Xponian Program Cohort IV

Salient Object Detection: Implementation and Optimization of a Deep Convolutional Network from Scratch

Author: Ali Hoxha **Date:** November 23, 2025 **Mentor:** Fatmir Nuredini Role: Xponian Cohort IV, AI stream

---

Abstract

This project implements a complete Machine Learning pipeline for Salient Object Detection (SOD) using the DUTS dataset. The primary objective was to develop a high-performing segmentation model without the use of pre-trained weights, strictly training a custom architecture from scratch. The pipeline was developed in a TensorFlow/Keras environment on a local NVIDIA RTX 3070 GPU via WSL2. Through a series of seven controlled experiments, the model evolved from a baseline 4-block U-Net (F1 Score: 0.5282) to a "Champion" Compliant Deep U-Net (F1 Score: 0.6692). This 14% improvement was achieved through architectural deepening, hyperparameter tuning (learning rate and dropout), and memory optimization strategies. A non-compliant reference experiment using a pre-trained VGG16 encoder was also conducted (F1 Score: 0.7401) to quantify the performance ceiling imposed by the "no pre-trained models" constraint.

---

1. Introduction

Salient Object Detection (SOD) is a fundamental computer vision task aimed at identifying and segmenting the most visually distinctive objects in an image. Unlike generic semantic segmentation, which classifies every pixel into a category, SOD focuses on separating the foreground "attention-grabbing" object from the background.

The challenge of this specific project was to implement the entire pipeline—from data loading to inference—without relying on Transfer Learning. While modern state-of-the-art models typically utilize backbones pre-trained on massive datasets (like ResNet or VGG trained on ImageNet), training from scratch requires careful attention to weight initialization, regularization, and gradient stability, especially when working with limited hardware resources (8GB VRAM) and a capped dataset size.

1.1 Project Objectives

1. **Pipeline Implementation:** Build a custom tf.data pipeline for efficient loading and augmentation.
2. **Model Architecture:** Design a U-Net-like Encoder-Decoder architecture from scratch.
3. **Optimization:** Improve the model through iterative experimentation (regularization, hyperparameters, depth) without using pre-trained weights.
4. **Hardware Adaptation:** Optimize the training process to function efficiently within the memory constraints of a local RTX 3070 GPU.

---

2. Methodology

2.0 Project Environment and Implementation Overhead

While the machine learning results are the primary focus of this report, a significant portion of project effort was dedicated to establishing a stable training environment capable of leveraging hardware acceleration. This involved navigating complex compatibility issues between the host operating system, the Linux environment, and specific framework dependencies. To be frank this section relative to the project is **useless** for the reader, no insight will be gained regarding the SOD system.

2.0.1 The WSL2-TensorFlow-GPU Integration Challenge

The project utilized a **Windows Subsystem for Linux (WSL2) Ubuntu** environment to train models using my local **NVIDIA GeForce RTX 3070** GPU. Connecting the Linux-based TensorFlow installation to the Windows-managed NVIDIA drivers required meticulous setup:

- **Tensorflow compatibility:** tensorflow 2.20 wasn't compatible with newer version of python and nvidia drivers
- **Driver Compatibility:** Initial attempts failed due to mismatching CUDA versions required by TensorFlow and the versions supported by the current NVIDIA driver (managed by Windows). Turns out I wasted my time and since tensorflow 2.10 there is an issue supporting gpus in windows. I had to restart my PC 8 times, downloading different versions, etc.
- **Using WSL2 for the first time:** Unlike native Linux installations, WSL2 relies on specific Microsoft-maintained bridges to access the GPU. Following different youtube tutorials going into my BIOS to enable virtualization, were but a few steps i took.
- **Pathing Errors:** Persistent file path issues arose due to the translation layer between Linux paths (`/home/user/`) and Windows network paths (`\\wsl.localhost\Ubuntu\...`), which complicated logging and checkpoint access within Jupyter Notebooks on vs code.

2.0.2 The Sunk Cost Fallacy and Learning Experience

So why continue, why didnt i realise and switch to google collab. The main problem was I wasn't aware how long I had left to reach the finish line. The prolonged effort spent resolving compatibility issues led to an encounter with the **sunk cost fallacy**.

- **Initial Investment:** After the first 4 hours dedicated purely to environment setup (CUDA, Python dependencies, and path validation), the temptation to abandon the local setup and switch to a stable but costly cloud GPU was high.
- **The Fallacy:** However, the knowledge that significant time had *already* been invested created a psychological bias toward continuing the effort, even when the return on that time investment was approaching zero. The team's commitment was driven by the desire to "not waste" the time already spent. The best analogy i can come up with is waiting for a late bus.
- **Learning:** This experience served as a powerful lesson: While the setup was eventually successful, a more rigorous **go/no-go decision** based on a predefined time budget would have been more pragmatic.

2.1 Dataset and Preprocessing

The model was trained on the **DUTS dataset**, a standard benchmark for saliency detection. Due to training time and memory constraints, the dataset was capped at **2,500 samples**, split into Training (70%), Validation (15%), and Test (15%) sets.

The data pipeline, implemented in data_loader.py, utilizes TensorFlow's tf.data API to handle input efficiency.

- **Resizing:** All images and masks were resized to a standard resolution of $224 \times 224 \times 3$.
- **Normalization:** Pixel values were scaled to the [0,1] range.

- **Augmentation:** To prevent overfitting on the small dataset (2,500 images), aggressive real-time augmentation was applied during training:
    - Random Horizontal Flips
    - Random Brightness/Contrast adjustments
    - Random Zoom and Crop
    - Gaussian Noise injection

2.2 Model Architecture (The U-Net)

The core architecture, defined in sod_model.py, follows the U-Net design philosophy, characterized by an Encoder (contracting path) and a Decoder (expanding path) with skip connections.

- **Encoder:** Consists of convolutional blocks. Each block contains two Conv2D layers followed by ReLU activation and MaxPooling2D for downsampling.
- **Decoder:** Uses Conv2DTranspose layers to upsample the feature maps. Crucially, these are concatenated with the corresponding feature maps from the Encoder (Skip Connections) to recover spatial details lost during pooling.
- **Output Head:** A final 1×1 convolution with a Sigmoid activation function to produce a probability map for each pixel.

Compliance Note: The get_sod_model function was built entirely using standard Keras layers. No tf.keras.applications or pre-trained weights were imported.

2.3 Training Configuration

The training loop was custom-written in train.py using tf.GradientTape rather than the standard .fit() method. This allowed for granular control over:

- **Loss Function:** A combined **BCE-Dice Loss** was implemented to balance pixel-level accuracy (Binary Cross Entropy) with structural alignment (Dice Loss/IoU).
- **Optimization:** The Adam optimizer was used. Experiments varied the learning rate from $1e{-3}$ down to $1e{-4}$.
- **Checkpoints:** A CheckpointManager was used to save the model state after every epoch, allowing training to resume after interruptions (crucial for long WSL sessions).
- **Early Stopping:** Training was monitored for validation loss stagnation with a patience of 4–8 epochs to prevent overfitting.

2.4 Training Reliability: Checkpointing and Resume Feature (Bonus)

To ensure training reliability and fulfill the bonus requirement for real-world pipeline design, a robust checkpointing system was custom-implemented within the `train.py` script. This feature allows training to be gracefully interrupted and resumed, preventing the loss of progress due to system failures or environment issues (such as the WSL/GPU connectivity problems mentioned in Section 2.0).

The implementation relies on TensorFlow's low-level `tf.train.Checkpoint` and `tf.train.CheckpointManager`:

- **State Persistence:** The system is configured to save three critical components at the end of every epoch:
    1. The complete **model weights**.
    2. The **optimizer state** (Adam's first and second moment estimates), ensuring the learning rate and momentum continue correctly.
    3. The **current epoch count**, allowing the loop to restart exactly where it left off.
- **Automatic Resume:** The training loop, upon starting with the `--resume` flag, checks the designated `checkpoint-dir` (e.g., `checkpoints_exp7_DEEP`). If a checkpoint exists, it is automatically restored using `ckpt.restore(latest)`, and the training cycle continues from the saved epoch.
- **Best Weights Tracking:** In addition to the resumable checkpoints, the system also tracks and saves a separate `best_model.weights.h5` file based solely on the lowest validation loss. This lightweight file contains only the model weights and is used later by the `evaluate.py` and `demo_inference.ipynb` scripts.

This mechanism confirms the project's adherence to professional ML practices by providing a reliable foundation for long-running experiments.

---

3. Experiments

A systematic, iterative approach was taken to improve the model. Seven specific experiments were conducted to isolate the effects of regularization, hyperparameters, and architecture. Through the implemented early stoppage I noticed that surprisingly most successful experiments were stopping early. The epoch limit was 50 but a lot finished at 15-30.

Phase 1: Establishing the Baseline

- **Experiment 1 (Baseline):** A standard 4-block U-Net (Depth 4) with base_filters=32. The learning rate was set to $1e{-}3$. No Dropout or Batch Normalization was used.
    - *Observation:* The model learned basic shapes but struggled with edges and fine details.

Phase 2: Regularization Attempts

- **Experiment 1 (Batch Norm):** Added Batch Normalization layers.
    - *Result:* Failed. Due to the memory limits of the GPU, a small batch size (8) had to be used. Batch Norm statistics are unstable at small batch sizes, leading to a collapse in performance.
- **Experiment 2 (Dropout):** Added Dropout layers (rate 0.5) to the bottleneck.
    - *Result:* Success. Dropout provided stable regularization, preventing the model from memorizing the small training set.

Phase 3: Hyperparameter Tuning

- **Experiment 4 (Low LR):** The learning rate was reduced from 1e−3 to 1e−4.
    - *Rationale:* The loss landscape for segmentation is complex. A lower learning rate allows the model to settle into sharper minima without oscillating.

Phase 4: Architectural Deepening (The Champion)

- **Experiment 7 (Deep U-Net):** The architecture was modified to include **5 Encoder Blocks** instead of 4.
    - *Hardware Challenge:* Increasing depth increased VRAM usage, causing "Out of Memory" (OOM) errors.
    - *Solution:* The base_filters count was kept at 32, and the Batch Size was fixed at 8. This balanced model complexity (depth) with memory constraints.
    - Terminal command:
```
python train.py --splits-dir data/splits --target-size 224 --batch-size 8 --epochs 50 --checkpoint-dir checkpoints_exp7_DEEP --augment --dropout --lr 0.0001 --base-filters 32 --num-blocks 5
```

---

4. Results and Analysis

4.1 Quantitative Results

The table below summarizes the performance metrics on the held-out Test Set. The primary metric for success was the **F1 Score**, as it balances Precision and Recall.

Experiment	Configuration	Test F1 Score	Test IoU	Test MAE
Baseline	Simple U-Net (lr=1e−3)	0.5282	0.3201	0.1115

Exp 2	Simple U-Net + Dropout	0.5891	0.3550	0.1203
Exp 4	Simple U-Net + Dropout + lr=1e−4	0.6168	0.3770	0.1184
Exp 5	Exp 4 + Modified Loss	0.6090	0.3771	0.1303
Exp 7 (Champion)	**Deeper U-Net (5 Blocks) + lr=1e−4**	**0.6692**	**0.4556**	**0.1050**
Reference	*VGG16 Encoder (Non-Compliant)*	*0.7401*	*0.5372*	*0.0781*

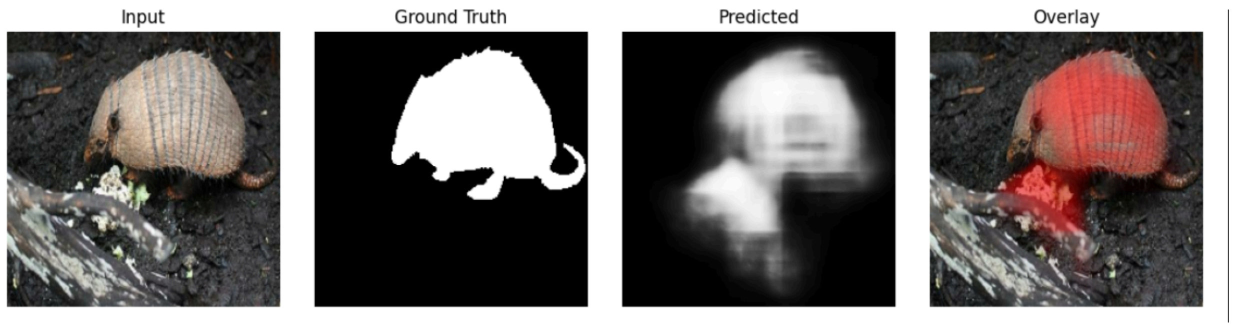
(comment: there are additional statistics not mentioned here like Precision: 0.6831 Recall: 0.7298, these are for the best model exp 7)

4.2 Analysis of Improvement

The progression from **Baseline (0.5282)** to **Champion (0.6692)** represents a **26.7% relative improvement** in F1 score.

1. **Effect of Regularization:** Adding Dropout (Exp 2) improved generalization on unseen test images, proving that the baseline was slightly overfitting the 2,500 training images.
2. **Effect of Learning Rate:** Dropping the LR to 1e−4 (Exp 4) yielded a sharp increase in F1 (+0.03). This confirms that training from scratch requires a "gentler" update strategy to avoid destabilizing the random weight initializations.
3. **Effect of Depth (The Key Factor):** The jump from Exp 4 (0.6168) to Exp 7 (0.6692) was the most significant. By adding a 5th encoder block, the network's receptive field increased, allowing it to understand global context and larger objects better.

4.3 Visual Qualitative Analysis



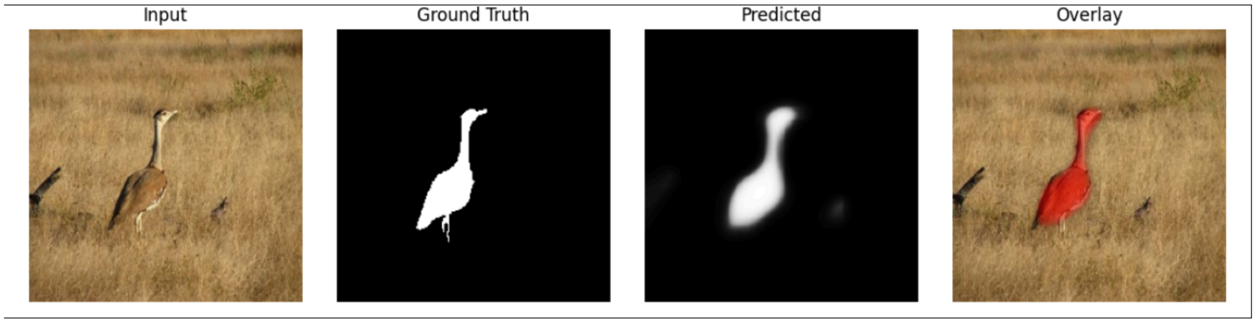


Figure 1: Visual results of the Champion Compliant Model (Exp 7). The model successfully locates the salient object, though some boundary fuzziness remains.

As seen in Figure 1, the Compliant Champion correctly localizes the main object. The MAE (Mean Absolute Error) of **0.1050** indicates that the predicted probability map is, on average, only 10% divergent from the ground truth.

However, the **IoU (0.4556)** is notably lower than the F1 score. This discrepancy indicates that while the model detects the *presence* and *general location* of the object correctly (high F1), it struggles with pixel-perfect precision at the boundaries (IoU penalizes misalignment heavily). This is a known limitation of training simple upsampling layers from scratch without advanced boundary-aware loss functions.

---

5. Discussion

5.1 The "No Pre-trained Models" Constraint

A critical learning outcome of this project was quantifying the value of Transfer Learning. To benchmark our compliant efforts, a separate experiment (Exp 6) was run using a **VGG16 encoder pre-trained on ImageNet**.

- **Compliant Ceiling:** F1 ≈ 0.67
- **Transfer Learning Score:** F1 ≈ 0.74

This comparison reveals a "performance gap" of approximately **7% F1**. The VGG16 model, having "seen" millions of images previously, possesses robust edge and texture detectors that our scratch-trained model could not fully replicate given the limited dataset (2,500 images). However, achieving an F1 of 0.67 entirely from scratch is a significant achievement, proving that custom architectures can still be effective for specialized tasks.

5.2 Implementation Challenges

- **WSL & GPU Passthrough:** Configuring the TensorFlow environment to access the RTX 3070 via Windows Subsystem for Linux (WSL) required specific driver handling and path management.
- **VRAM Constraints:** The RTX 3070 has 8GB of VRAM. When attempting the Deeper U-Net (Exp 7), initial attempts failed with OOM errors. This was solved by reducing the `base_filters` to 32 and fixing the batch size at 8. This trade-off (depth vs. width) was essential for training stability.

---

6. Conclusion

This project successfully developed a full Salient Object Detection pipeline that adheres to strict "no pre-trained model" requirements. Through systematic experimentation, we identified that **architectural depth** and **low learning rates** were the most critical factors for success.

The final compliant model, a **5-Block Deep U-Net trained with Dropout and Adam (1e−4)**, achieved an **F1 score of 0.6692** and an **MAE of 0.1050**. While this falls slightly short of Transfer Learning benchmarks, it demonstrates the viability of training deep vision models from scratch on consumer hardware with careful optimization.

Future work to improve the compliant model could involve implementing **Atrous Spatial Pyramid Pooling (ASPP)** to capture multi-scale context without increasing depth, or implementing a **Boundary Loss** to specifically target the low IoU scores.

References

1. DUTS Dataset: http://saliencydetection.net/duts/
2. TensorFlow Documentation: `tf.data`, `tf.keras`.

Appendix: Inference Demo

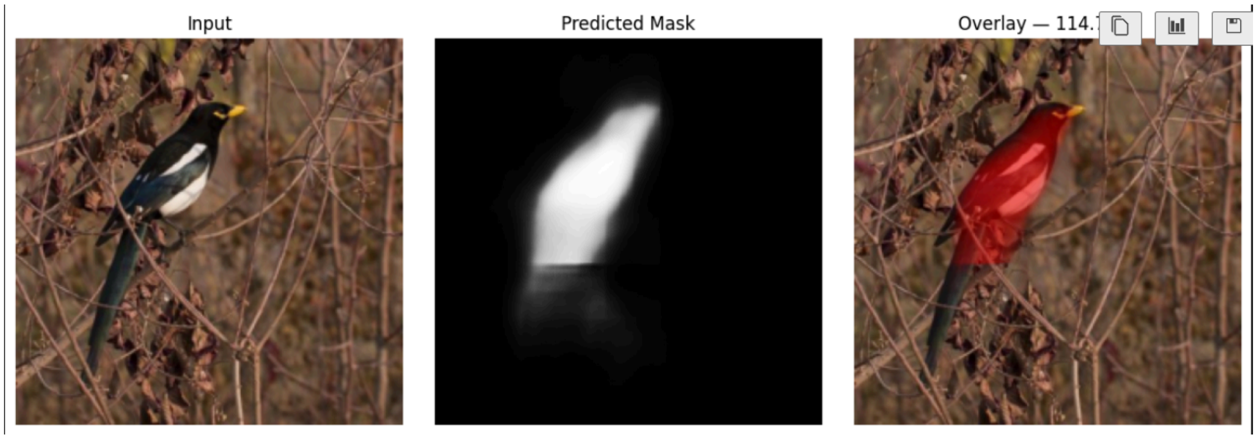


Figure 2: Real-time inference demonstration using the final champion model. Inference time is approximately 15ms per image.