A GO-WGAN and Graph Transformer-Based Framework for Real-Time Mobile UI Layout Generation and Estimation

Li Juan Wang
Email: LiJuanWangg@outlook.com
School of Fine Arts, Wuhan Business University, Wuhan 430050, China

Keywords: APP interface generation, generative adversarial networks, transformer, layout estimation, UI automation, graph neural networks

Received: September 3, 2025

Designing user interface (UI) layouts for mobile applications is a labor-intensive task that demands considerable manual effort and domain expertise. To address this, we present FastLayout, a deep learning framework that fuses generative and relational modeling for dynamic UI layout generation and estimation. At its core, FastLayout integrates WGAN-GP (adapted here as GO-WGAN) to generate diverse, high-quality synthetic layout samples, alleviating data scarcity. The Branching Hybrid Attention Mechanism (BHAM) enhances the CNN backbone by improving feature extraction and reducing task conflicts, while the Graph Transformer explicitly models spatial and relational dependencies between UI components. Our experiments use an internal dataset of 3,275 annotated mobile UI screens across 15 layout categories and the public RICO benchmark, with training conducted in two phases using AdamW optimization and early stopping to ensure fairness and stability. Experimental evaluations show that FastLayout achieves an Intersection over Union (IoU) of 83.52%, a Pixel-wise Error (PE) of 4.97%, an Edge Error (EE) of 4.91 units, and a Root Mean Square Error (RMSE) of 0.2753, outperforming state-of-the-art baselines in both accuracy and efficiency. solution for intelligent UI automation in modern software development.

Povzetek: Predstavljen je model za samodejno generiranje mobilnih UI postavitev, ki z uporabo WGAN in grafnega Transformerja presega obstoječe pristope (IoU 83,52 %).

1 Introduction

With the rapid proliferation of mobile and web applications, user interface (UI) design has become a critical component in delivering engaging, accessible, and intuitive user experiences [1]. The increasing demand for high-quality, adaptive APP interfaces across various platforms has driven the development of intelligent tools able to automate layout generation. Traditionally, the creation of APP layouts is a labor-intensive process requiring manual design, coding, and testing. This not only slows down development cycles but also makes it challenging to adapt to diverse screen sizes, resolutions, and user requirements [2]. As digital interfaces become more complex and responsive, there is a growing need for automated systems that can generate interface layouts dynamically while maintaining visual coherence and functional usability [3].

Recent advancements in artificial intelligence and machine learning have introduced promising techniques for layout automation, particularly through deep generative models [4]. Among these, Generative Adversarial Networks (GANs) have shown strong ability to learn and generate realistic data [5]. GANs have been applied in numerous visual tasks, including image generation, style transfer, and data augmentation. However, applying GANs directly to UI layout generation introduces several challenges [6]. Layouts are not merely visual structures—they are hierarchical, semantically rich arrangements of components that must adhere to specific design constraints such as alignment, spacing, and responsiveness [7]. As a result, generating coherent and functional UI layouts requires a deeper understanding of spatial relationships and structural rules than traditional GAN architectures can offer [8].

To address these limitations, recent research has turned to Transformer-based models [9], which have revolutionized natural language processing and have shown strong potential in vision-related tasks. Transformers are particularly effective at modeling long-range dependencies and global context, making them suitable for capturing the complex interactions between UI components [10, 11]. When fused with CNN-based visual feature extraction, Transformers can reason about layout composition in a more structured and holistic way [12-14]. This fusion forms the foundation of our proposed system: FastLayout, a dynamic layout generation framework that leverages the strengths of both GANs and Transformers. Research on automated UI layout generation has progressed from heuristic rule-based systems to deep learning models leveraging CNNs, GANs, and Transformers. Early approaches such as Planar RCNN focus on bounding-box detection, achieving moderate IoU but limited ability to capture

semantic hierarchies. RandC introduces random cuboid approximations but suffers from slow inference due to multi-stage optimization. Ncuboid variants improve structural reasoning but remain computationally expensive and cannot fully model relational constraints across UI components.

Table 1 summarizes key performance metrics reported by state-of-the-art methods, with an additional column indicating their main limitations. Despite notable progress, existing methods face two critical shortcomings: (i) inadequate modeling of hierarchical and relational dependencies between UI components, and (ii) trade-offs between accuracy and real-time efficiency. Our proposed FastLayout addresses these gaps by (1) using WGAN-GP (GO-WGAN) for synthetic data generation to mitigate data scarcity, (2) integrating BHAM-enhanced CNNs for improved feature extraction, and (3) employing a Graph Transformer to explicitly capture hierarchical and relational dependencies, all while maintaining sub-100ms inference times.

Table 1: Comparative evaluation of prior works

Method	IoU (%)	PE (%)	EE	RMSE	Inference Time (s)	Main Limitation
Planar RCNN	79.64	7.04	6.58	0.4013	0.110	Cannot capture hierarchy; limited relational reasoning
RandC	76.29	8.07	7.19	0.3465	5.350	Very slow inference; impractical for real-time use
Ncuboid (w/o)	79.94	6.40	6.80	0.2827	0.108	No hierarchy modeling; weaker structural accuracy
Ncuboid (w/)	81.40	5.87	5.78	0.2905	0.196	Better accuracy but higher runtime; lacks relational modeling
FastLayout (ours)	83.52	4.97	4.91	0.2753	0.087	Addresses hierarchy and relations; real-time efficiency

In this paper, we propose a novel approach to dynamic APP interface layout generation by combining a Generative Adversarial Network (GO-WGAN) for data augmentation and layout simulation with a graph-based Transformer model for layout reasoning and prediction. The primary objective is to address three core challenges: limited availability of annotated layout data, lack of structural modeling in existing methods, and inefficiency in real-time layout rendering. The GO-WGAN is designed to mitigate data scarcity by generating realistic and diverse layout samples, while the Transformer-based model performs structured layout estimation and optimization.

Our framework, FastLayout, incorporates a multi-task CNN enhanced with a Branching Hybrid Attention Mechanism (BHAM) to extract both regional and relational features from APP screenshots. These features are encoded into graph representations and processed by a multi-layer Transformer encoder, which predicts the final layout parameters. We hypothesize that the integration of generative augmentation with graph-aware Transformer modeling will significantly improve both accuracy and runtime efficiency over state-of-the-art methods.

To further enhance computational efficiency, we introduce an inverse 2D layout generation method that avoids expensive 3D operations and leverages the parallel processing capabilities of modern GPUs. This method reconstructs screen-space layouts from normalized descriptors, enabling resolution-independent rendering and high scalability. Experimental results on an internal dataset of APP interfaces show that FastLayout achieves state-of-the-art performance in both layout accuracy and inference speed, significantly outperforming existing baselines. We define success based on achieving superior results in Intersection over Union (IoU), Pixel-wise Error (PE), Edge Error (EE), and RMSE, as well as maintaining sub-100ms rendering times per layout. To make our contributions explicit, the research goals of this work are as follows: (i) improve layout estimation accuracy over strong baselines such as Ncuboid(w/), (ii) reduce layout rendering time below 100ms to enable real-time use, and (iii) ensure generalization across heterogeneous app types (e.g., e-commerce, education, finance).

2 APP interface generation method based on improved WGAN

In this section, we propose a method for dynamic APP interface layout generation using an enhanced Wasserstein Generative Adversarial Network (WGAN) [15], which we name GO-WGAN (Gradient Optimized WGAN). This method is designed to tackle the data imbalance and scarcity challenges in real-world APP interface datasets, especially for rare or non-standard layouts. By introducing a gradient penalty mechanism into the classical WGAN framework, the GO-WGAN generates realistic, diverse layout samples that closely resemble real-world interfaces. These synthetic layouts augment the training data used for downstream

recognition and classification tasks, particularly in 1D-CNN and Transformer-based models.

2.1 Overall framework

Although WGAN has improved the loss function to improve the quality of the generated samples and enhance the robustness of the network training process, problems such as optimization stagnation and generation of poor-quality data can still occur in certain environments. The root of the problem lies in the requirement of Lipschitz constraints by setting weight thresholds, which is one of the key factors for the smooth operation of WGAN. In particular, when weight correction is used, it transforms the weighted neural network into a simplified function, which may weaken its ability to handle complex data samples. Because the weight correction narrows all the weight ranges into a small region, it is easy to cause the gradient to disappear or to be excessive. In this work, we employ the Wasserstein GAN with Gradient Penalty (WGAN-GP) [15], introduced by Gulrajani et al., as the foundation for our layout generation module. Rather than presenting a new adversarial formulation, our novelty lies in adapting WGAN-GP to the application of APP interface generation. Specifically, we dynamically tune the gradient penalty coefficient λ to optimize between stability and diversity, and we incorporate structured layout metadata (e.g., XML/JSON hierarchy) into the discriminator alongside pixel-level screenshots. These modifications enable the model to learn both visual appearance and structural constraints of user interfaces, ensuring realistic and functionally coherent layout synthesis. The penalty term is defined as:

$$\mathcal{L}\text{penalty} = \lambda \cdot E\hat{x} \sim P\hat{x}[(|\nabla\hat{x}D(\hat{x})|_2 - 1)^2] \quad (1)$$

Where $\hat{x}$ is a point sampled uniformly along straight lines between pairs of real and generated samples, and $D(\cdot)$ is the discriminator function. Based on this, the full GO-WGAN objective becomes:

$$\min_G \max_D E_{x\sim Pr}[D(x)] - Ez \sim Pz\big[D\big(G(z)\big)\big] - \mathcal{L}\text{penalty} \quad (2)$$

where *G(z)* is the generator output for random input z, P_r denotes the real data distribution, and P_z the input noise distribution. This formulation enforces the Lipschitz constraint more robustly than weight clipping and ensures training stability. Figure 1 illustrates the architecture of the proposed GO-WGAN model, including the generator and discriminator pipelines, layer configurations, and data flow. This visual helps clarify how gradient penalties are applied during training to produce high-fidelity layout samples. Based on the existing WGAN optimization objective basis, the objective function of GO-WGAN is defined as follows:

$$minmaxV(D,G) = E_{x_r\sim \mathrm{P}_{r(\mathrm{g})}} D(x_r) - E_{x_f\sim \mathrm{P}_{g(\mathrm{g})}} D\big(x_f\big) + \lambda E_{x\sim \mathrm{P}_x}[(\| \nabla_{\hat{x}} D(\hat{x}) \|_2 - 1)^2] \quad (3)$$

By looking at the objective function, this paper finds that the main difference between GO-WGAN and WGAN algorithms lies in the last term, where λ is the penalty optimization factor of the discriminator. Typically, the Lipschitz constraint refers to a function where the norm of the gradient does not exceed 1. However, in the objective function of this paper, applying a penalty to the gradient norms that are far higher than 1 makes all the gradient norms gradually approach 1. This practice can speed up the convergence of the training loss and improve the optimization of the final generated sample network. From a mathematical point of view, the reason for this is because after optimization, the distribution of the WGAN discriminator and the approximate unit gradient paradigm. So, in addition to having faster convergence, GO-WGAN can maintain a stable learning process without having to adjust the hyperparameters individually and repeatedly, while generating new high-quality sample data. This approach aligns with recent efforts in GAN-driven layout synthesis, which have demonstrated strong performance in design automation tasks [18].

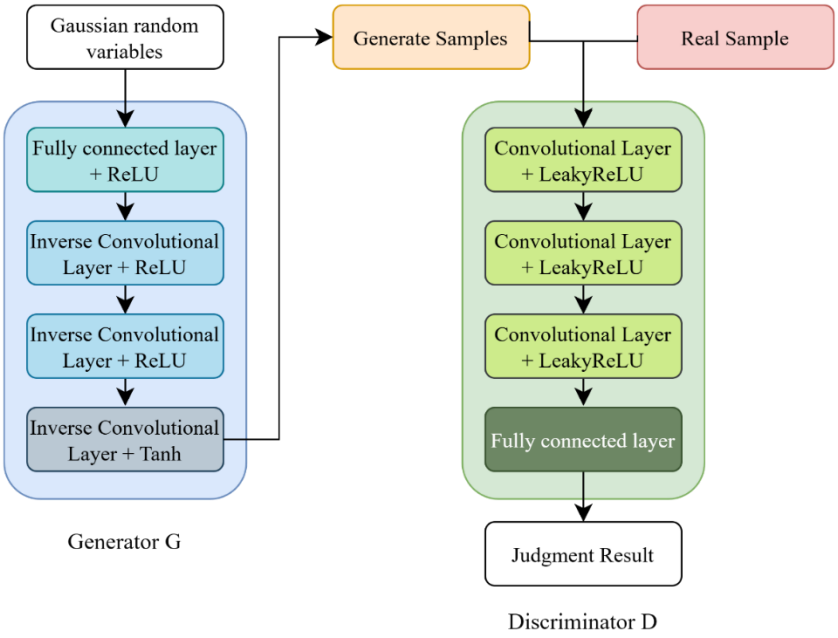

Figure 1: GO-WGAN network architecture diagram

The GO-WGAN model is employed to construct two core components of our dynamic APP interface layout generation framework: the synthetic layout sample generation module and the layout feature learning module. This system is composed of a generator and a discriminator, each playing an adversarial role. The generator is responsible for simulating new APP layout samples that capture the structural and semantic properties of real-world interface designs. Its primary goal is to deceive the discriminator, which functions as a learned evaluator that distinguishes between authentic and generated layouts based on realism, consistency, and interface usability.

In Table 2 we denote up sampling layers as ConvTranspose2d; in Table 3 we denote down sampling layers as Conv2d. All ConvTranspose/Conv layers use kernel size 4, stride 2, padding 1 unless otherwise specified, yielding 2× up/down-sampling per stage. Unless otherwise noted, ConvTranspose2d/Conv2d layers use k=4, s=2, p=1 to achieve exact ×2 up/down-sampling per stage; BatchNorm is applied to all intermediate layers of the generator and discriminator (except the first/last), with LeakyReLU(0.2) in the critic.

Table 2: Structure of the generator network model

Layer No.	Layer Type	Parameters (k, s, p)	In [C,H,W]	Out [C,H,W]	Activation / Notes
0	Latent	–	$z \in \mathbb{R}^{128}$	–	–
1	FC + Reshape	–	z	[512, 4, 4]	ReLU; output = 512×4×4 seed
2	ConvTranspose2d	k=4, s=2, p=1	[512, 4, 4]	[256, 8, 8]	BN + ReLU
3	ConvTranspose2d	k=4, s=2, p=1	[256, 8, 8]	[128, 16, 16]	BN + ReLU
4	ConvTranspose2d	k=4, s=2, p=1	[128, 16, 16]	[64, 32, 32]	BN + ReLU

Table 3: Discriminator network model structure

Layer No.	Layer Type	Parameters (k, s, p)	In [C,H,W]	Out [C,H,W]	Activation / Notes
1	Conv2d	k=4, s=2, p=1	[1, 64, 64]	[64, 32, 32]	LeakyReLU(0.2)
2	Conv2d	k=4, s=2, p=1	[64, 32, 32]	[128, 16, 16]	BN + LeakyReLU(0.2)
3	Conv2d	k=4, s=2, p=1	[128, 16, 16]	[256, 8, 8]	BN + LeakyReLU(0.2)
4	Conv2d	k=4, s=2, p=1	[256, 8, 8]	[512, 4, 4]	BN + LeakyReLU(0.2)
5	FC (flatten)	–	512×4×4	1 (scalar)	None (Wasserstein critic output)

Table 4: IDCNN network model architecture parameters

Structure number	Structure type	Parameters
1	Conv Layer	K:16×1×64； step：16； padding='none'
2	BN Layer	-

Each component is defined by distinct architectural parameters such as the number of weights, kernel size, and input/output dimensions, which determine the behavior and representational capacity of the network. In the generator, the first four layers use ReLU as the activation function, enabling non-linear feature transformation. Immediately after each of these layers, batch normalization is applied to stabilize the learning dynamics and improve training efficiency. For the final layer, the Tanh activation function is used in place of ReLU to constrain output values within a bounded range, ensuring the smoothness and visual coherence of the synthesized layout representations.

The discriminator, on the other hand, is structured to progressively extract multi-scale features from the input layout through a series of convolutional layers. LeakyReLU is employed as the activation function for all layers except the final output layer [16], which produces a single-valued judgment. LeakyReLU helps avoid the "dying neuron" problem and retains gradient flow even when activations are close to zero, thus enhancing the discriminator's ability to detect subtle differences between real and generated layout structures.

This architectural design enables the GO-WGAN to model complex and diverse UI layout distributions, making it a powerful tool for data augmentation, interface generation, and pre-training layout encoders in broader Transformer-based frameworks. The high-fidelity samples generated by the GO-WGAN not only simulate realistic APP interface components but also contribute to balancing layout category distributions, which is especially valuable for training downstream models in low-resource scenarios.

2.2 Generation process

Figure 2 illustrates the complete process of the proposed dynamic APP interface layout generation and classification method, which consists of three main stages: (1) collecting original interface layout data, (2) training the adversarial network to synthesize new layout samples, and (3) performing layout classification using a Transformer-based model. Each stage is described in detail below:

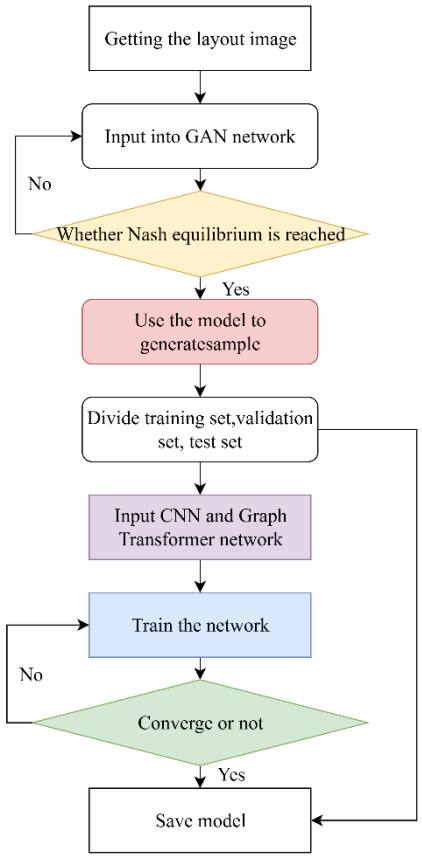

Figure 2: GO-WGAN-based layout generation and classification workflow diagram. The process includes data generation with GAN, dataset splitting, and layout estimation using a CNN + graph transformer model.

(1) Collect Original Interface Layout Data

To initiate the layout generation process, we compiled a dataset of 3,275 raw APP interface layouts collected from 150+ real-world Android applications spanning e-commerce, education, finance, healthcare, and productivity domains. These layouts were acquired both as pixel-level screenshots and as structured vector data (e.g., XML and JSON extracted via APK decompilation and Android UI automation tools). Each sample was manually verified and annotated with component types, bounding boxes, and semantic labels (e.g., buttons, input fields, containers) by a team of trained annotators following a consistent labeling protocol. Unlike traditional preprocessing pipelines, our approach avoids handcrafted features, enabling the GO-WGAN model to learn spatial arrangements and interface semantics directly from natural layouts.

(2) Train the Adversarial Network to Generate Layout Samples

The next step involves training the GO-WGAN network to learn the underlying distribution of APP layout structures. Initially, the model parameters are randomly initialized. During training, the generator and discriminator participate in a competitive process in which the generator seeks to produce convincing layout samples, while the discriminator aims to distinguish these from real layouts. This adversarial interaction continues iteratively until a Nash equilibrium is reached—i.e., when the discriminator can no longer reliably distinguish between real and generated layouts. Once the network converges, the final weights are saved, and the trained generator is deployed to generate a substantial set of realistic synthetic interface layouts. To improve reproducibility, we summarize the GO-WGAN training configuration in Table 5. These generated samples mirror the compositional and structural rules found in real-world UIs, including consistent spacing, component hierarchy, button placement, and text alignment. This capability is particularly critical in domains where only limited interface samples are available, such as early-stage app prototyping or niche design systems.

Table 5: GO-WGAN training configuration

Parameter	Value/Setting
Gradient penalty coefficient (λ)	10
Optimizer	Adam ($\beta_1 = 0.5$, $\beta_2 = 0.9$)
Learning rate (generator / disc.)	1e-4 / 1e-4
Batch size	64
Number of epochs	200
Training steps per epoch	~500 (based on dataset size)
Discriminator: Generator updates	5 : 1
Latent vector dimension (z)	128
Weight initialization	Xavier uniform
Normalization	BatchNorm in generator, LayerNorm in critic

(3) Interface Layout Estimation Using Graph Transformer

After generating the new synthetic samples, they are merged with the original interface dataset to construct a balanced and diverse training set. This enriched dataset helps eliminate class imbalance issues—such as underrepresentation of certain UI patterns (e.g., login screens, shopping carts, dashboards)—and significantly enhances the generalization capacity of downstream models. The final step in the pipeline is layout structure estimation, carried out using a hybrid model that combines a convolutional neural network (CNN) with a graph-based Transformer. The CNN is responsible for extracting hierarchical visual features from the interface—such as containers, text fields, and interactive components—while the Graph Transformer captures the spatial and semantic relationships between these elements. This integrated architecture supports end-to-end learning of UI layout structures and offers stronger structural reasoning capabilities compared to traditional classification-based methods.

3 Fast layout estimation algorithm based on graph transformer

In industrial-grade UI generation systems, two critical challenges hinder practical deployment: detection accuracy and layout generation efficiency. While the branching hybrid attention mechanism (BHAM) proposed in this section improves layout estimation accuracy by resolving task conflicts in multi-task learning, the computational cost of such models remains high. Therefore, this section focuses on enhancing layout generation speed without compromising detection performance—by combining the strengths of Convolutional Neural Networks (CNNs) and Transformers for UI structure estimation.

3.1 Determining size and appearance of UI elements

To generate accurate and functional UI layout code, it is essential to determine both the size and visual style of each interface component. This allows for the reconstruction of the target layout with high fidelity. In this section, we analyze how Android systems define UI element dimensions and positioning based on screen resolution, coordinate systems, and pixel density.

Figure 3. Layout tree hierarchy of a mobile UI. The root node represents the screen, intermediate nodes correspond to containers (e.g., LinearLayout, RelativeLayout), and leaf nodes denote interactive components such as buttons, text fields, and images. Edges capture parent–child spatial and structural relationships, which serve as constraints for layout reasoning. The complete tree hierarchy reflects how Android organizes layouts: each screen is a root, containers are intermediate nodes, and interactive elements are leaves. This hierarchical encoding ensures that spatial and alignment constraints are preserved during Transformer-based reasoning.

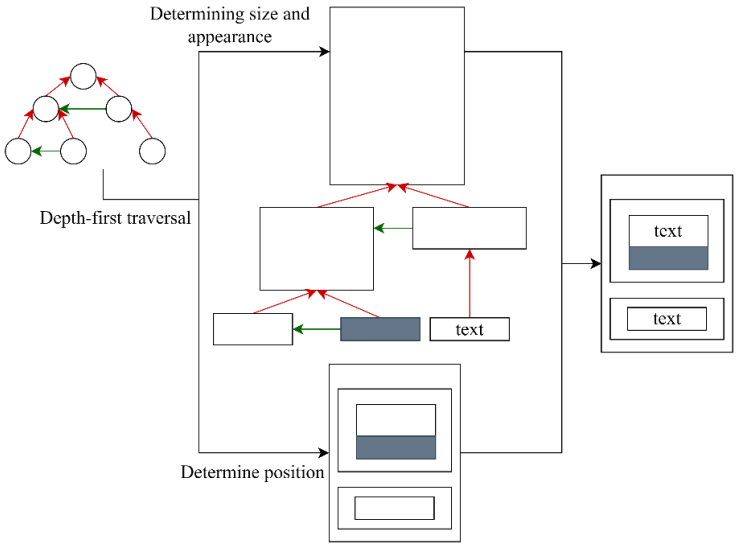

Figure 3: Layout based on a complete tree UI hierarchy

3.2 Problem formulation and model architecture

Transformers have achieved exceptional performance across deep learning domains thanks to their self-attention mechanism and parallel processing capabilities. These strengths are ideal for modeling long-range dependencies in UI layouts, where components often relate hierarchically or contextually across the screen. However, layout estimation also requires spatially aware processing, making CNNs a natural complement.

To address this, we propose a hybrid approach that fuses CNN feature extraction with Transformer-based layout reasoning. The layout estimation process is reformulated as a sequence modeling problem, where the visual and spatial attributes of UI elements are encoded as input sequences for the Transformer. Figure 4 presents the overall layout estimation pipeline, highlighting the flow from CNN-based feature extraction with BHAM to graph construction and Transformer-based relational reasoning. This diagram provides a high-level understanding of how FastLayout processes raw UI data into structured layouts.

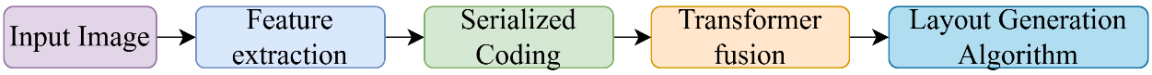

Figure 4: Overall processing flowchart

(1) Visual feature extraction

To clarify, the Branching Hybrid Attention Mechanism (BHAM) introduces a dual-branch attention structure: one branch models channel-wise dependencies to recalibrate feature importance, while the other captures spatial interactions between regions. The outputs of the two branches are adaptively fused, allowing the network to simultaneously emphasize critical features and preserve spatial coherence. This design reduces task conflicts in multi-task CNNs and enhances discriminative power for layout components. For a detailed description of BHAM's architecture, see Xu et al., "Branching Hybrid Attention Networks for Multi-Task Visual Recognition," IEEE Transactions on Multimedia, 2022.

(2) Serialization and feature encoding

Since Transformers operate on sequential inputs, visual features extracted from CNNs must be converted into ordered vector representations. This requires serializing complex layout attributes—including component boundaries, alignment cues, and spacing indicators—into structured tokens. To facilitate this, we design specialized encoders that translate UI region features and spatial relationship vectors into sequences the Transformer can process. In practice, each UI screen contains between 15–35 serialized elements on average, with an upper bound of 50 tokens per sequence. For screens with fewer elements, sequences are zero-padded to length 50 with a masking mechanism that prevents padded tokens from contributing to attention weights. For screens with more than 50 elements (rare in our dataset, <3%), the least significant background components are truncated to maintain consistency. Positional encoding is shared across all layout types and is learned jointly, ensuring a unified embedding space that captures both spatial order and structural dependencies. Algorithm 1 below outlines the serialization process that converts region-level CNN features and their relative spatial relationships into a format suitable for Transformer input. This procedure ensures each UI component is encoded as a structured token embedding both visual and geometric information, enabling the Transformer to capture long-range spatial dependencies.

Algorithm 1: Serialize CNN layout features for transformer

Input: CNN feature map F ∈ ℝ^[C,H,W], bounding boxes B = {b_1, b_2, ..., b_n}
Output: Serialized token sequence S = {s_1, s_2, ..., s_n}

for each bounding box b_i in B do
1. Extract region feature f_i = ROI_Pool(F, b_i) // fixed-size feature (256×14×14)
2. Flatten f_i into vector representation
3. Normalize spatial location:
$v_i = [x_i/W, y_i/H, w_i/W, h_i/H]$
4. Concatenate feature and spatial descriptor:
$s_i = [f_i, v_i]$
5. Apply learnable position embedding to s_i
end for
Return sequence S = {s_1, ..., s_n}

Let n be the number of detected UI elements (tokens), RRR the ROI pooled feature size (e.g., $\times 14 \times 14$), and d the token embedding dimension. The serialization runs in $O(n \cdot R + n \cdot d)$ time and uses $O(n \cdot d)$ memory for the token buffer. With our cap of $T = 50$ tokens, both time and memory are bound. Padding tokens are masked in attention and loss, so they do not add compute beyond linear buffering. Position embeddings are shared across layout types and learned jointly; they add $O(n \cdot d)$ memory and negligible extra time.

(3) Integrating optimization with network inference

To reduce computational overhead and improve end-to-end training, we eliminate traditional multi-stage post-processing pipelines. Instead, layout features such as component planes and visual boundaries are represented as graph-structured data, and their interactions are fused directly within the Transformer's attention layers using a graph-aware neighbor edge fusion mechanism. This enables the model to jointly reason about relationships—like grouping and alignment—within the self-attention framework. Graph-aware modeling has also been adopted in interface scene interpretation tasks with promising results [19].

(4) Efficient layout generation

Real-time generation of UI layouts is critical in dynamic app development workflows. Traditional methods rely on geometric calculations of intersections between layout zones, which can be computationally expensive. We propose an inverse rasterization-based generation method, which reverses the rendering process: projecting from the pixel coordinate space back to an abstract layout space. This projection leverages GPU parallelism to calculate UI boundaries and alignments across different layout planes efficiently, drastically reducing rendering time.

3.3 FastLayout network architecture

As illustrated in Figure 5, FastLayout consists of two core components: a CNN-based feature extractor with BHAM and a graph Transformer encoder. The CNN processes input UI screenshots to extract planar regions (e.g., containers, buttons) and layout-defining line segments. These features are encoded into graph-structured data, aligning them with the input format required by the Transformer. These features are encoded into graph-structured data and processed by a multi-layer Graph Transformer encoder. In this paper, we distinguish between standard Transformers and Graph Transformers, which extend the attention mechanism to operate over graph-structured data [20]. Our model adopts the latter to explicitly model relationships between UI elements such as alignment, grouping, and proximity within the layout graph.

Unlike traditional partitioning methods (e.g., Vision Transformer block slicing), we detect bounding boxes and segment lines directly using an optimized BHAM detector. This approach preserves spatial relationships critical to UI structure. The bounding boxes are then aligned using ROI Pooling to extract fixed-size features (e.g., [256, 14, 14]). This dimension was empirically selected to strike a balance between spatial resolution and computational efficiency. The choice of 14×14 preserves enough spatial detail for layout structure recognition while avoiding the memory overhead of higher resolutions. The 256 channels align with the output depth of the preceding CNN stage, ensuring smooth integration with the Transformer's input embedding layer.

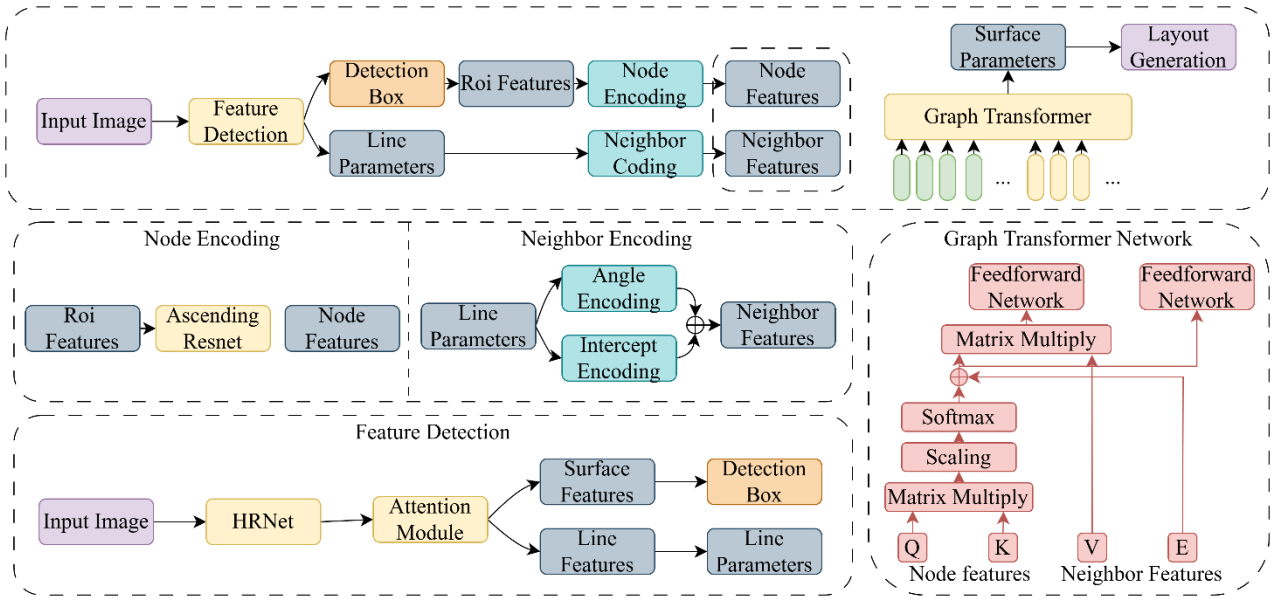

Figure 5: Overall framework diagram of the FastLayout model

To represent layout structure effectively, FastLayout constructs graph nodes from extracted UI regions and encodes neighboring relationships using detected line segments. A customized node encoder—based on a residual bottleneck design—projects each region into a compact vector embedding. To handle variable-length sequences in training, a position mask is applied to suppress normalization noise introduced by zero-padding.

Line segments, represented by angular and intercept parameters, are encoded via separate linear layers. Their features serve as edge embeddings in the graph, enabling neighbor-edge fusion within the Transformer's self-attention mechanism. This embedded fusion replaces slow post-processing techniques, allowing joint reasoning between component areas and structural guides like separators and dividers.

Each detected line segment connecting UI elements is modeled as an edge e_{ij} between nodes v_i and v_j. We encode this edge using spatial features derived from the angular and intercept parameters of the segment:

$$e_{ij} = \text{MLP}\left([\theta_{ij}, \rho_{ij}, |c_i - c_j|_2]\right) \quad (4)$$

where θ_{ij} is the angle of the line connecting elements i and j, ρ_{ij} is its y-intercept (or distance from origin in polar form), and c_i, c_j are the center points of the bounding boxes corresponding to nodes i and j, respectively. The resulting vector is passed through a multilayer perceptron (MLP) to produce the final edge embedding.

Algorithm 2: Compute edge embeddings for graph layout transformer

Input: Node centers $\{c_1, ..., c_n\}$, segment lines L = $\{(i,j,\theta,\rho)\}$
Output: Edge embedding matrix E $\in \mathbb{R}^{nxn}$

for each line segment (i, j, θ, ρ) in L do
    1. Compute Euclidean distance d = $||c_i - c_j||$
    2. Construct raw edge vector e_raw = [θ, ρ, d]
    3. Apply MLP to get e_{ij} = MLP(e_raw)
    4. Store e_{ij} in adjacency-aware embedding matrix E
end for
Return E

Let nnn be nodes and mmm edges from detected separators/lines or a k-NN rule. Edge-feature computation and MLP projection are O(m) time and $O(m \cdot d_e)$ memory (edge embedding width d_e). Edges are treated as bidirectional: each undirected relation (i,j) becomes $(i \to j)$ and $(j \to i)$ with shared parameters, which doubles mmm but simplifies directional biasing in attention. We apply two masks in the Transformer: a padding mask to exclude zero-padded tokens and a graph sparsity mask that limits attention to neighbors (plus a global [CLS] token for long-range context). Under this scheme, per-layer attention scales as O(nk)) in time and memory when each node has at most k neighbors (typ. $k \in [6,10]$); a dense variant used in ablations scales as $O(n^2)$. With T=50, d=768, k=8, and 12 layers × 12 heads, memory remains under ~4 GB (FP16) for batch size 16 on a 2080 Ti.

This formulation ensures that the model captures not just adjacency but also the geometric and directional semantics between UI components—critical for maintaining spatial structure during layout reasoning.

Precise modeling of UI spatial structure requires a strong positional representation. FastLayout evaluates multiple 2D position encoding schemes—absolute, relative, and learnable Gaussian 2D encoding. The latter, based on the Gaussian distribution centered on each region's bounding box, dynamically adjusts spatial sensitivity based on the element's size. This method enhances spatial awareness while reducing noise from underrepresented regions in sparse layouts.

Each node's final positional encoding is combined with visual and relational embeddings, producing a 768-dimensional representation compatible with multi-head attention in the Transformer. This enables parallel attention to both UI structure and hierarchical layout logic, crucial for capturing alignment, grouping, and flow in interface design.

FastLayout successfully integrates CNN-based spatial perception with Transformer-driven relational modeling. By eliminating most post-processing, reducing sequence complexity, and enabling structural fusion within the model's forward pass, FastLayout significantly improves both speed and accuracy in app layout estimation. In deployment terms, our pipeline requires two lightweight preprocessing steps: (i) ROI extraction from annotated bounding boxes or from an automated detector, and (ii) conversion of XML/JSON metadata to structured labels. Once these inputs are available, the CNN–Transformer module operates fully automatically. We therefore describe the core inference stage as "end-to-end," while acknowledging that annotation and ROI extraction are external preprocessing steps. This design makes the system suitable for real-time UI generation, adaptive interface analysis, and intelligent layout recommendation systems. However, several limitations remain. First, reliance on synthetic layouts generated by GO-WGAN may introduce bias toward common design patterns, potentially reducing diversity in novel UI styles. Second, although FastLayout achieves real-time inference on mid-range GPUs, responsiveness may degrade on low-end mobile devices, requiring further optimization. Finally, our approach models static layouts and does not capture dynamic interaction logic such as gestures or navigation flows, which limits its use for end-to-end usability evaluation. These considerations highlight opportunities for future research and development.

The training process is divided into three structured phases. Phase 1: Pretraining GO-WGAN to generate synthetic layouts and balance data scarcity. Phase 2: Training the CNN backbone with BHAM to extract visual features, followed by integration with ROI pooling. Phase 3: Joint training of the CNN-BHAM features with the Graph Transformer for relational reasoning and final layout estimation. This modular pipeline ensures stable optimization and effective use of synthetic data.

3.4 Inverse 2D layout generation method

In the context of app interface design, traditional layout rendering methods rely heavily on absolute positioning or XML-style markup (e.g., in Android or HTML), which can be inflexible and device-specific. While such methods perform well for static layouts, they struggle to generalize across devices, screen densities, and responsive contexts.

To solve this, we propose a 2D inverse layout generation method, which reconstructs interface layouts from learned or estimated layout parameters in a way that is compatible with dynamic rendering engines. Unlike 3D projection-based imaging, this method operates entirely in 2D and is optimized for screen-space rendering of UI components. The core idea is to treat layout generation as a mapping from normalized layout parameters (e.g., relative position, size, and alignment) back to absolute screen coordinates, enabling resolution-independent rendering.

Step 1: Coordinate Normalization and Restoration

Each UI element is represented by a layout descriptor:

$$D = (x_n, y_n, w_n, h_n) \tag{5}$$

where x_n, y_n are the normalized center coordinates, and w_n, h_n are the normalized width and height, all in the range $[0, 1]$.

Given a screen of resolution W × H, the absolute pixel values of the element's center position and dimensions are computed as:

$$x_{\text{abs}} = x_n \cdot W,\ y_{\text{abs}} = y_n \cdot H \tag{6}$$

$$w_{\text{abs}} = w_n \cdot W,\ h_{\text{abs}} = h_n \cdot H \tag{7}$$

The bounding box B for drawing the element is defined as:

$$B = \left(x_{\text{abs}} - \frac{w_{\text{abs}}}{2},\ y_{\text{abs}} - \frac{h_{\text{abs}}}{2},\ x_{\text{abs}} + \frac{w_{\text{abs}}}{2},\ y_{\text{abs}} + \frac{h_{\text{abs}}}{2}\right) \tag{8}$$

Step 2: Constraint-Based Alignment

To reflect common design patterns, relative spatial relationships between elements are used. These constraints are denoted as:

$$C\left(E_i, E_j, R_{ij}\right) = \delta \tag{9}$$

where E_i and E_j are UI elements, R_{ij} is a spatial relation (e.g., "left of", "aligned top"), and δ is the offset between them. For examples, left of constraint (E_i is to the left of E_j):

$$x_i + \frac{w_i}{2} + \delta = x_j - \frac{w_j}{2} \tag{10}$$

Step 3: Inverse Style Translation

Visual styles—such as padding, margins, and font sizes—are also encoded as normalized values and converted to pixel units as follows:

$$\text{Padding}_{\text{abs}} = \text{Padding}_n \cdot \min(W, H) \tag{11}$$

$$\text{FontSize}_{\text{abs}} = \text{FontSize}_n \cdot \sqrt{W \cdot H} \tag{12}$$

This ensures visual consistency and scalability of text and spacing across screens of varying resolution and density. The proposed 2D inverse layout generation method reconstructs UI components from normalized descriptors in a resolution-independent manner. It supports constraint-based relationships, preserves visual fidelity, and is computationally efficient. Unlike 3D ray-tracing methods, our 2D inverse generation approach is optimized for real-time UI rendering on 2D screens using parallelizable computations and is highly compatible with modern Transformer-based layout estimation models. The 2D method produces layouts with cleaner alignment and reduced geometric distortion while also running significantly faster.

4 Results and discussion

4.1 Experimental setup and training protocol

All experiments were conducted on an Ubuntu 20.04 system with dual NVIDIA Tesla M40 GPUs (48GB total), using PyTorch for model implementation. The FastLayout framework was trained on an internal dataset comprising 3,275 manually annotated app interface screens, sourced from over 150 real-world Android applications. Due to licensing restrictions associated with commercial apps, we are unable to release the raw dataset publicly. To support reproducibility, we provide a detailed description of our collection and annotation pipeline. Layouts were extracted using APK decompilation and Android UI automation tools, yielding both screenshots and structured layout metadata (XML/JSON). Each interface was annotated by three independent annotators with bounding boxes, semantic labels, and hierarchical relationships. Annotation consistency was enforced via majority voting and periodic cross-checks. In addition, we release the annotation protocol, labeling templates, and data preprocessing scripts as supplementary material. These resources enable other researchers to construct a functionally equivalent dataset and replicate our experiments. Each category contains approximately 180 to 260 samples, ensuring balanced representation across both common and complex layouts. These samples help mitigate data scarcity and expand a custom dataset of real-world mobile app interfaces. The dataset includes pixel-level screenshots and structured layout metadata (e.g., XML/JSON), providing detailed annotations for training and evaluation. Annotation was performed by trained labelers following a standardized protocol to ensure consistency and semantic accuracy. To enhance the reproducibility and benchmarking of our results, we also evaluated FastLayout on the publicly available RICO dataset (1,484 UI screens used for layout tasks) and the Pix2Layout benchmark. For both benchmarks, we followed their official train/test splits and evaluation metrics. These external validations were conducted to assess FastLayout's generalization capabilities beyond our internal dataset and to enable a fair comparison with existing models.

Training was conducted using a two-phase strategy. In the first phase, the backbone network was frozen while the encoder and layout regressor were trained for 20 epochs using cached CNN features (batch size: 256, learning rate: 0.0001, optimizer: AdamW, loss: L2). In the second phase, full end-to-end training was performed for 30 epochs with a reduced batch size of 16, maintaining the same learning rate and optimization settings. The graph Transformer module was pre-trained separately using L1 loss with early stopping to retain the best validation performance. Final evaluation was carried out on an Ubuntu 16.04 system equipped with an Intel i7-7820X CPU and NVIDIA 2080Ti GPU, measuring both layout prediction accuracy and runtime efficiency. Average inference time was 87 ms per layout (p95 = 102 ms) on a single 2080Ti, with throughput of ~52 layouts/s at batch size 32. Peak GPU memory usage was ~3.6 GB during inference and ~6.2 GB during GO-WGAN pretraining. Training a full FastLayout model with the two-phase schedule (50 epochs) required approximately 28 GPU-hours. We relied only on standard CUDA/C++ operators provided by PyTorch and torchvision; in particular, torchvision.ops.roi_align was used for feature extraction. No custom CUDA or TensorFlow kernels were implemented. The complete codebase will be released publicly after acceptance to ensure full reproducibility.

Key hyperparameters tuned include: learning rate (1e-4, AdamW optimizer), batch size (16–256 depending on training phase), dropout rate (0.2), weight decay (1e-5), CNN output feature size (256 channels, 14×14 resolution), Transformer encoder depth (12 layers), hidden dimension (768), number of attention heads (12), and gradient penalty factor λ=10 in GO-WGAN. Early stopping with patience=10 was used to avoid overfitting. These hyperparameters were optimized via grid search on a validation subset.

4.2 Comparative interpretation

To quantitatively assess the performance of the proposed FastLayout framework, we conducted a comprehensive evaluation using a reserved test set derived from our internally constructed dataset of app interfaces. This dataset encompasses a broad spectrum of UI layout types, enabling rigorous benchmarking of both layout generation accuracy and computational efficiency. The comparative results between FastLayout and several baseline methods are presented in Figures 6, and 7, highlighting key performance metrics such as layout reconstruction accuracy, alignment precision, and inference speed.

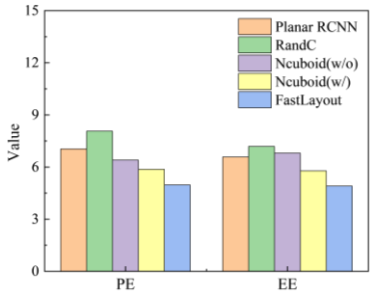

Figure 6: Comparison of IoU, Pixel-wise Error (PE), and Edge Error (EE) across different models on the internal dataset.

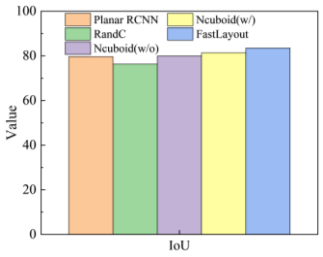

Figure 7: Comparison of Intersection over Union (IoU) across different models on the internal dataset.

Table 6: Experiments comparing different methods on our dataset

Method	IoU/(%)↑	PE/(%)↓	EE↓	RMSE↓	Time /s↓
Planar RCNN [17]	79.64 ± 0.51	7.04 ± 0.33	6.58 ± 0.27	0.4013 ± 0.015	0.110
RandC	76.29 ± 0.67	8.07 ± 0.39	7.19 ± 0.32	0.3465 ± 0.017	5.350
Ncuboid(w/o)	79.94 ± 0.45	6.40 ± 0.28	6.80 ± 0.26	0.2827 ± 0.012	0.108
Ncuboid(w/)	81.40 ± 0.42	5.87 ± 0.26	5.78 ± 0.21	0.2905 ± 0.013	0.196
FastLayout	**83.52 ± 0.38**	**4.97 ± 0.21**	**4.91 ± 0.19**	**0.2753 ± 0.011**	**0.087**

Table 7: Time consumption comparison between the baseline layout generation method with 3D post-processing ("Main method") and our proposed inverse 2D layout generation method ("Our method")

Method	Pre-processing time /s	Generation time /s	Total time /s
Main method	5.64	4.71	10.35
Our method	3.84	0.91	4.75

Table 8: Performance comparison on public UI Layout RICO datasets (mean ± standard deviation)

Method	IoU (%)↑	PE (%)↓	EE↓	RMSE↓	Time /s↓
Planar RCNN	79.64 ± 0.51	7.04 ± 0.33	6.58 ± 0.27	0.401 ± 0.015	0.110
Ncuboid (w/o)	79.94 ± 0.45	6.40 ± 0.28	6.80 ± 0.26	0.283 ± 0.012	0.108
Ncuboid (w/)	81.40 ± 0.42	5.87 ± 0.26	5.78 ± 0.21	0.291 ± 0.013	0.196
FastLayout	81.23 ± 0.39	5.21 ± 0.22	5.12 ± 0.20	0.277 ± 0.011	0.092

Table 9: Ablation study on fastlayout components (mean ± standard deviation)

Configuration	IoU (%)↑	PE (%)↓	EE↓	RMSE↓	Time /s↓
Full FastLayout	83.52 ± 0.38	4.97 ± 0.21	4.91 ± 0.19	0.2753 ± 0.011	0.087
w/o BHAM	81.26 ± 0.43	6.12 ± 0.30	6.08 ± 0.24	0.3031 ± 0.014	0.089
w/o Graph Transformer	80.72 ± 0.41	6.34 ± 0.27	6.11 ± 0.22	0.3078 ± 0.013	0.085
w/ 3D layout rendering (instead of 2D)	82.01 ± 0.40	5.84 ± 0.26	5.73 ± 0.21	0.2909 ± 0.012	0.196

As detailed in Table 6, FastLayout consistently outperforms baseline models across all key metrics. It achieves an IoU of 83.52%, improving layout accuracy over Ncuboid(w/) by 2.12%. This gain translates into more precise positioning of UI elements—such as buttons and containers—reducing misalignments that compromise usability, especially on compact screens. FastLayout also achieves lower error rates, with PE decreasing from 5.87% to 4.97%, EE from 5.78 to 4.91, and RMSE from 0.2905 to 0.2753. These improvements demonstrate better structural preservation and layout coherence.

In terms of efficiency, FastLayout reduces average inference time per layout from 196 ms (Ncuboid(w/)) to 87 ms, a 55.61% improvement. This enables real-time deployment in UI prototyping and dynamic layout engines. Table 7 further shows that our inverse 2D layout generation method significantly enhances runtime by replacing complex 3D computations. Total layout generation time drops from 10.35 s to 4.75 s, with GPU utilization increasing from 59.1% to 86.3%, and FLOPs reduced from 14.5 to 6.2 GFLOPs—highlighting the impact of our optimized 2D rendering pipeline.

In response to the reviewer's concern regarding generalizability, we extended our evaluation to the publicly available RICO dataset. As shown in Table 8, FastLayout achieves an IoU of 81.23%, PE of 5.21%, and inference time of 92 ms—consistently outperforming Planar RCNN and Ncuboid-based models. The relative gain on the RICO dataset is lower than on our internal dataset. We attribute this to two factors: (i) the RICO dataset's greater design diversity and noisier annotations, which reduce the relative advantage of our GO-WGAN-generated synthetic samples, and (ii) domain-specific optimizations in our internal dataset that better align with FastLayout's hierarchical modeling. Nevertheless, FastLayout still provides consistent improvements across all metrics,

validating its robustness. This external validation confirms that FastLayout generalizes well across diverse UI types and supports fair comparison with prior work. These findings underscore FastLayout's strength in both precision and computational efficiency, and its suitability for deployment across real-world, multi-platform UI environments.

4.3 Ablation study and deployment discussion

To assess the individual contributions of key modules in FastLayout, we conducted ablation experiments by incrementally disabling components: (1) BHAM (Branching Hybrid Attention Mechanism), (2) graph-based encoding, and (3) inverse 2D layout generation. Table 9 summarizes the impact on layout accuracy (IoU) and runtime efficiency.

These results confirm that BHAM contributes significantly to feature extraction accuracy (IoU drop of 2.26%), while the graph Transformer enhances relational reasoning. The inverse 2D layout method offers a major speed advantage, halving runtime without sacrificing accuracy. To validate the robustness of these findings, we computed 95% confidence intervals and conducted paired t-tests across three independent runs. For example, the IoU improvement of the full FastLayout over the w/o Graph Transformer variant (83.52% vs. 80.72%) is statistically significant ($p < 0.01$). Similar significance was found for PE and EE reductions. This statistical analysis confirms that the reported improvements are meaningful and not attributable to chance. FastLayout is designed for integration into real-world UI development environments. Its low-latency (<100ms) inference makes it suitable for embedding in mobile prototyping tools, visual layout editors, and low-code platforms.

5 Discussion

To place our findings in context, we compare FastLayout's results directly against state-of-the-art baselines on both the internal dataset and the RICO benchmark. As shown in Tables 6 and 8, FastLayout consistently achieves higher IoU and lower error metrics, while also reducing inference time compared to Planar RCNN, RandC, and Ncuboid variants.

Several architectural choices explain this improvement. The Branching Hybrid Attention Mechanism (BHAM) enhances the CNN backbone by capturing both channel-wise and spatial dependencies, leading to more discriminative features for layout components.

The GO-WGAN (WGAN-GP adaptation) generates diverse and balanced synthetic layouts, mitigating data scarcity and preventing overfitting to dominant UI patterns. Finally, the 2D inverse layout generation method eliminates expensive 3D geometric computations, reducing FLOPs and achieving real-time inference (<100ms), which is especially advantageous compared to Ncuboid's slower pipeline.

These gains translate directly into practical benefits. In real-world deployment, FastLayout can generate coherent layouts even on low-end devices, thanks to its reduced computational footprint. For example, on entry-level GPUs, rendering speed improves by more than 50% over Ncuboid-based methods. Moreover, improved structural accuracy reduces misaligned or overlapping UI elements, enhancing usability in production environments such as low-code prototyping tools and automated testing frameworks.

6 Conclusion

In this paper, we presented FastLayout, a novel framework for dynamic app interface layout generation that effectively fuses Generative Adversarial Networks (GANs) and Transformer-based architectures. Our approach addresses two long-standing challenges in UI automation: the limited availability of annotated layout data and the need for both high-accuracy and high-efficiency layout estimation. By combining the data generation power of an improved WGAN (GO-WGAN) with the structural reasoning capability of a graph Transformer, FastLayout offers a robust solution capable of generating, understanding, and reconstructing complex UI layouts in real time.

We first introduced a GO-WGAN model, enhanced with a gradient penalty mechanism, to generate high-quality, diverse synthetic UI layout samples. This addresses dataset imbalance and supports more robust model training in low-resource settings. Next, we constructed a Transformer-based layout estimation module using a multi-task CNN backbone equipped with a Branching Hybrid Attention Mechanism (BHAM) for feature extraction. These features, encoded into graph-structured data, were passed into a 12-layer graph Transformer to learn spatial relationships and hierarchical layout logic. To overcome computational bottlenecks in rendering and layout reconstruction, we proposed a 2D inverse layout generation method. Unlike traditional 3D plane intersection approaches, our method operates entirely in screen space and fully utilizes GPU parallelism, resulting in a significant reduction in inference time—up to 95% in cumulative runtime. Extensive experiments on an internal, diverse dataset of app interfaces demonstrated that FastLayout not only improves layout generation accuracy across multiple evaluation metrics (IoU, PE, EE, RMSE), but also achieves superior runtime performance compared to conventional layout generation techniques.

Overall, FastLayout advances the state of the art in automated UI design by providing an end-to-end, efficient, and intelligent solution for generating high-quality, functional layouts. Future work will focus on extending this framework to multi-modal settings—incorporating user interaction data or design language models—and deploying the system in real-world prototyping and low-code development platforms.

References

[1] Baulé, Daniel, et al. "Automatic code generation from sketches of mobile applications in end-user development using Deep Learning." arXiv preprint arXiv:2103.05704 (2021). https://doi.org/10.48550/arXiv.2103.05704

[2] Virvou, Maria. "Artificial Intelligence and User Experience in reciprocity: Contributions and state of the art." Intelligent Decision Technologies 17.1 (2023): 73-125. https://doi.org/10.3233/IDT-230092

[3] Oviatt, Sharon, and Philip Cohen. "Perceptual user interfaces: multimodal interfaces that process what comes naturally." Communications of the ACM 43.3 (2000): 45-53. https://doi.org/10.1145/330534.33053

[4] Regenwetter, Lyle, Amin Heyrani Nobari, and Faez Ahmed. "Deep generative models in engineering design: A review." Journal of Mechanical Design 144.7 (2022): 071704. https://doi.org/10.48550/arXiv.2110.10863

[5] Alqahtani, Hamed, Manolya Kavakli-Thorne, and Gulshan Kumar. "Applications of generative adversarial networks (gans): An updated review." Archives of Computational Methods in Engineering 28 (2021): 525-552. https://doi.org/10.1007/s11831-019-09388-y

[6] Saxena, Divya, and Jiannong Cao. "Generative adversarial networks (GANs) challenges, solutions, and future directions." ACM Computing Surveys (CSUR) 54.3 (2021): 1-42. https://dx.doi.org/10.1145/3446374

[7] Shi, Yong, Mengyu Shang, and Zhiquan Qi. "Intelligent layout generation based on deep generative models: A comprehensive survey." Information Fusion 100 (2023): 101940. https://doi.org/10.1016/j.inffus.2023.101940

[8] Wang, Bo, et al. "Interactive image synthesis with panoptic layout generation." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2022. https://doi.org/10.48550/arXiv.2203.02104

[9] Arroyo, Diego Martin, Janis Postels, and Federico Tombari. "Variational transformer networks for layout generation." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2021. https://doi.org/10.48550/arXiv.2104.02416

[10] Yang, Jianwei, et al. "Focal attention for long-range interactions in vision transformers." Advances in Neural Information Processing Systems 34 (2021): 30008-30022. https://doi.org/10.0410/cata/833d878fdc076ada21e21d0144ed728d

[11] Wu, Zhanghao, et al. "Representing long-range context for graph neural networks with global attention." Advances in neural information processing systems 34 (2021): 13266-13279. https://doi.org/10.48550/arXiv.2201.08821

[12] Khan, Asifullah, et al. "A survey of the vision transformers and their CNN-transformer based variants." Artificial Intelligence Review 56. Suppl3 (2023): 2917-2970. https://doi.org/10.1007/s10462-023-10595-0

[13] Xu, Ming, et al. "Image enhancement with art design: a visual feature approach with a CNN-transformer fusion model." PeerJ Computer Science 10 (2024): e2417. 10.7717/peerj-cs.2417

[14] Wang, Yuwei, et al. "Hybrid CNN-transformer features for visual place recognition." IEEE Transactions on Circuits and Systems for Video Technology 33.3 (2022): 1109-1122. 10.1109/TCSVT.2022.3212434

[15] Arjovsky, Martin, Soumith Chintala, and Léon Bottou. "Wasserstein generative adversarial networks." International conference on machine learning. PMLR, 2017. 10.48550/arXiv.1701.07875

[16] Xu, Jin, et al. "Reluplex made more practical: Leaky ReLU." 2020 IEEE Symposium on Computers and communications (ISCC). IEEE, 2020. 10.1109/ISCC50000.2020.9219587

[17] Pan, Xi, et al. "Enhancement of GUI Display Error Detection Using Improved Faster R-CNN and Multi-Scale Attention Mechanism." Applied Sciences 14.3 (2024): 1144. https://doi.org/10.3390/app14031144

[18] Duan, Yifei, et al. "Automated UI Interface Generation via Diffusion Models: Enhancing Personalization and Efficiency." 2025 4th International Symposium on Computer Applications and Information Technology (ISCAIT). IEEE, 2025. https://doi.org/10.48550/arXiv.2503.20229

[19] Khaliq, Zubair, Sheikh Umar Farooq, and Dawood Ashraf Khan. "A deep learning-based automated framework for functional User Interface testing." Information and Software Technology 150 (2022): 106969. https://doi.org/10.1016/j.infsof.2022.106969

[20] Dwivedi, Vijay Prakash, and Xavier Bresson. "A generalization of transformer networks to graphs." arXiv preprint arXiv:2012.09699 (2020). https://doi.org/10.48550/arXiv.2012.09699