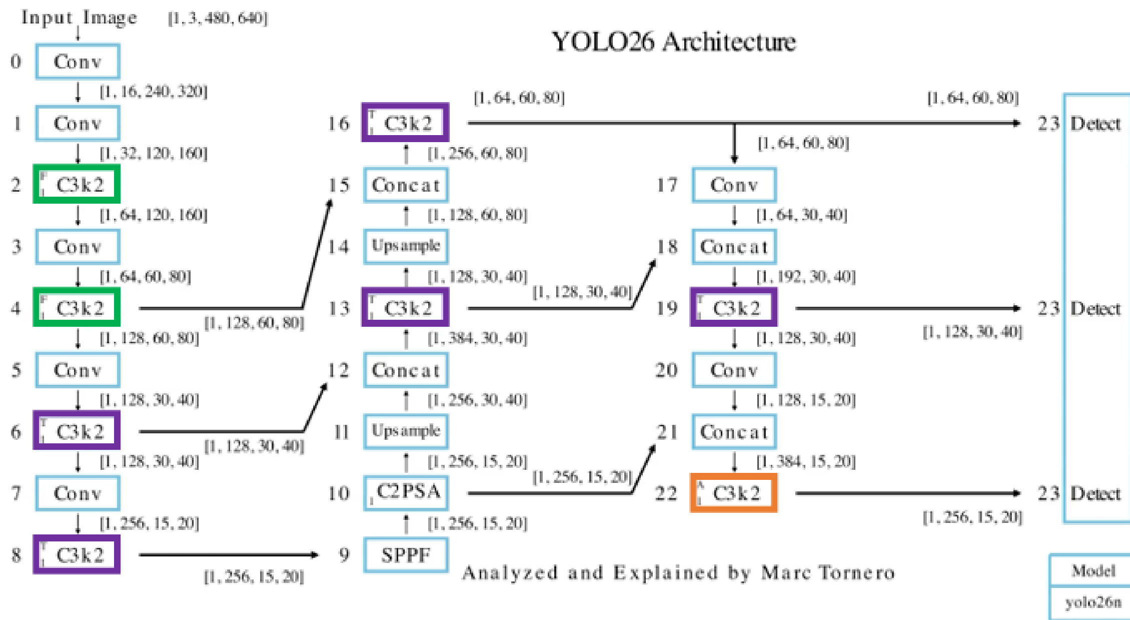


Figure 13.

- Early backbone stages (Stage 2 and 4 in green): c3k=False is used since feature maps are large (high spatial resolution). Lightweight bottlenecks capture low-level features such as edges and textures while keeping computation low.
- Deeper backbone stages (Stage 6 and 8 in purple): c3k=True is employed as feature maps become smaller (lower spatial resolution, higher channel depth). Heavier processing enables deeper, nonlinear transformations that capture object parts and semantic patterns.
- Neck layers (Fusion Stage 13 and Final Stages 16 (P3) and 19 (P4)): The fusion stage and the final P3 and P4 stages use c3k=True to efficiently integrate multi-scale features.

The Final Stage 22 in orange (P5) employs a special module newly introduced in YOLO26. We discuss this module later, after the C2PSA module, because it shares similar attention components.

The alternating use of c3k=False and c3k=True balances efficiency and expressiveness: using only c3k=False favors speed but limits high-level feature extraction, reducing accuracy for complex objects, whereas using only c3k=True enhances accuracy but significantly increases inference latency in early stages with large feature maps.

YOLO26 balances these considerations by employing lightweight blocks in early layers for efficiency, heavier blocks in deeper layers for richer feature extraction, and in the neck for strong semantic reasoning. This placement aligns with general CNN design principles, prioritizing efficiency when feature maps are large and capacity when spatial resolution is low, optimizing the network's speed-accuracy trade-off.

To justify the deliberate selection of C3k2 configurations, we train three YOLO26n variants from scratch for up to 3000 epochs on the COCO dataset: (1) the original architecture, which employs a mixture of c3k=False and c3k=True settings; (2) a variant in which all C3k2 modules use c3k=False; and (3) a variant in which all C3k2 modules use c3k=True. Stage 22 is kept unmodified in these experiments, as its design is examined separately later in the paper.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

 doi:10.20944/preprints202603.2518.v1

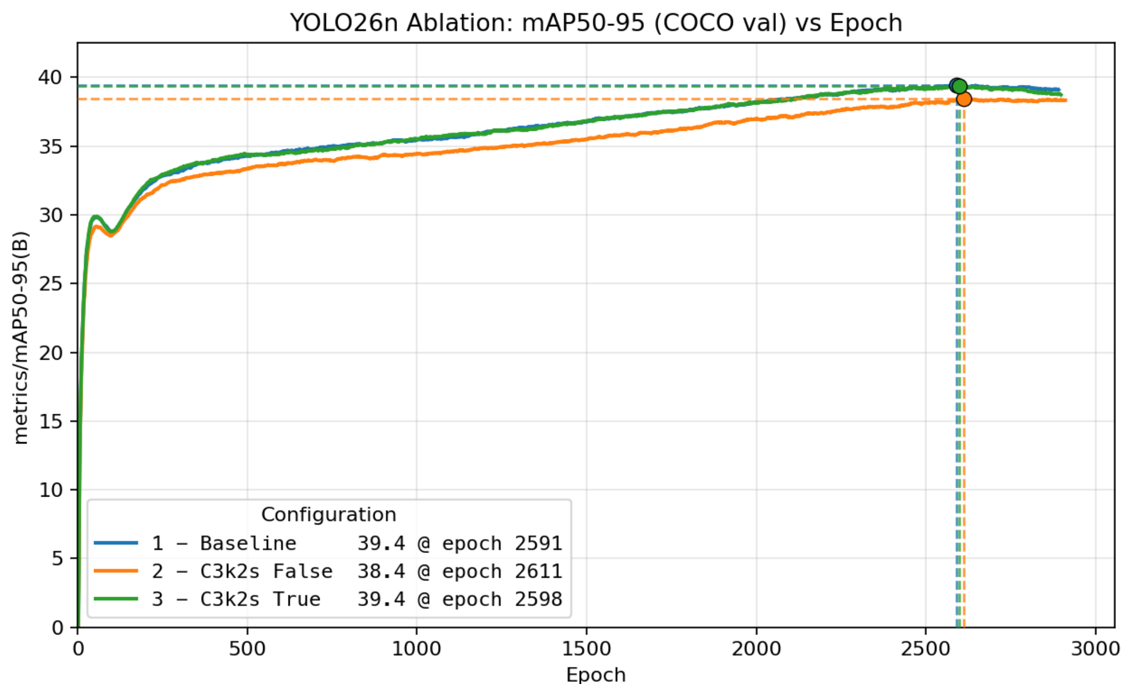


Figure 14.

Benchmarking on TensorRT 10 (FP16) with an H100 GPU (Table 2) confirms the accuracy–latency trade-off introduced by the C3k2 configuration. Model 3 (all C3k2 modules set to True) achieves very similar accuracy to the baseline (0.3930 vs. 0.3933 mAP@0.5:0.95), but incurs the highest latency (1.11 ms), making it a net regression in efficiency. Conversely, Model 2 (all C3k2 modules set to False) is the fastest configuration (0.86 ms), but suffers a from an accuracy drop to 0.3813 mAP@0.5:0.95.

Overall, Model 1 (the default YOLO26n mix of C3k2 False and True) provides the best accuracy–speed balance. It preserves the highest accuracy while avoiding the additional runtime cost observed when enabling C3k2 True throughout the network. These results indicate that setting all C3k2 modules to True provides no measurable accuracy benefit yet increases latency, whereas retaining the mixed design—using the more expensive configuration only where it is most impactful—yields the most practical trade-off.

Table 2.

Architecture	mAP@0.5:0.95	Latency (ms)
1 – Baseline	**0.3933** ☑	**0.99**
2 – C3k2s False	0.3813	0.86 ☑
3 – C3k2s True	0.3930	1.11

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 April 2026 doi:10.20944/preprints202603.2518.v1

9. SPPF Module (Spatial Pyramid Pooling Fast)

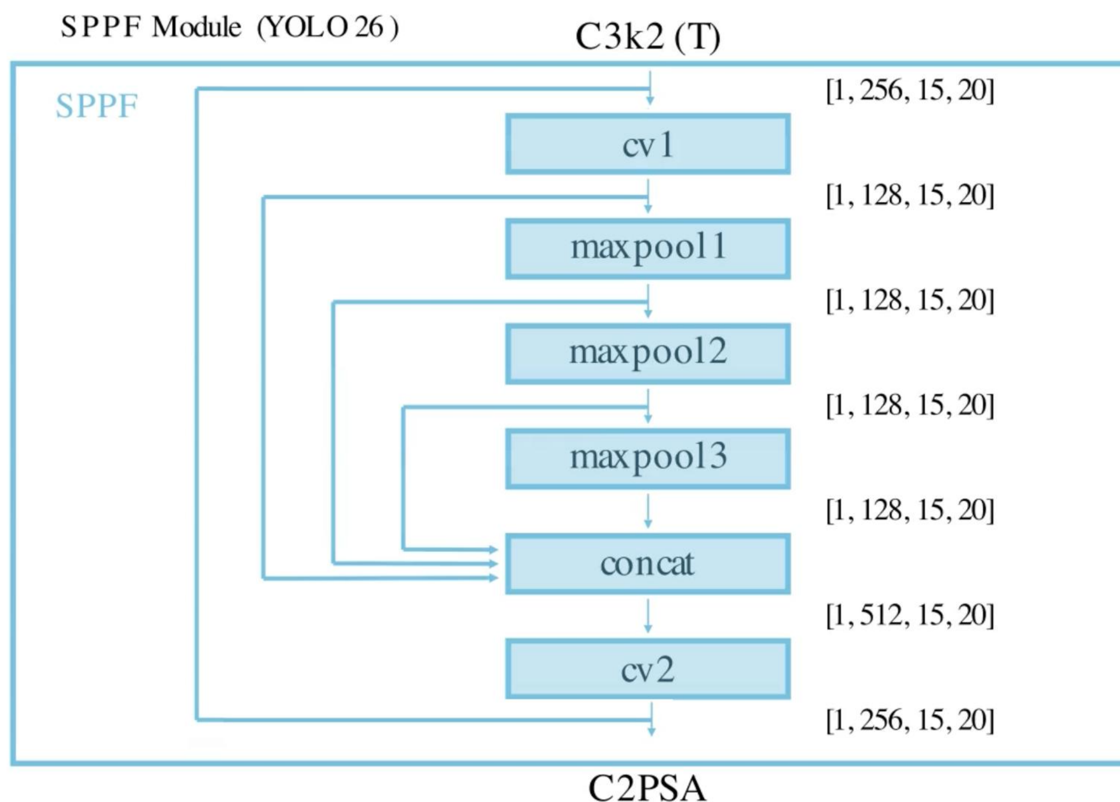


Figure 15.

9.1. Objective

The SPPF block enriches deep features with multi-scale spatial context. Its primary goals are:

- Multi-scale feature aggregation: Applies three max-pooling operations (kernel size=5) to capture spatial information at multiple scales, combining fine details and broad context.
- Feature fusion: Concatenates outputs from pooling operations to create a rich, multi-scale feature map, enhancing the network's ability to detect objects of varying sizes.
- Efficient downsampling: Preserves spatial relationships while reducing resolution, ensuring compact and meaningful feature representation.
- Optimized design: Streamlines traditional SPP, reducing computations while maintaining scalability for real-time applications.

SPPF was introduced in YOLOv5 as a faster alternative to the original SPP module (used in YOLOv5 first releases). Subsequent YOLO architectures, including YOLOv6 [6], YOLOv8 [8], YOLOv10 [10], and YOLO11 [11].

YOLO26 adopted the SPPF as the standard pooling module due to its combination of accuracy and efficiency but added the residual connection from the input to the output, not used in previous versions. They also hardcoded the two convolutions (cv1, and cv2) to not have activation (act=False), unlike previous versions which did have it (act=True).

9.2. Flow

1. Initial convolution (cv1): A 1×1 convolution reduces the number of channels to prepare the input for multi-scale pooling.
2. Multi-scale max-pooling: Three parallel max-pooling operations (kernel size = 5) capture features at different receptive fields while preserving spatial relationships.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

3. Concatenation: Outputs of all pooling operations and the initial convolution are concatenated along the channel dimension, producing a multi-scale feature map.
4. Final convolution (cv2): A 1×1 convolution projects the concatenated feature map back to the desired number of channels, creating a compact, enriched representation for downstream processing.

To analyze the contribution of the SPPF module and evaluate its main design choices, we train five YOLO26n variants from scratch for 3000 epochs on the COCO dataset. Among these, three differ only in the max-pooling kernel size used in the SPPF module, one omits the SPPF module entirely, and one disables the newly introduced residual connection. The following figure shows the five configurations: (1) default kernel_size=5, (2) no SPPF module, (3) no shortcut, (4) kernel_size=3, and (5) kernel_size=7.

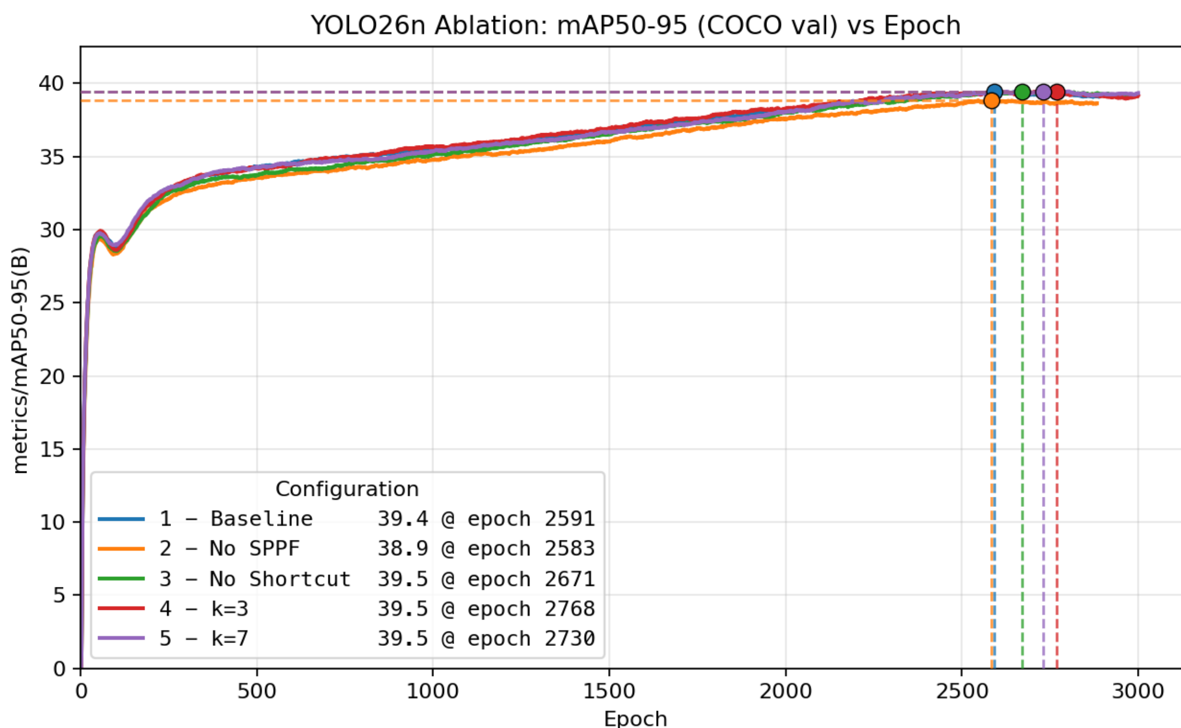


Figure 16.

Benchmarking the five SPPF configurations with TensorRT 10 (FP16) on an H100 GPU (Table 3) highlights a nuanced accuracy–latency trade-off. The baseline configuration (Model 1, SPPF with k=5) delivers strong overall performance, achieving 0.3933 mAP@0.5:0.95 with a latency of 0.99 ms. Removing the SPPF module entirely (Model 2) yields the lowest latency (0.96 ms), confirming that eliminating this block improves inference speed; however, it also produces the lowest accuracy (0.3866 mAP@0.5:0.95), indicating that the loss of multi-scale feature aggregation degrades detection quality.

Among the tested variants, Model 3 (No Shortcut) achieves the highest mAP@0.5:0.95 (0.3941), but also incurs the highest latency (1.01 ms). This makes it the most accuracy-focused option, although the accuracy gain over the baseline is small and comes at a modest runtime cost. Model 5 (k=7) provides the most favorable overall trade-off: it attains 0.3935 mAP@0.5:0.95, essentially matching—and slightly exceeding—the baseline, while also reducing latency to 0.98 ms. In this sense, k=7 stands out as a particularly attractive alternative, since it improves efficiency without sacrificing accuracy.

By contrast, Model 4 (k=3) slightly reduces accuracy (0.3922 mAP@0.5:0.95) while also increasing latency (1.00 ms), offering no clear practical advantage over the baseline.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Overall, these results suggest that removing SPPF is beneficial only when minimizing latency is the primary objective, whereas increasing the kernel size to k=7 yields the best balance between detection performance and inference speed. If maximum accuracy is prioritized above all else, the No Shortcut variant is the strongest option.

Table 3.

Architecture	mAP@0.5:0.95	Latency (ms)
1 – Baseline	**0.3933**	**0.99**
2 – No SPPF	0.3866	0.96
3 – No Shortcut	**0.3941** ☑	**1.01**
4 – k=3	0.3922	1.00
5 – k=7	**0.3935**	**0.98**

10. C2PSA Module (Cross Stage Partial with Position-Sensitive Attention)

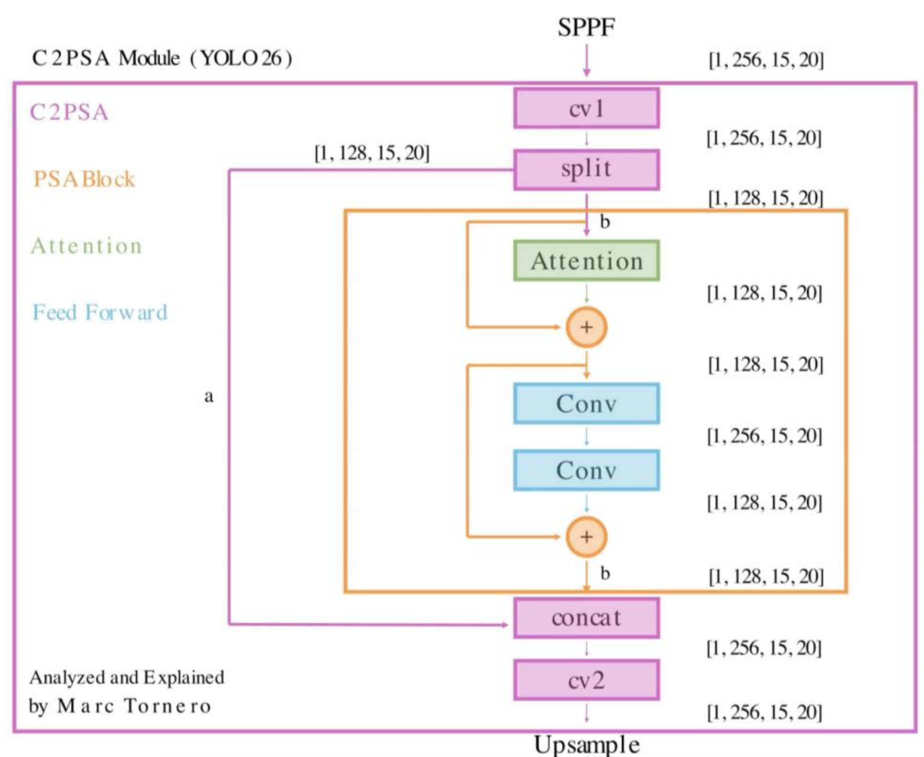


Figure 17.

10.1. Objective

The C2PSA (Cross-Stage Partial with Position-Sensitive Attention) module is a key block in YOLO26, positioned after the SPPF and serving as the transition between the backbone and the neck. Its primary function is to enrich feature representations by combining convolutional processing with advanced attention mechanisms.

The C2PSA enhances feature extraction and processing through several complementary mechanisms:

- Dual-path processing: Input features are split into two pathways. One path undergoes direct convolutional processing to preserve local details, while the other applies attention-based transformations via PSABlock modules to capture long-range dependencies.
- Attention mechanisms: Each PSABlock leverages multi-head self-attention to model relationships between distant spatial locations, making the network more effective at handling complex and distributed object patterns.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 April 2026 doi:10.20944/preprints202603.2518.v1

- Spatial awareness: Position-sensitive encodings are incorporated to preserve relative spatial arrangements, strengthening localization accuracy.
- Feature refinement: Lightweight feed-forward layers within the PSABlock refine attended features, ensuring efficient propagation and richer semantic context.
- Feature fusion: Outputs from convolutional and attention pathways are merged, resulting in more expressive feature maps that balance local detail with global context.
- Scalability: Unlike YOLOv10 [10], where the PSA module was restricted to a single Attention + Feed Forward structure, YOLO26's C2PSA allows multiple PSABlocks to be stacked. Smaller versions (nano, small, medium) contain one PSABlock, while larger models (large, extra-large) contain two in sequence.

By integrating convolutional and attention-driven processing, the C2PSA module establishes attention as a central component in YOLO26, improving the discriminative power of the backbone–neck interface while preserving computational efficiency.

The standard attention mechanism, introduced in the original Transformer paper [34], forms the basis for many modern models. Figure 18 illustrates this mechanism.

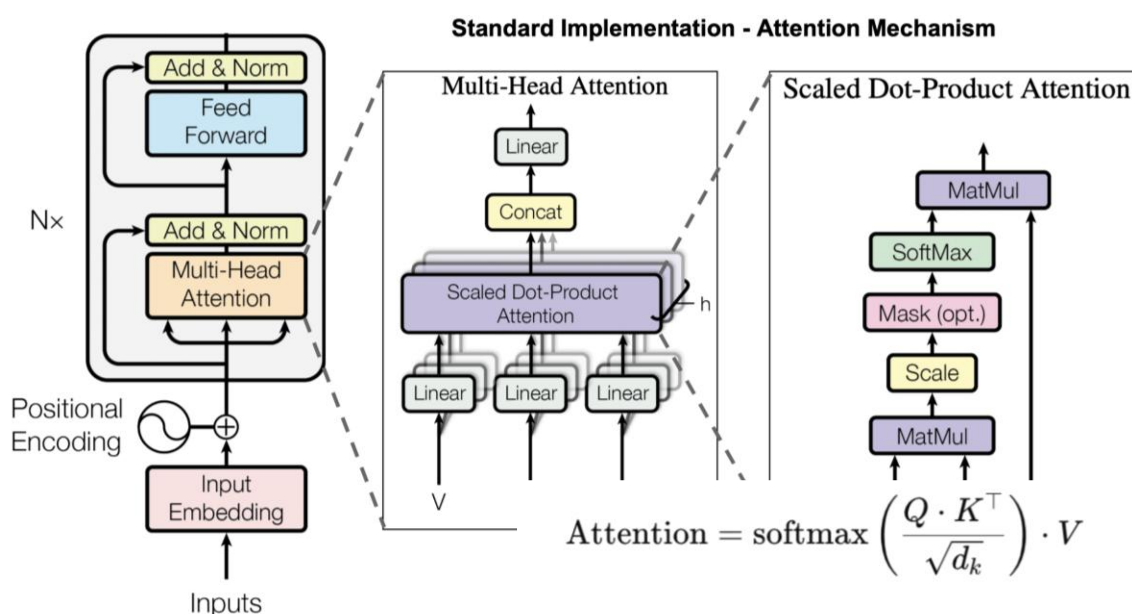
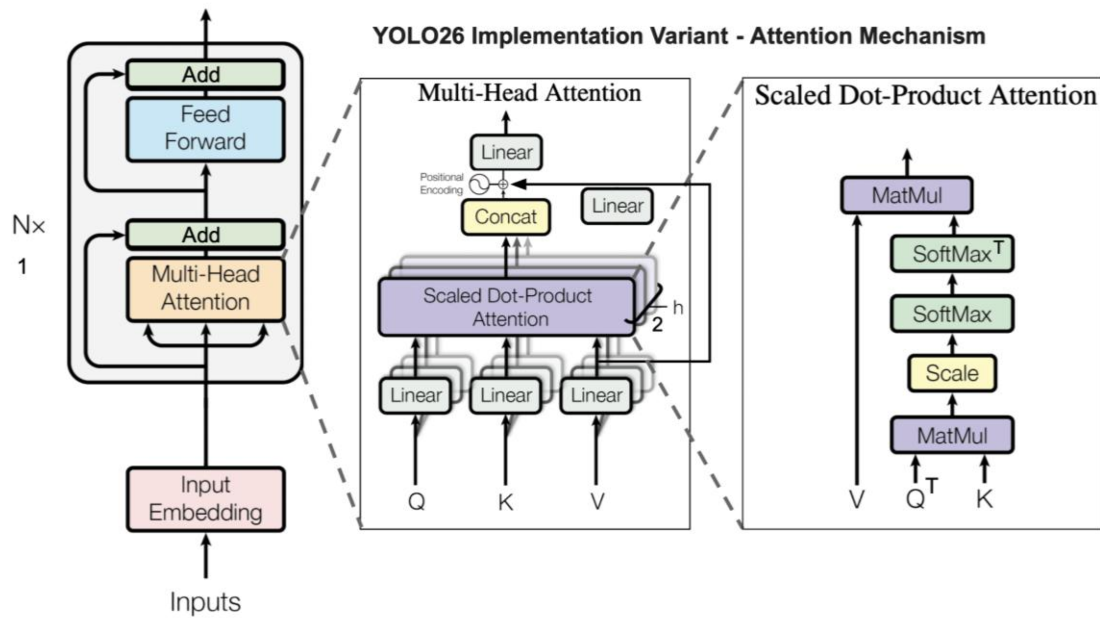


Figure 18.

As seen in Figure 19, YOLO26 adopts a variant of this mechanism. Although mathematically similar to the standard method, this variation is optimized for vision tasks and requirements within the YOLO architecture, such as improving computational efficiency and simplifying implementation.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 April 2026 doi:10.20944/preprints202603.2518.v1



$$\text{Attention} = V \cdot \left(\text{softmax} \left(\frac{Q^{\top} \cdot K}{\sqrt{d_k}} \right) \right)^{\top}$$

Figure 19.

10.2. Flow

- Initial convolution (cv1): A 1×1 convolution preprocesses the input tensor, decoupling the module's internal operations from preceding feature representations while maintaining spatial resolution.
- Split: The tensor is partitioned along the channel dimension into two branches: (a) a skip path that preserves identity features for later concatenation, and (b) a processed path that passes through the PSA block(s) for attention-based refinement.
- PSA block(s) for multi-head attention:

Queries, keys, and values are computed using a single 1×1 convolution for efficiency. This step applies a linear, activation-free projection that maps features into a new representation space while avoiding additional nonlinearity that may hinder optimization.

The channels are then split into multiple attention heads, allowing each head to learn different spatial relationships.

Scaled dot-product attention is computed to capture global similarity across spatial positions.

Softmax normalization is applied to produce interpretable attention weights.

A weighted sum of the value vectors is then generated, producing context-aware features enriched with global dependencies.

Positional encoding: A depthwise convolution [28] is applied to the value tensor to introduce spatial awareness. The resulting positional information is added element-wise to the attention output, making the representation position-sensitive without overwhelming the original features.

- Feed-Forward Network (FFN):

Applies a position-wise transformation to each spatial location independently, where 1×1 convolutions adjust channel representations without mixing information across neighboring pixels.

Uses an expand–compress design to enrich feature representations efficiently.

Incorporates skip connections to preserve original context and avoid over-smoothing.

- Concatenation and final convolution (cv2):

Merges the skip path (a) with the processed path (b) from the PSA block(s).

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 April 2026 doi:10.20944/preprints202603.2518.v1

Final 1×1 convolution stabilizes the combined features and projects them into a compact representation suitable for downstream tasks.

To analyze the contribution of the C2PSA module and evaluate its main design choices, we train six YOLO26n variants from scratch for 3000 epochs on the COCO dataset. The following Figure 20 presents the six configurations: (1) the standard YOLO26n model; (2) a variant with the attention ratio set to 0.25 instead of the default 0.5; (3) a variant with the attention ratio set to 0.75; (4) a variant with the attention ratio set to 1.0; (5) a variant in which the PSABlock shortcut connection is disabled; and (6) a variant in which the C2PSA module is removed entirely.

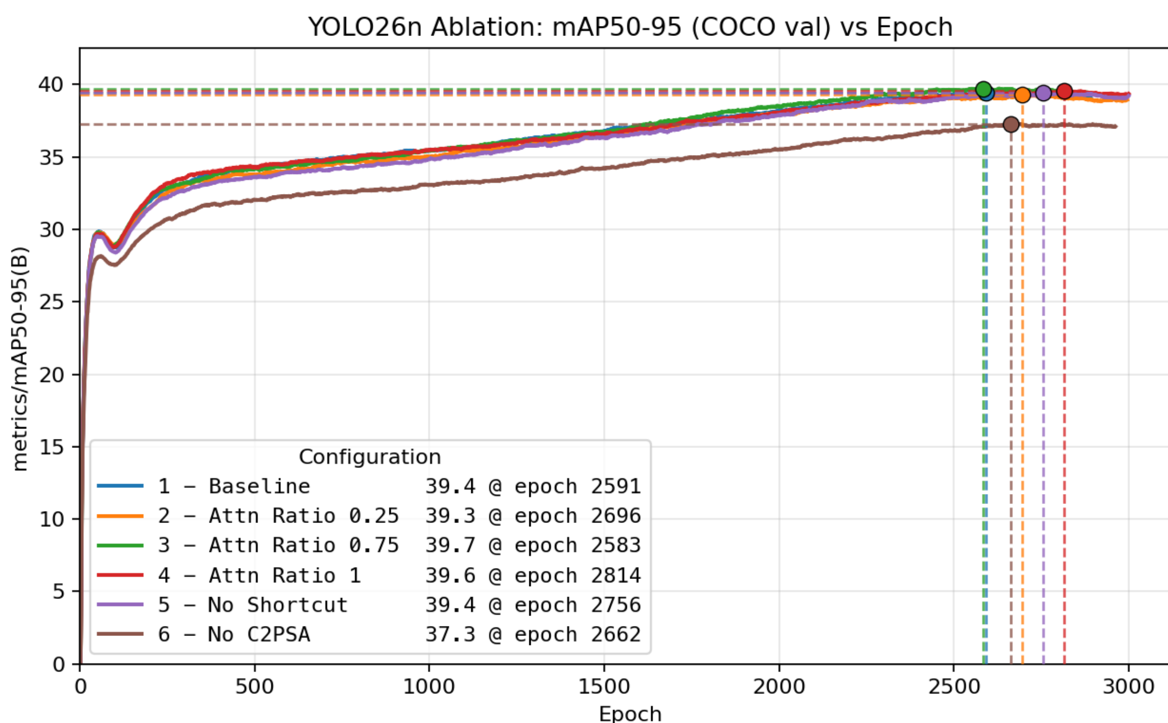


Figure 20.

Benchmarking the six configurations on TensorRT 10 (FP16) with an H100 GPU (Table 4) shows that changes to the C2PSA design affect accuracy more than latency. The baseline configuration (Model 1, attention ratio 0.5) achieves 0.3933 mAP@0.5:0.95 with a latency of 0.99 ms, providing strong overall performance. Reducing the attention ratio to 0.25 (Model 2) lowers accuracy to 0.3909 while maintaining the same latency, indicating that decreasing the attention capacity is not beneficial in this setting.

Increasing the attention ratio to 0.75 (Model 3) yields the highest mAP@0.5:0.95 (0.3961) with no latency penalty relative to the baseline, making it the most favorable configuration among those tested. This result suggests that Model 3 is a strong alternative to the baseline, as it improves detection accuracy while preserving the same inference time. Setting the attention ratio to 1.0 (Model 4) slightly reduces accuracy compared with Model 3, but still outperforms the baseline at the same 0.99 ms latency.

Model 5 (No Shortcut) produces exactly the same mAP@0.5:0.95 and latency as the baseline, suggesting that removing the shortcut has no measurable effect on performance in this experiment. Finally, Model 6 (No C2PSA) is the fastest configuration at 0.92 ms, but it also yields by far the lowest accuracy (0.3704 mAP@0.5:0.95). This indicates that although removing C2PSA improves speed, it substantially weakens detection performance.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Overall, Model 3 provides the best accuracy–latency trade-off, since it achieves the highest accuracy without increasing inference time. By contrast, Model 6 is preferable only when minimizing latency is the dominant objective and the corresponding loss in accuracy is acceptable.

Table 4.

Architecture	mAP@0.5:0.95	Latency (ms)
1 – Baseline	**0.3933**	**0.99**
2 – Attn Ratio 0.25	0.3909	0.99
3 – Attn Ratio 0.75	**0.3961 ☑**	**0.99**
4 – Attn Ratio 1	0.3940	0.99
5 – No Shortcut	0.3933	0.99
6 – No C2PSA	0.3704	0.92

11. Upsample and Concatenation Layers

11.1. Upsample Layer

11.1.1. Objective

Increase spatial resolution for multi-scale feature fusion. Enhances resolution for tasks like object detection, improving localization accuracy. Prepares feature maps for concatenation with higher-resolution maps from earlier layers.

- Spatial Restoration: Doubles height and width using nearest-neighbor interpolation
- Lightweight: No learnable parameters; efficient for real-time systems.

11.1.2. Flow

1. Scale Factor: Increases the spatial dimensions of the input feature map by a factor of 2, allowing finer spatial detail recovery.
2. Interpolation Mode: Uses nearest-neighbor interpolation to replicate pixel values efficiently without adding computational complexity.

11.2. Concat Layer

11.2.1. Objective

Merge feature maps from different stages or scales. Allows the network to leverage complementary information from multiple stages. Improves the model's capacity to capture patterns at different spatial scales. Essential for multi-scale feature decoding in detection heads or other downstream tasks.

- Feature Reuse: Combines low- and high-level features to enrich representations.
- Channel-Wise Fusion: Increases diversity of feature channels.
- Supports Skip Connections: Enables integration of features from earlier layers and multi-scale decoding

11.2.2. Flow

1. Tensor Concatenation: Merges a list of tensors along a specified dimension (default: channel dimension)
2. Flexible Input: Can combine features from both current and previous layers, supporting complex network architectures.

To analyze the effect of the upsampling strategy and concatenation operations, we train four YOLO26n variants from scratch for 3000 epochs on the COCO dataset. Three variants differ only in the interpolation method used for upsampling: (1) nearest neighbor, (2) bilinear, and (3) bicubic. All three achieve similar mAP@0.50:0.95. In the fourth experiment, the concatenation layers are removed and replaced with identity layers to examine the effect of eliminating feature fusion.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

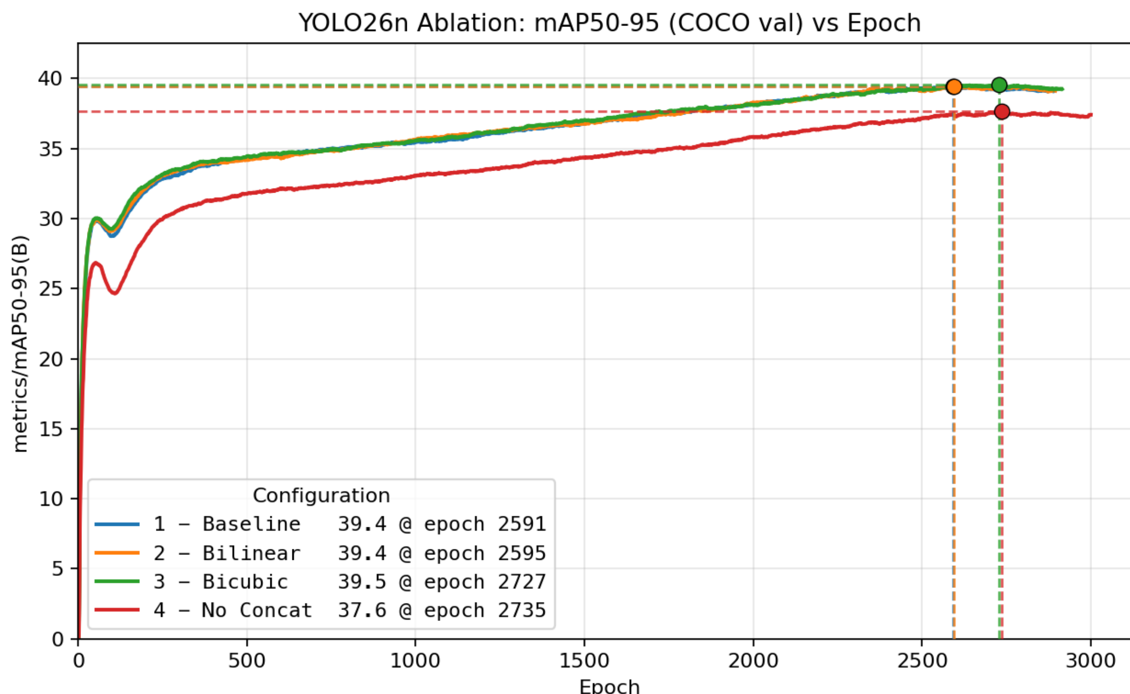


Figure 21.

Benchmarking the different interpolation methods on TensorRT 10 (FP16) with an NVIDIA H100 GPU (Table 5) shows that interpolation choice has a measurable impact on both accuracy and latency. The baseline configuration (Model 1, nearest interpolation) achieves 0.3933 mAP@0.5:0.95 with a latency of 0.99 ms, providing solid overall performance. Replacing nearest interpolation with bilinear interpolation (Model 2) yields the best overall result, achieving the highest mAP@0.5:0.95 (0.3954) while also slightly reducing latency to 0.98 ms. This makes bilinear a clear improvement over the baseline in both accuracy and efficiency.

Bicubic interpolation (Model 3) also improves accuracy relative to the baseline, reaching 0.3950 mAP@0.5:0.95, but incurs the highest latency among the interpolation variants at 1.00 ms. Although its accuracy remains close to that of bilinear, it offers no latency advantage, making bilinear the more practical choice of the two. Therefore, among the tested interpolation methods, bilinear provides the most favorable accuracy–latency trade-off.

Finally, removing the concatenation layers (Model 4) slightly reduces latency to 0.97 ms, but causes a substantial drop in accuracy to 0.3743 mAP@0.5:0.95. This indicates that concatenation is important for preserving detection performance, and that eliminating it yields only marginal speed benefits at a significant cost in accuracy.

Table 5.

Architecture	mAP@0.5:0.95	Latency (ms)
1 – Baseline	0.3933	0.99
2 – Bilinear	**0.3954** ☑	**0.98** ☑
3 – Bicubic	0.3950	1.00
4 – No Concat	0.3743	0.97

12. Feature Refinement Module (C3k2 with Attention)

12.1. Objective

The C3k2 (attn = True) module is a hybrid feature refinement block in the YOLO26 architecture that combines the efficient partial-channel processing of the C3k2 module with the attention-based

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 April 2026 doi:10.20944/preprints202603.2518.v1

contextual modeling of the C2PSA module. Its main purpose is to improve feature quality by jointly capturing:

- Local structural patterns through Bottleneck-based convolutional refinement
- Long-range spatial dependencies through the PSA Block
- Feature preservation through CSP-style split pathways
- Computational efficiency by applying the most expensive operations to only part of the channels.

Like the standard C3k2 design, the module first divides the feature map into partial paths so that some information is preserved with minimal transformation. However, unlike the C3k2 (False) variant, the transformed branch does not stop at a Bottleneck. Instead, it is further refined by a PSA Block, allowing the module to combine compact convolutional processing with attention-guided enhancement.

This gives the module several important advantages:

- Partial feature processing: Only a subset of channels is heavily processed, reducing redundancy and improving efficiency.
- Local refinement through the Bottleneck: The Bottleneck compresses, processes, and restores channels, helping the network emphasize edges, corners, textures, and other localized patterns.
- Global context through attention: The PSA Block enriches the refined features with broader spatial relationships, making the representation more aware of distributed object structure.
- Multi-level feature retention: The module preserves an untouched split branch, the pre-attention processed branch, and the fully refined branch, enabling richer feature fusion.
- Improved optimization: Residual connections in both the Bottleneck and the PSA Block help maintain information flow and stabilize training.

Overall, the C3k2 (attn = True) module can be viewed as a CSP-style multi-path refinement block that first performs compact local convolutional enhancement and then applies attention-based semantic refinement, producing features that are both efficient and highly expressive.

13.2. Flow

- Initial convolution (cv1): A 1×1 convolution is first applied to the input tensor. As seen in Figure 22, the feature map remains at [1, 256, 15, 20], meaning spatial resolution is preserved while the channels are prepared for internal processing.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Figure 22.

- Split: The output of the initial convolution is divided along the channel dimension into two equal parts.
- The y[0] branch serves as a preserved shortcut branch that bypasses the heavier transformations and is sent directly toward the final concatenation. The y[1] branch is used as the main refinement path.
- Bottleneck refinement: The y[1] branch first passes through a Bottleneck block composed of two convolutions. The first convolution reduces channels from 128 to 64, creating a compressed intermediate representation, the second convolution restores channels from 64 back to 128.
- A residual connection then adds the original y[1] back to the transformed output. This Bottleneck stage allows the module to refine local patterns efficiently while preserving the original feature information.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 April 2026 doi:10.20944/preprints202603.2518.v1

- PSA Block refinement: The output of the Bottleneck is then passed into a PSA Block, which further enhances the features using attention-based processing.
- Inside this PSA Block, an Attention submodule captures broader spatial dependencies and contextual relationships, its output is added residually to the incoming feature map, the result then passes through a lightweight two-layer convolutional feed-forward subnetwork. As seen in Figure 22, this subnetwork expands channels from 128 to 256 and then projects them back from 256 to 128, a second residual addition is applied after this feed-forward stage. The PSA Block therefore refines the Bottleneck output by incorporating global context while still preserving the original branch information through internal skip connections.
- Refined output (y[2]): After the Bottleneck and PSA Block, the final processed branch becomes y[2].
- Concatenation: Three tensors are concatenated along the channel axis: y[0] the preserved shortcut branch, y[1] the original split refinement branch, and y[2] the fully refined branch after Bottleneck + PSA processing. This concatenation is important because it preserves: raw partial features, intermediate branch features, and deeply refined attention-enhanced features.
- Final convolution (cv2): A final 1×1 convolution fuses the concatenated tensor and projects it from 384 channels back to 256 channels, producing the final output of shape [1, 256, 15, 20]. This final step learns inter-channel relationships across all three branches and generates a richer feature representation suitable for the next stage of the network.

To justify the introduction of the new C3k2 configuration in YOLO26, we train three YOLO26n variants from scratch for up to 3000 epochs on the COCO dataset: (1) the original architecture, which employs the new C3k2 configuration with attn=True; (2) a variant with C3k2 set to c3k=False, as examined earlier; and (3) a variant with C3k2 set to c3k=True.

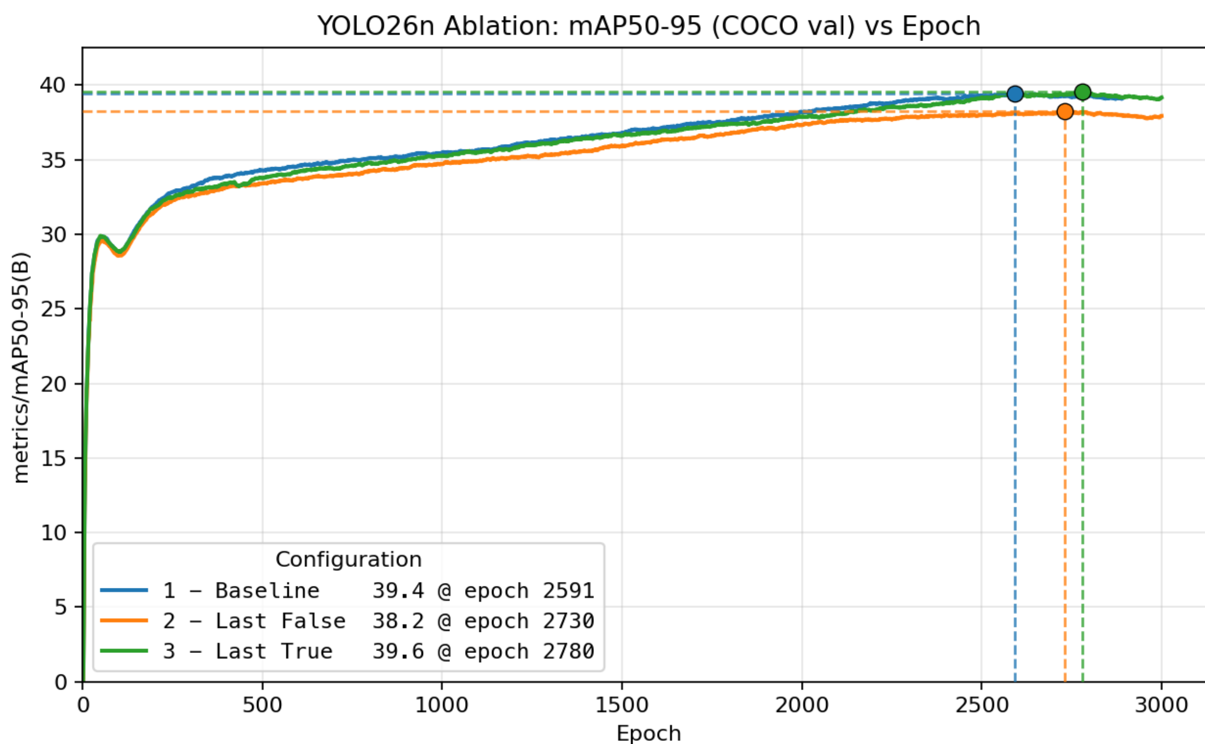


Figure 23.

Benchmarking the final C3k2 module on TensorRT 10 (FP16) with an NVIDIA H100 GPU (Table 6) shows a clear accuracy–latency trade-off. The baseline configuration (Model 1, with attn=True) achieves the highest mAP@0.5:0.95 (0.3933), but also has the highest latency (0.99 ms). This indicates that the attention-enabled version provides the best detection performance, although at a modest runtime cost.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Model 2 (attn=False, c3k=False), corresponding to the C3k2 False variant in the final stage, is the fastest configuration at 0.94 ms, but also produces the lowest accuracy (0.3819 mAP@0.5:0.95). Therefore, although this design offers the greatest speed improvement, it does so at a substantial cost in accuracy.

Model 3 (attn=False, c3k=True), which is effectively equivalent to using the C3k2 True variant in the last stage, reduces latency to 0.97 ms while maintaining a relatively high accuracy of 0.3921 mAP@0.5:0.95. This makes it a more balanced alternative than Model 2, since it improves efficiency with only a small reduction in detection performance compared with the baseline.

Overall, the baseline model remains the best accuracy-focused choice, while Model 3 offers the most practical compromise between accuracy and latency. Model 2 is preferable only when minimizing inference time is the primary objective. In this sense, the results support the introduction of the attn flag in YOLO26, as the added bottleneck + PSA structure appears to improve mAP@0.5:0.95 relative to the other C3k2 alternatives.

Table 6.

Architecture	mAP@0.5:0.95	Latency (ms)
1 – Baseline	**0.3933** ☑	**0.99**
2 – Last False	0.3819	0.94 ☑
3 – Last True	0.3921	0.97

16. Conclusion

This paper presented a block-level, modular analysis of YOLO26n, clarifying the objective, internal flow, and representational transformations of its primary components—from convolutional stems and bottleneck-style refinement blocks to multi-scale aggregation, and attention mechanisms. By decomposing the network into interpretable building blocks and explicitly tracking tensor shape evolution, we provide a practical technical reference that connects architectural design intent to observable feature transformations.

In addition to architectural interpretation, we introduced a controlled ablation suite designed to quantify the marginal contribution of key design choices under a fixed, fully specified training and benchmarking protocol. Across ablations spanning activation functions, C3k2 variants, SPPF settings, and attention module configurations, we measured the impact of each change on COCO mAP50–95 and inference latency, reporting absolute performance to a baseline. This isolates which modules provide meaningful accuracy–efficiency gains and which offer limited return under the tested constraints, enabling evidence-based guidance for practitioners deciding what to keep, simplify, or replace in compute-limited deployments.

To avoid conflating architectural effects with recipe and data effects, our experiments train from scratch on MS COCO train2017 and evaluate on val2017, and we explicitly note that some Ultralytics-reported COCO benchmarks rely on external-data pretraining (e.g., Objects365) prior to COCO fine-tuning. Consequently, the goal of this work is not to reproduce or surpass headline benchmark numbers, but to provide a fair, reproducible comparison of architectural variants under identical conditions and a transparent methodology that others can extend.

The analysis workflow used here—Objective → Flow, supported by tensor tracking and standardized ablations—generalizes beyond YOLO26n and can be applied to future YOLO releases or other real-time detectors. As models continue to evolve rapidly, such structured attribution studies can help the community understand which architectural changes truly drive progress, and where future innovation is most likely to yield substantial improvements.

Appendix A. Training Parameters for the YOLO26n Ablation Studies

Ultralytics 8.4.6 🚀 Python-3.10.12 torch-2.4.1+cu124
CUDA:0 (NVIDIA H100 80GB HBM3, 81110MiB)

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 April 2026 doi:10.20944/preprints202603.2518.v1

CUDA:1 (NVIDIA H100 80GB HBM3, 81110MiB)
CUDA:2 (NVIDIA H100 80GB HBM3, 81110MiB)
CUDA:3 (NVIDIA H100 80GB HBM3, 81110MiB)
CUDA:4 (NVIDIA H100 80GB HBM3, 81110MiB)
CUDA:5 (NVIDIA H100 80GB HBM3, 81110MiB)
CUDA:6 (NVIDIA H100 80GB HBM3, 81110MiB)
CUDA:7 (NVIDIA H100 80GB HBM3, 81110MiB)

Table 7.

task: detect
mode: train
model: /ultralytics/ultralytics/cfg/models/26/yolo26n.yaml
data: /ultralytics/ultralytics/cfg/datasets/coco.yaml
epochs: 3000
time: null
patience: 300
batch: 320
imgsz: 640
save: true
save_period: -1
cache: false
device: 0,1,2,3,4,5,6,7
workers: 4
project: coco-train
name: coco-train-8gpu-3000-yolo26n
exist_ok: false
pretrained: false
optimizer: auto
verbose: true
seed: 0
deterministic: true
single_cls: false
rect: false
cos_lr: true
close_mosaic: 10
resume: false
amp: true
fraction: 1.0
profile: false
freeze: null
multi_scale: 0.0
compile: false
overlap_mask: true
mask_ratio: 4
dropout: 0.0
val: true
split: val
save_json: false
conf: null
iou: 0.7
max_det: 300

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 April 2026
doi:10.20944/preprints202603.2518.v1

half: false
dnn: false
plots: true
source: null
vid_stride: 1
stream_buffer: false
visualize: false
augment: false
agnostic_nms: false
classes: null
retina_masks: false
embed: null
show: false
save_frames: false
save_txt: false
save_conf: false
save_crop: false
show_labels: true
show_conf: true
show_boxes: true
line_width: null
format: torchscript
keras: false
optimize: false
int8: false
dynamic: false
simplify: true
opset: null
workspace: null
nms: false
lr0: 0.01
lrf: 0.01
momentum: 0.937
weight_decay: 0.0005
warmup_epochs: 3.0
warmup_momentum: 0.8
warmup_bias_lr: 0.1
box: 7.5
cls: 0.5
dfl: 1.5
pose: 12.0
kobj: 1.0
rle: 1.0
angle: 1.0
nbs: 64
hsv_h: 0.015
hsv_s: 0.7
hsv_v: 0.4
degrees: 0.0
translate: 0.1
scale: 0.5

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 April 2026 doi:10.20944/preprints202603.2518.v1

shear: 0.0
perspective: 0.0
flipud: 0.0
fliplr: 0.5
bgr: 0.0
mosaic: 1.0
mixup: 0.0
cutmix: 0.0
copy_paste: 0.0
copy_paste_mode: flip
auto_augment: randaugment
erasing: 0.4
cfg: null
tracker: botsort.yaml
save_dir: /ultralytics/coco-train/coco-train-8gpu-3000-yolo26

Appendix B. Benchmarking Parameters for the YOLO26n Ablation Studies

CUDA:0 (NVIDIA H100 80GB HBM3, 81110MiB)

Table 8.

model: best.pt
mode: coco.yaml
imgsz: 640
half: true
device: 0
batch: 1
dynamic: false
format: engine
deterministic: true

References

1. Lin, T.-Y.; Maire, M.; Belongie, S.; Bourdev, L.; Girshick, R.; Hays, J.; Perona, P.; Ramanan, D.; Zitnick, C.L.; Dollár, P. Microsoft COCO: Common Objects in Context. **2015**.
2. Redmon, J.; Divvala, S.; Girshick, R.; Farhadi, A. You Only Look Once: Unified, Real-Time Object Detection. **2016**.
3. Redmon, J.; Farhadi, A. YOLOv3: An Incremental Improvement. **2018**.
4. Bochkovskiy, A.; Wang, C.-Y.; Liao, H.-Y.M. YOLOv4: Optimal Speed and Accuracy of Object Detection. **2020**.
5. Khanam, R.; Hussain, M. What Is YOLOv5: A Deep Look into the Internal Features of the Popular Object Detector. **2024**.
6. Li, C.; Li, L.; Jiang, H.; Weng, K.; Geng, Y.; Li, L.; Ke, Z.; Li, Q.; Cheng, M.; Nie, W.; et al. YOLOv6: A Single-Stage Object Detection Framework for Industrial Applications. **2022**.
7. Wang, C.-Y.; Bochkovskiy, A.; Liao, H.-Y.M. YOLOv7: Trainable Bag-of-Freebies Sets New State-of-the-Art for Real-Time Object Detectors. **2022**.
8. Yaseen, M. What Is YOLOv8: An In-Depth Exploration of the Internal Features of the Next-Generation Object Detector. **2024**.
9. Wang, C.-Y.; Yeh, I.-H.; Liao, H.-Y.M. YOLOv9: Learning What You Want to Learn Using Programmable Gradient Information. **2024**.
10. Wang, A.; Chen, H.; Liu, L.; Chen, K.; Lin, Z.; Han, J.; Ding, G. YOLOv10: Real-Time End-to-End Object Detection. **2024**.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 April 2026 doi:10.20944/preprints202603.2518.v1

11. Khanam, R.; Hussain, M. YOLOv11: An Overview of the Key Architectural Enhancements. **2024**.
12. Tian, Y.; Ye, Q.; Doermann, D. YOLOv12: Attention-Centric Real-Time Object Detectors Latency (Ms) MS COCO MAP (%)., doi:10.0.
13. Lei, M.; Li, S.; Wu, Y.; Hu, H.; Zhou, Y.; Zheng, X.; Ding, G.; Du, S.; Wu, Z.; Gao, Y. YOLOv13: Real-Time Object Detection with Hypergraph-Enhanced Adaptive Visual Perception. **2025**.
14. Jiang, T.; Zhong, Y. ODverse33: Is the New YOLO Version Always Better? A Multi Domain Benchmark from YOLO v5 to V11. **2025**.
15. Terven, J.; Cordova-Esparza, D. A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS. **2024**, doi:10.3390/make5040083.
16. Sapkota, R.; Meng, Z.; Churuvija, M.; Du, X.; Ma, Z.; Karkee, M. Comprehensive Performance Evaluation of YOLOv12, YOLO11, YOLOv10, YOLOv9 and YOLOv8 on Detecting and Counting Fruitlet in Complex Orchard Environments. **2026**, doi:10.1016/j.agrcom.2026.100125.
17. Sapkota, R.; Cheppally, R.H.; Sharda, A.; Karkee, M. YOLO26: Key Architectural Enhancements and Performance Benchmarking for Real-Time Object Detection. **2026**.
18. Ramos, L.T.; Sappa, A.D. A Decade of You Only Look Once (YOLO) for Object Detection: A Review. *IEEE Access* 2025, *13*, 192747–192794.
19. Sapkota, R.; Flores-Calero, M.; Qureshi, R.; Badgujar, C.; Nepal, U.; Poulose, A.; Zeno, P.; Vaddevolu, U.B.P.; Khan, S.; Shoman, M.; et al. YOLO Advances to Its Genesis: A Decadal and Comprehensive Review of the You Only Look Once (YOLO) Series. *Artif. Intell. Rev.* **2025**, *58*, doi:10.1007/s10462-025-11253-3.
20. Jegham, N.; Koh, C.Y.; Abdelatti, M.; Hendawi, A. YOLO Evolution: A Comprehensive Benchmark and Architectural Review of YOLOv12, YOLO11, and Their Previous Versions. **2025**.
21. Hidayatullah, P.; Syakrani, N.; Sholahuddin, M.R.; Gelar, T.; Tubagus, R. *YOLOv8 to YOLO11: A Comprehensive Architecture In-Depth Comparative Review*; 2025;
22. Li, X.; Wang, W.; Wu, L.; Chen, S.; Hu, X.; Li, J.; Tang, J.; Yang, J. Generalized Focal Loss: Learning Qualified and Distributed Bounding Boxes for Dense Object Detection. **2020**.
23. Chakrabarty, S. YOLO26: An Analysis of NMS-Free End to End Framework for Real-Time Object Detection. **2026**.
24. Hidayatullah, P.; Tubagus, R. YOLO26: A Comprehensive Architecture Overview and Key Improvements A PREPRINT; 2026;
25. Lin, T.-Y.; Dollár, P.; Girshick, R.; He, K.; Hariharan, B.; Belongie, S. Feature Pyramid Networks for Object Detection. **2017**.
26. Liu, S.; Qi, L.; Qin, H.; Shi, J.; Jia, J. Path Aggregation Network for Instance Segmentation. **2018**.
27. Tan, M.; Pang, R.; Le, Q. V. EfficientDet: Scalable and Efficient Object Detection. **2020**.
28. Chollet, F. Xception: Deep Learning with Depthwise Separable Convolutions. **2017**.
29. Ioffe, S.; Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. **2015**.
30. Hendrycks, D.; Gimpel, K. Gaussian Error Linear Units (GELUs). **2023**.
31. Ramachandran, P.; Zoph, B.; Le, Q. V. Searching for Activation Functions. **2017**.
32. Wang, C.-Y.; Liao, H.-Y.M.; Yeh, I.-H.; Wu, Y.-H.; Chen, P.-Y.; Hsieh, J.-W. CSPNet: A New Backbone That Can Enhance Learning Capability of CNN. **2019**.
33. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep Residual Learning for Image Recognition. **2015**.
34. Vaswani, A.; Brain, G.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. *Attention Is All You Need*; 2023;

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.