ConvNeXt Backbones in YOLO for Object Detection

Logan Lechuga, Jared Shi, Rayan Syed, Peter Zhao
{llechuga,jaredshi,rsyed,pyzhao}@bu.edu

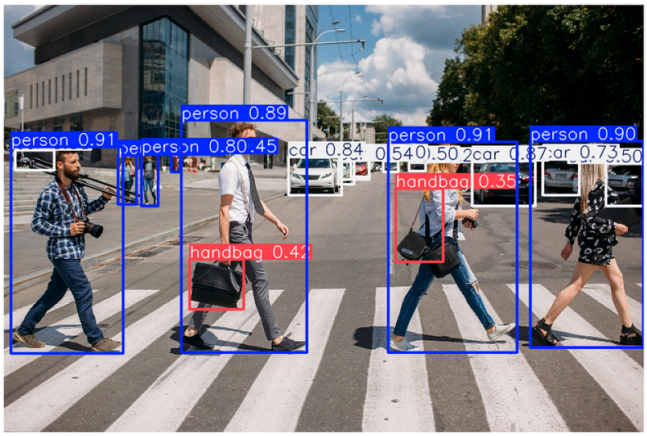

Figure 1. YOLO detection of objects in an image

1. Task

Our task is to improve real-time object detection by integrating a modern ConvNeXt [1] backbone within a YOLO (You Only Look Once) [2] framework. Object detection involves simultaneously localizing objects within an image by using bounding boxes and classifying them. We aim to detect a variety of common objects, such as people, cars, and animals, as illustrated in Fig. 1.

The central challenge lies in understanding whether architectural advances in ConvNeXt improve YOLO's detection capability, particularly for small or densely packed objects. Backbones that emphasize richer features may help recognize small or occluded targets but can also alter feature-map compatibility and computational load within YOLO's detection head. Our goal is to analyze this interaction systematically and quantify how a ConvNeXt backbone changes detection performance.

2. Related Work

2.1 You Only Look Once (YOLO)

The You Only Look Once (YOLO) family of detectors [2] introduced a unified, single-stage framework that performs classification and localization jointly, enabling real-time object detection. Subsequent versions, such as YOLOv2-YOLOv8 [3], have progressively improved accuracy and speed through architectural refinements such as residual blocks, anchor boxes, feature pyramid connections, and better training schedules. However, YOLO models still struggle to maintain accuracy on small or densely overlapping objects, largely because their backbones have limited ability to capture fine-grained spatial details across scales.

2.2 YOLO Architecture

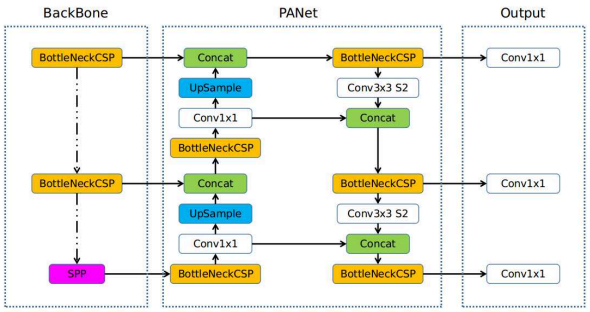

Figure 2. Original YOLO Architecture

The YOLOv5 model consists of three core components: the backbone, neck, and head, as seen in Fig. 2.

YOLO's backbone consists of a convolutional neural network (CNN) responsible for extracting feature maps with critical information like edges, textures, and semantic information from the input image, breaking it down layer by layer. The default YOLOv5 backbone is a variant of CSPDarknet (Cross Stage Partial Darknet). In YOLO, the feature maps from the last three stages of this CSP network are sent over to different layers of the YOLO neck for further feature aggregation. These are typically referred to as P3, P4, and P5 in order from largest to smallest feature maps.

YOLO's neck ingests these multi-scale feature maps produced by the backbone and mixes and refines them accordingly using a PANet (Path Aggregation Network). This feature aggregation is described more thoroughly in section 2.3.

YOLO's head applies convolutional layers to the refined feature maps P3, P4, and P5 from the neck to perform the job of detecting and classifying objects. The head associated with feature map P3 focuses on detecting small objects, P2's focuses on medium objects, and P1's focuses on large objects. The bounding boxes put on the potential objects are of the size of anchor boxes predefined beforehand. YOLO predefines nine anchor boxes to sufficiently capture most object shapes and sizes and places class probability on each box decided to be a valid object.

2.3 Feature Aggregation: FPN and PANet

Modern YOLO models rely heavily on feature aggregation to handle objects of varying sizes. This is primarily split into two steps: a Feature Pyramid Network (FPN) and a PANet, as mentioned before.

The FPN implements a top-down path that propagates high-level semantic information from deep layers to shallow layers, enriching the smaller feature maps from the backbone with contextual knowledge.

The PANet enhances FPN by adding a bottom-up path, creating a shortcut for strong, precise localization signals from the shallow (high-resolution) layers to reach the deep (low-resolution) layers faster. This two-way flow ensures all final feature maps contain both robust semantic knowledge and accurate positional details, which is crucial for detection across scales.

2.4 ConvNeXt

ConvNeXt [1] modernized classical CNNs by adopting design elements inspired by Vision Transformers, such as large depthwise convolutions, inverted bottlenecks, Layer Normalization instead of BatchNorm, and Gaussian Error Linear Unit (GELU) activation, allowing ConvNeXt to match transformer performance on large-scale image-classification benchmarks while also retaining the computational advantages of CNNs.

Recent studies have explored integrating ConvNeXt with YOLO architectures in specialized domains. For example, CMNI-YOLO [4] applied this hybrid design to digital pathology, showing improved detection of mitotic figures. Similarly, ConvNeXt-YOLO [5] proposes a framework that replaces YOLO's backbone with ConvNeXt to enhance feature extraction and multi-scale representation. Their work demonstrated improved detection accuracy, but was evaluated on limited, domain-specific datasets and lacked analysis of large-scale generalization and standardized benchmarks.

Building on this, our project investigates whether integrating ConvNeXt as a backbone within standard YOLOv5 frameworks can achieve robust, reproducible performance on large-scale, standardized benchmarks.

3. Approach

3.1 Models

We used the official YOLOv5 implementation from Ultralytics to avoid re-implementing the detection pipeline from scratch. We had several reasons for choosing YOLOv5 over newer versions of the YOLO framework. The YOLOv5 code is highly readable and accessible with an active public repository, which is critical for our use-case. On the other hand, YOLOv8 and newer versions have a lot of code for the backbone and other core modules hidden behind APIs, making it less reproducible and harder to alter.

Specifically, we chose to work with YOLOv5s as our primary baseline. It has around 7.5M parameters. We also decided to additionally work with YOLOv5m, which has 21.2M parameters, to provide a more robust experiment and comparison.

For ConvNeXt integration, we chose to integrate the ConvNeXt-Tiny backbone from the HuggingFace PyTorch Image Models (timm) library in place of YOLO's default CSPDarknet backbone. This small ConvNeXt variant has around 29.6M parameters, making it the most fair model to benchmark the smaller YOLOv5 variants against.

3.2 Modified Architecture

As explained in section 2.2, the YOLO neck receives three feature maps P3, P4 and P5 from the last three stages of the CSP network.

However, replacing the CSPDarknet backbone with a ConvNeXt network makes this entire feature map extraction process change. CSPDarknet builds feature maps layer by layer, which allows the neck to index into the backbone. However, the ConvNeXt network outputs these feature maps all together, which makes indexing no longer feasible. Additionally the feature sizes needed to be realigned to match those of YOLO's neck, so many modifications were needed.

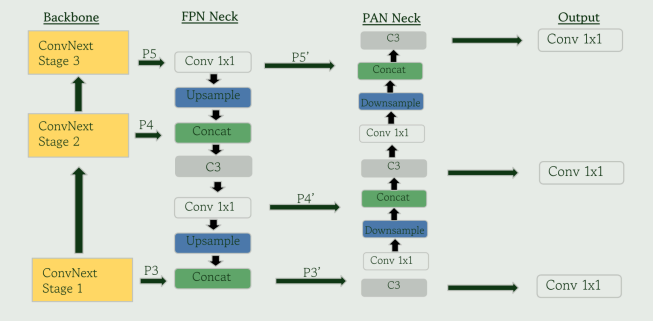

Figure 3. Modified ConvNeXt YOLO architecture with the CSPDarknet backbone replaced with ConvNeXt-Tiny

As seen in the architecture diagram of Fig. 3, this original CSPDarknet backbone was replaced with the ConvNeXt stages. A deep dive on the implementation details for the aforementioned architectural changes is provided below.

Starting with the backbone, the original CSPDarknet backbone (10 layers) was replaced with a single, pretrained, modified ConvNeXt-Tiny module. Our new ConvNeXt class handles the entire backbone functionality by loading in ImageNet pretrained weights using the timm library and extracting multi-scale features from the last three stages, corresponding to the required P3, P4, and P5 scales.

The primary engineering challenge here was ensuring that the new ConvNeXt features were correctly passed to the existing YOLOv5 PANet Neck. The raw feature maps output by ConvNeXt-Tiny had channel dimensions of (192, 384, 768). These dimensions needed to be mapped to the YOLOv5 Neck's expected channel dimensions of (128, 256, 512). This was achieved using 1x1 convolutions within the ConvNextTiny module. We chose smaller output channels because ConvNeXt-Tiny is a lighter model, preventing a channel bottleneck. Finally, since the YOLOv5 YAML parser expects a singular output containing all three of these feature maps from the backbone, we implemented a Global Feature Registry (_CONVNEXT_FEATURES) pattern to pass the multi-scale features (P3, P4, P5) to the Neck. new helper modules (ConvNextP3 and ConvNextP4) retrieve the appropriate features from this registry as needed instead of relying on the sequential layer output.

As far as the original YOLOv5 PANet Neck (FPN + PAN structure) and the detection Head are concerned, everything including the predefined anchor boxes were preserved to isolate the performance change after training solely to the backbone substitution.

3.3 Loss Function

To train, we minimized the same composite loss function as the original YOLO model:

$$\mathcal{L}_{\text{total}} = \lambda_{\text{box}}\mathcal{L}_{\text{box}} + \lambda_{\text{cls}}\mathcal{L}_{\text{cls}} + \lambda_{\text{obj}}\mathcal{L}_{\text{obj}}$$

L_{box}, the bounding box localization loss, uses CIoU Loss to measure the deviation between the predicted box and the ground-truth box, encouraging correct size and position.

L_{cls}, the classification Loss, uses Binary Cross-Entropy (BCE) Loss to measure the error in predicting the correct class label for an object that has been determined to be present.

L_{obj}, the objectness score loss, also uses Binary Cross-Entropy (BCE) Loss to determine if a specific box should contain any object at all.

λ_{box}, λ_{cls}, and λ_{obj}, are the hyperparameters that determine the relative importance of each loss component in the total loss calculation. These values are often adjusted to ensure the model prioritizes accurate bounding box prediction over classification, or vice versa. We decided to keep the default YOLOv5 hyperparameters for all training runs.

3.4 Training

Table 1. Parameter sizes across YOLO models

Model	Parameter Size
YOLOv5s	7.5M
YOLOv5m	21.2M
ConvNeXt-YOLO	**29.8M**
YOLOv5l	56.5M

We defined our baseline as YOLOv5s. After comparing the parameter sizes of our new ConvNeXt-YOLO model with the rest of the YOLO models as seen in Table 1, we decided that we would run experiments with YOLOv5m as well. YOLOv5l, on the other hand,

was far too large to make a fair comparison so we decided to omit it from our experiments.

Both the YOLO models we decided to benchmark were loaded in with their official pretrained weights. Even though they were already pretrained on the MS-COCO dataset, we wanted to create identical training conditions to provide a fair comparison to our ConvNeXt-YOLO so we decided to fine-tune it for an additional 50 epochs on MS-COCO again. 50 epochs was decided after carefully considering a balance between seeing substantial results while not taking too much compute or time. The resulting metrics were similar to those which YOLO provided, as expected.

For our new ConvNeXt-YOLO model, we loaded in the pretrained weights for the ConvNeXt backbone (from ImageNet-1k), but initialized the neck and head with random weights. It was trained under the same settings as our baseline, with 50 epochs on the MS-COCO dataset.

Table 2. Training time across YOLO models

Model	Training Time (hrs)
ConvNeXt-YOLO	17.5
YOLOv5m	16.1
YOLOv5s	7

All experiments were conducted in PyTorch, leveraging open-source libraries for model definition and training. We successfully set up our YOLOv5 codebase in a reproducible format and set up scripts for training and validating the model on the SCC. To match YOLO's setup, we logged all training output via Comet, an end-to-end model evaluation platform for AI developers. Training was conducted on BU's SCC using 4 cores and an NVIDIA RTX 6000 GPU. As seen in Table 2, we found that the training times for models is consistent with model sizes, with our ConvNeXt model taking the longest at 17.5 hours. More in depth details on subsequent results are detailed in section 6.

4. Datasets

As mentioned earlier, we use the MS-COCO dataset [6], which contains approximately 118,000 training images and 5,000 validation images annotated with bounding boxes and class labels across 80 object categories. The dataset includes objects of varying sizes and levels of occlusion and readily available for training, making it suitable for evaluating both overall and small-object detection performance. This dataset is the standard for benchmarking YOLO.

Through early preprocessing and transformations within the YOLO pipeline, many augmentations are applied. Some of these include mosaic, affine transforms, color augmentations, spatial occlusions, and label augmentations to improve generalization.

Note that the test labels in this dataset are closed source, which is common in object detection datasets to maintain benchmark fairness and accuracy. Due to this, our results will only include values from train and validation splits.

5. Evaluation Metrics

Performance will be measured using mean Average Precision (mAP), the standard metric for object detection. Average Precision (AP) is first computed for each class by integrating the precision-recall curve across multiple confidence thresholds. The mean of these AP values across all classes yields mAP, providing a single measure that captures both localization accuracy (correct bounding box placement) and classification accuracy (correct object labeling).

We will report mAP@[.5:.95], the COCO evaluation protocol used in YOLO papers, which averages AP over ten intersection-over-union thresholds ranging from 0.5 to 0.95 in steps of 0.05.

We will consider the project successful if the ConvNeXt-backed YOLO model achieves equal or higher mAP than the baseline YOLO model, YOLOv5s, under identical training conditions.

6. Results

This section presents a thorough comparison of our ConvNeXt-YOLO model compared to the YOLOv5 baselines, focusing on detection accuracy (mAP), training dynamics, and model throughput.

6.1 Training Curves

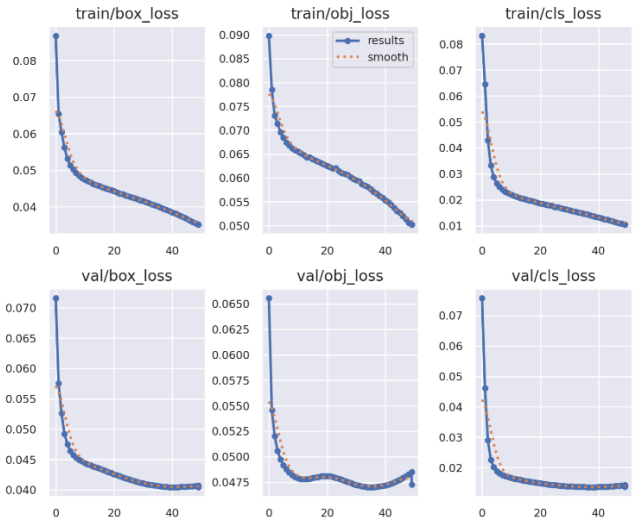

Figure 4. Training curves for ConvNeXt-YOLO

Fig. 4 illustrates the training dynamics for the ConvNeXt-YOLO model over 50 epochs. The graphs show the convergence of the loss function per each component (L_{box}, L_{cls}, L_{obj}) for both training and validation sets. A consistent reduction in all three loss components demonstrates successful training.

It is worth mentioning that training loss at times is noticeably higher than the validation loss, but this in fact is expected for object detection models, especially with YOLO. As explained in section 4, the early augmentations applied help improve generalization. The mosaic augmentation in particular has a unique approach of combining four images into a single large training image, but this results in a decreased training accuracy. There are no augmentations applied for the validation split, leading to more stability there.

Additionally, the objectiveness loss appears to follow an abnormal trajectory, but our analysis leads us to believe that the ConvNeXt architecture resulted in high object detection performance and subsequent convergence quickly. Once this was converged, the other two loss components led to this one varying throughout the later half of training.

Table 3. Final Loss Values

Model	Box	Objective	Classification
Train	0.035	0.050	0.010
Val	0.040	0.047	0.013

As seen in Table 3, the final loss values for both training and validation line up with the convergence displayed in the curve. We consider the training procedure a success.

6.2 Detection Accuracy

Figure 5. mAP@[0.5:0.95] over epochs

With respect to our desired mAP@[.5:.95] metric, ConvNeXt-YOLO's performance matched what we see in the training curves, as seen in Fig. 5. The mAP@[.5:.95] continued increasing as the backbone aligned better with the YOLO architecture and approached convergence.

The YOLO models' fine-tuning leads to drops in mAP initially due to the expected fluctuation and recalibration in early training, but quickly returns back to its optimal mAP values, hence the unusual curve.

Table 4. mAP[0.5:0.95] for all models over 50 Epochs

Model	mAP@[.5:.95]
YOLOv5s (Baseline)	0.3709
YOLOv5m	0.4478
ConvNeXt-YOLO (Us)	**0.4229**

As shown in Table 4, the ConvNeXt-YOLO model achieved an mAP@[.5:.95] of 0.4229, which is higher than the YOLOv5s baseline (0.3709), successfully meeting the project's success criterion.

However, when comparing ConvNeXt-YOLO to YOLOv5m, a model with a more similar parameter count, YOLOv5m achieved a higher mAP@[.5:.95] of 0.4478. We strongly believe sufficient hyperparameter fine-tuning along with more ideal training conditions (e.g., smaller learning rate, more epochs) would lead to

stronger convergence and a potential surpassing of YOLOv5m.

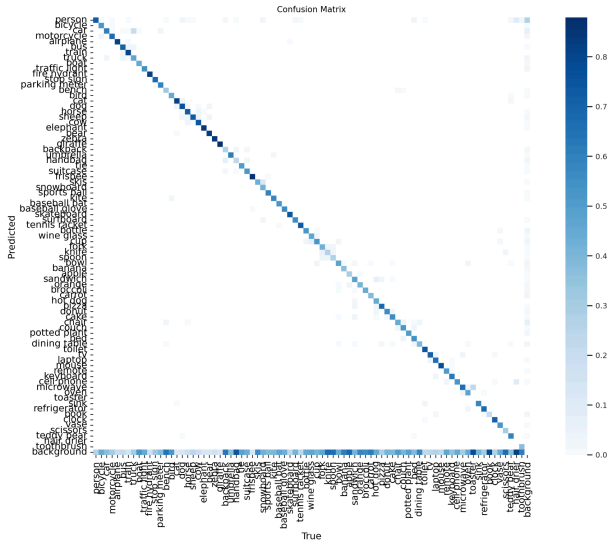

Figure 6. Confusion Matrix for ConvNeXt-YOLO

Along with high mAP@[.5:.95], we plot a confusion matrix, as seen in Fig. 6, to further demonstrate our ConvNeXt-YOLO's strong performance for object detection. We see a clear diagonal line, representing frequent, correct classifications by the model. There are a few misclassifications, but these are few and low confidence.

Figure 7. Sample Model Results. Above are the predictions from ConvNeXt-YOLO and below are the ground truth labels

Fig. 7 shows some visual examples of the predicted bounding boxes from our ConvNeXt YOLO against the ground truth. While the accuracy is noticeably high, something to notice is in some images, such as the one on the bottom right, a noticeably small object is picked up by our model and attempted to give a class label despite it not being in the ground truth. While it is classified incorrectly, it makes a reasonable prediction of a chalk board looking strikingly similar to a laptop, showing a stronger ability to notice smaller objects.

6.3 Model Throughput

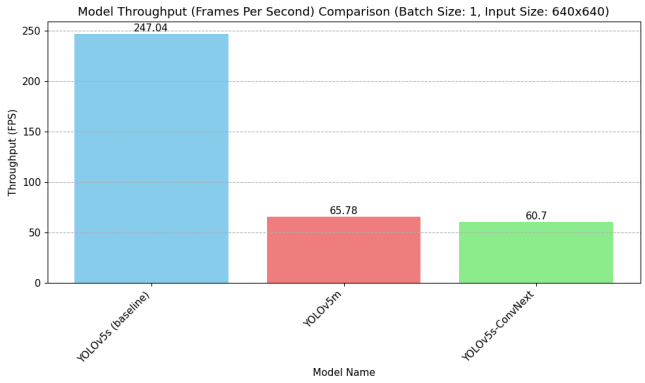

Figure 8. Throughput across YOLO models

Besides performance, another core metric for real-time object detection is model throughput. We use this to quantify the efficiency of our proposed ConvNeXt-YOLO architecture relative to the YOLOv5 baselines. The experiment was standardized across all models by using a batch size of 1 and an input size of 640 x 640. The model was first subjected to 100 warm-up runs, followed by 300 test runs. The final throughput was calculated by taking the average of the 300 test runs. The results in Fig. 8 show that the lightest model, YOLOV5s, achieved the maximum throughput at 247.04 FPS. In addition, the intermediate YOLOv5m model achieved 65.78 FPS while the ConvNeXt-YOLO model achieved the lowest throughput at 60.7 FPS, but it was still quite competitive with YOLOv5m. However, this result highlights the computational overhead introduced by the ConvNeXt architecture, which may limit its suitability for applications requiring real-time processing.

7. Conclusion

We successfully demonstrated that integrating the modernized ConvNeXt-Tiny backbone into the YOLOv5s framework significantly enhances detection

accuracy. The ConvNeXt-YOLO model achieved an mAP@[.5:.95] of 0.4229, exceeding the YOLOv5s baseline's 0.3709 on the MS-COCO dataset, thus meeting the project's success criterion. This accuracy boost validates that the rich, multi-scale features extracted by the ConvNeXt backbone, despite its architectural differences, are compatible with and enhance YOLO's detection pipeline.

However, the gain in accuracy came at a significant cost in efficiency. The ConvNeXt-YOLO model was unable to surpass the performance of the smaller YOLOv5m model, which achieves a higher mAP@[.5:.95] of 0.4478 with 9M fewer parameters. This suggests that while the ConvNeXt architecture provides superior features, its integration may not be optimally configured for the YOLO pipeline.

Further work is needed in robust hyperparameter tuning (e.g., learning rate and batch size) and longer training periods (e.g., 300 epochs like the original YOLOv5) to allow the randomly initialized components to fully converge, more strongly align with the backbone and potentially outperform YOLOv5m. The trade-off remains a core focus: balancing the powerful feature representation of modern backbones against the computational efficiency required for real-time applications.

Through this project, we gained invaluable hands-on experience by applying theoretical concepts to a real-world problem. We achieved an in-depth exploration and modification of the YOLO architecture, directly implementing changes to existing CNN modules. Finally, we gained practical experience with the entire deep learning model lifecycle, from training and evaluation to working with computing clusters and sending batch jobs.

References

[1] Z. Liu, H. Mao, C-Y. Wu, C. Feichtenhofer, T. Darrell, S. Xie. A ConvNet for the 2020s, arXiv.org, 2022.

[2] J. Redmon, S. Divvala, R. Girshick, A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection, arXiv.org, 2016.

[3] J. Terven, D-M. Córdova-Esparza, J-A. Romero-González. A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS, Machine Learning and Knowledge Extraction, 5(4):1680-1716, 2023.

[4] Y. Topuz, S. Yıldız, S. Varlı. ConvNeXt Mitosis Identification—You Only Look Once (CNMI-YOLO): Domain Adaptive and Robust Mitosis Identification in Digital Pathology. Laboratory Investigation, 2024. doi:10.1016/j.labinv.2024.102130.

[5] B. Jiang, Z. Lin, Z. Li, C. Jia, H. Lu. A Computer Vision Framework for Enhanced Object Detection Using ConvNeXt-YOLO Architecture with Multi-Scale Signal Processing. In: Proc. 2024 IEEE 7th International Conference on Automation, Electronics and Electrical Engineering (AUTEEE), Shenyang, China, 2024. doi:10.1109/AUTEEE62881.2024.10869786.

[6] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, C. L. Zitnick. Microsoft COCO: Common Objects in Context. In: ECCV, 2014.