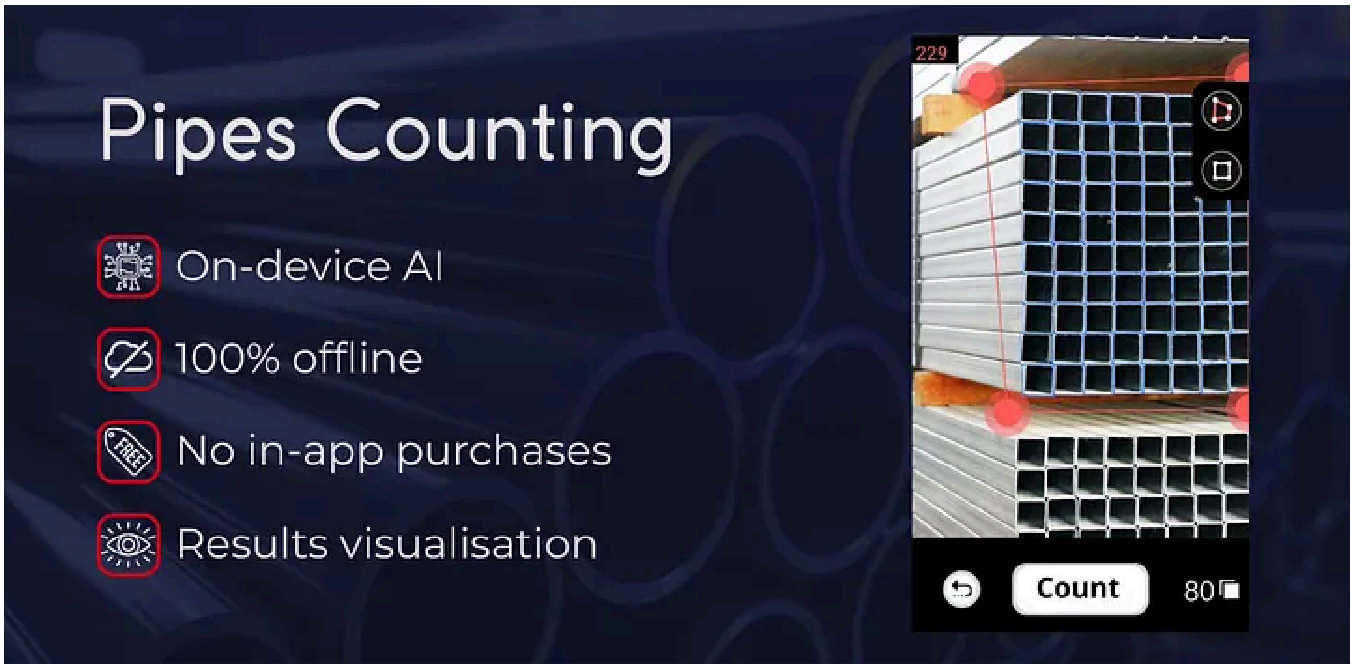
Developing Pipes Counting: How to Build an Offline AI Tool for Metalurgy

8 min read · 6 days ago

Ivan Fenster Follow



In this article, we, 11th-grade students from Letovo School, want to share our experience developing the mobile app **PipesCounting**, created for automated counting of pipes in a bundle. Over 6 months, we went from idea to app publication, including collecting and expanding a dataset, training models, and building the app. Here we want to share our experience and the difficulties we faced to make the process easier for those who will embark on something similar.

With the PipesCounting app, we took second place in the prestigious International AI Junior Contest in the “Metallurgy” category and brought it to the **AI Journey** international conference.

Meet our team:

Ivan Fenster
Alexander Povarov
Ignat Sukhanovskii
Konstantin Spirin
Andrey Bushin

What we wanted to build

We decided to develop an app with a model for counting pipes inside a bundle, applicable in metallurgical production. The model had to operate without an internet connection, balancing recognition quality and speed. It was also important that the model integrated into the app would require few resources: the app is intended for production use, where not all devices have high computing power.

Given these requirements, we chose **YOLOv8**: the model is well-suited for recognizing different objects, it’s easy to fine-tune for a specific class, and its speed allows a good balance between performance and quality.

Figure 1. Types of pipes shapes

Data

We set out to build an app that would recognise four types of pipes, most commonly used in production: round, square, angle, and channel (see Figure 1). For round and square tubes, we found some open-source labeled datasets for model training. However, for angle and channel bars, there were no available labeled open-source datasets at all.

The solution turned out to be simple but labor-intensive: with five people and using **CVAT**, we manually labeled about 12,000 pipes for "angle" and "channel." Nevertheless, even that still wasn't enough for full-fledged training, so we resorted to expanding the dataset, using the following methods:

Augmentation

Due to the limited amount of data, and also to control the number of labeled images, we used augmentation — artificially expanding the dataset with simple transformations of the original images.

For pipe augmentation, we chose the following methods:

- Flip — Doubles the dataset while preserving the logic of pipe stacking.
- Rotation (±15°) — Teaches the model to handle small camera tilts.
- Brightness/Exposure — Imitates shadows and glare in the shop.
- Blur — Makes the model robust to blurry frames from mobile cameras.

- Mosaic — Combines 4 images into one, creating complex scenes.

Synthetic data

Another way to solve the lack of labeled data is to generate synthetic images with pre-defined labels. First, for each synthetic image we chose a resolution and a production background from preselected photos. Then we placed cut-out ends of real pipes according to one of three patterns:

1. **No pattern**

Pipes are randomly rotated and scattered across the image. Helps with recognizing single pipes or pipes outside bundles.

Figure 2. Examples of “no pattern” synthetic images

2. With a simple pattern

Pipes are stacked in rows but not nested into each other — it helps recognize such bundles.

Figure 3. Example of “simple pattern” synthetic images

3. With a complex pattern

Pipes are nested into each other, as in most real bundles.

Figure 4. Examples of “complex pattern” synthetic images

We also applied **CutOut** and **Blur** to pipe images to reduce the risk of overfitting to specific synthetics. A big advantage of synthetic data is instant annotation (bounding boxes), which removes manual annotation from the team.

What did we get?

Here are the sizes of datasets we obtained after collecting and “inflating” the data.

Table 1. Numbers of the images in each dataset

Here are some examples of images from the datasets we obtained.

Table 2. Examples of images in each dataset

Here is the final dataset that we have labeled ourselves and used for training (with augmentation, but no synthetic). For now, it's the only labeled open-sourced dataset with angle and channel bars.
Link: https://universe.roboflow.com/ignat/pipescounting_dataset-t7yey

YOLO training

To make the app effective in production, our team has tried to speed up the model's inference. The first idea was to reduce the input image resolution. Experimentally, we found that when reducing the resolution from 640×640 to 480×480, inference time is cut roughly in half (see Figure 5). At the same time, detection quality obviously drops, so it's important to choose a balance between speed and accuracy.

Figure 5. Visualisation of the inference time reduction after reducing the resolution of an image

In addition, even after artificially increasing the dataset, there were not many unique images, so to reduce the risk of overfitting, we limited training to **50 epochs** on one dataset.

As a result, we obtained the following training results:

Table 3. Results after training

As expected, higher-resolution images provide better quality. Unexpectedly, the synthetic data did not give a gain and, in places, turned out worse than “organic” data. This can be explained by the rather weak realism of the images and, possibly, their uniformity (the background images and pipe ends were limited).

Splitting the image into 4 parts and post-processing:

When analyzing the results, we noticed an interesting pattern: the model confidently handles bundles up to a certain threshold count of pipes, but when it’s exceeded, quality drops sharply — some objects simply stop being detected.

Figure 6. The drop of quality on the large bundles

To tackle this problem, we came out with an idea to split the image into four parts with a small overlap and run YOLO separately on each part (see Figure 7). After that, we found all overlapping pipes by determining the distance from the center of one pipe to the edge of another; among overlapping pipes, we kept only the larger one.

Figure 7. Visualisation of splitting image on four parts with small overlaps

Also, all pipes in a bundle in the photo should be approximately the same size. Therefore, to filter out outliers, we removed those that are significantly larger or smaller than the average size of all pipes in the photo. To do this, we found the median area of all pipes and deleted those whose area differed greatly from the median. This helped stabilize counting on dense stacks and reduce misses on large bundles.

Figure 8. Results of removing the ourliers

Figure 10. Results of the post processing on squares and circles

App development

The next step was to develop an Android mobile app that would allow applying the trained model. The app should let the user take a snapshot or choose a photo from the gallery, pass it to the model, and get a pipe count in a few seconds.

Our team decided to make the app fully offline due to the possible lack of internet in production. This decision was a serious constraint: you can't upload the model to Hugging Face Spaces and call it via API; instead, the model must be directly embedded in the app.

Since the model we used was a fine-tuned YOLO with post-processing in Python, we decided to use the **Kivy** framework for app development. It's not typical for mobile development, but it simplifies integrating the ML part of our product. For building the app itself, we chose the bundled **Buildozer** utility.

For the first month, the chosen strategy seemed successful: the Kivy markup language doesn’t differ much from other markup languages (it resembles a mix of HTML and CSS), and the capabilities were sufficient — access to the camera, to the photo gallery, frame capture, and subsequent processing were implemented without particular problems. Thus, the entire app functionality was successfully implemented step by step (the layout and interface are shown below).

Figure 11. App layout

The last step remained — build the app for Android. First of all, we ran into difficulties specifying the build requirements for less popular libraries, for example, **Ultralytics** (the YOLO library), for which there was no ready “recipe” for installation in Buildozer.

To keep the app size within **100 MB**, we quickly abandoned bundling **PyTorch** into the app (it inflated the size to ~**1.5 GB**). Instead, we used **TensorFlow Lite** (LiteRT), designed specifically for deployment; fortunately, YOLO weights can be imported into this format. With it, the APK size drops back to **70 MB**. However, a new task appears: using YOLO weights outside Ultralytics requires manually implementing pre-/post-processing (including **Non-Maximum Suppression**). We used **OpenCV** for this: building yet another library via Buildozer takes a lot of time and troubleshooting, so we prefer to use libraries already installed. Fortunately, the necessary OpenCV code lies in tutorials in the Ultralytics repository.

Of course, from a finished Python project to ready APK and AAB files uploaded to Google Play, there were still many difficulties: with Android NDK versions, the need to “sign” the resulting file, and differing lists of permissions for gallery access across Android versions. Despite that, a working APK was ultimately obtained.

Conclusions

As a result, we got a fully autonomous mobile app with an embedded YOLO model, capable of offline detection of four types of pipes right on the production site.

Over several months of app development, we manually labeled the data, applied augmentation and synthetic data generation to expand the dataset, trained the YOLO model, developed post-processing, and packaged it into an offline app.

Along the way, we faced countless technical challenges — from dataset imbalance to deployment on limited hardware — yet each one taught us something new about the full AI development cycle.

Today, Pipes Counting stands as a compact, offline AI tool ready for real industrial use. Our next steps include improving synthetic data generation, expanding the range of recognized profiles, and adapting the app for iOS.

Artificial Intelligence · Metallurgy · Computer Vision · Android App Development · Yolo

Follow

Written by Ivan Fenster

6 followers · 2 following

Responses (3)

Bgerby

What are your thoughts?

Klyde Carpizo
2 hours ago

Any repo link? This is a good by the way! Keep it up!

Reply

Ashraf Said
4 hours ago

This is an extraordinarily impressive technical project, especially for 11th-grade students! The journey from dataset collection to deployment demonstrates a rare combination of theoretical understanding and practical engineering. What stands out is... more

Reply

Vadim Korotkikh
16 hours ago

GitHub code repo link?

Reply

Recommended from Medium

In Towards AI by Eivind Kjosbakken

How to Enrich LLM Context to Significantly Enhance Capabilities

Learn how to empower your LLMs by leveraging additional metadata

3d ago 151

In AI Advances by Dr. Leon Eversberg

Efficient Multimodal Document Retrieval With ColQwen2

Learn how to perform state-of-the-art visual document search using multi-vector embeddings and vision-language models

4d ago 194 4

In Generative AI by Avijnan Chatterjee

This Free AI Tool Does What ChatGPT Can’t

Discover Google AI Studio’s hidden features: system prompts, Compare Mode, and real-time screen sharing. Free tutorial for analyzing...

5d ago 109

Maninder Singh

Building Vision Transformers (ViT) from Scratch

With enough data and compute, Vision Transformers (ViT) can out-perform CNNs on large benchmarks. I wanted to understand why — and more…

6d ago 266 3

In The Generator by Thomas Smith

OpenAI Finally Admits the Real Reason it Crippled GPT-5

And what it's doing to make things right

5d ago 868 28

In Artificial Intelligence in Plain English by Alpha Iterations

Build Agentic RAG using LangGraph

A practical guide to build Agentic RAG with complete project code

6d ago 36

See more recommendations