Executive Summary

The purpose of this paper is to describe the approach to automate Pipe Sleuth from structural assessment videos and our method for improving model accuracy with deep learning. The approach described uses Faster R-CNN\textsuperscript{1,2} a CNN-based object detection algorithm, and a Resnet-101\textsuperscript{3} -based feature extraction architecture. Training images were created from real sewer pipe assessment videos. The TensorFlow\textsuperscript{4} based deep learning model was optimized for inference using Intel's OpenVINO™\textsuperscript{5} toolkit. Here, we focus on detecting different types of anomalies in the pipeline assessment videos and examine inference performance with and without OpenVINO optimization.

Introduction

Regular inspection of underground water/sewage pipeline is required in the water utility industry. Inspection helps prioritize periodic maintenance tasks that help avoid pipe leakage, breakage, and blockage, which might result in property damage or safety hazards. Manual inspection itself is cumbersome and time-consuming.

NASSCO's PACP\textsuperscript{6} (Pipeline Assessment Certification Program) is the North American Standard for pipeline defect identification and assessment, providing standardization and consistency to the methods by which pipeline conditions are identified, evaluated and managed. This standard categorizes the anomalies into different types\textsuperscript{7}.

A camera-mounted rover is maneuvered by an operator into the underground pipelines to scan and record videos. The operator notes their observations and generates an inspection log and summary report, highlighting any problems discovered and their locations. Video recordings and summary reports are subsequently reviewed by a quality control department for their accuracy. As the length and number of these pipes are large, and the number of video logs being generated is high, the process of manually reviewing these data is time-consuming and error-prone. This has motivated utility companies to look for ways to make the pipeline inspection process more efficient. Here, we introduce an approach to automate pipeline inspection from the recorded videos using techniques in computer vision and machine learning.
Anomaly Detection via Deep Learning

Deep learning is a class of machine learning algorithms that uses deeply-nested neural network topologies for feature extraction, transformation, and data classification. The field of computer vision has seen a great deal of success leveraging these methods, particularly the convolutional neural network (CNN) for object detection and classification. In this paper, we treat each anomaly type as an object to be detected and classified. CNNs are widely used for detecting objects in images and known for their generalizability, particularly in the case of CNN topologies with many layers, as they are able to learn about low-level features (e.g., lines, edges, angles), and abstract them into high-level features (e.g., curved surfaces, textures, etc.) deep within the network. Figure 1 shows the typical CNN layers.

If we pass an image as input to the network, it is processed and read by various convolutional layers and pooling layers and finally we see the output as an object’s label at the last layer of the network. For each input image, we get a corresponding class label as an output. This technique is used to detect various objects in a given image. CNN divides the image into multiple regions and then classifies each region into various class labels. Therefore, it needs a lot of regions in various shapes and sizes to predict accurately, resulting in a very high computation time. To overcome this bottleneck and reduce the number of regions, Region-based Convolution Neural Network (R-CNN) can be used.

Selection of Object Detection Algorithm and Network Topology

R-CNN, Fast R-CNN and Faster R-CNN are the different algorithms in the R-CNN family. R-CNN extracts different regions from the input image using a “selective search” method and then checks for the object presence in those regions/boxes (Figure 2). First, it extracts these regions, and for each region, CNN is used to extract the specific features. These features are used to detect objects. This process consumes a lot of computation time, as each region is passed to the CNN separately.
Fast R-CNN\textsuperscript{12}, instead of running CNN for each region, runs the CNN once per image and then generates the convolutional feature maps (Figure 3). Selective search is used on these maps, and regions of proposal are extracted. A “Selective search” algorithm identifies the different regions of proposal by identifying the patterns, which forms an object, such as varying scales, colors, textures, and shapes. As the selective search method involves many image-processing techniques and is slow, the computation time for this CNN is still high.

Faster R-CNN\textsuperscript{1,2} replaces the selective search method with Region Proposal Network (RPN\textsuperscript{13}), which makes the algorithm much faster than R-CNN and Fast R-CNN (Figure 4). Faster R-CNN has two networks: one is a region proposal network (RPN) for generating region proposals, and a second one which uses these proposals to detect objects. First it extracts the feature maps from the input image using the convolutional network, and then passes those feature maps to the region proposal network (RPN).

The RPN uses a sliding window on those feature maps, and, for each window, it generates k anchor boxes of different sizes and shapes (Figure 5). For each anchor box, RPN finds the probability of object presence and bounding box coordinates for adjusting the anchors to better fit the object. It returns the object proposals along with their objectness score. Then an ROI pooling layer\textsuperscript{14} is applied to resize all proposals to the same size. These proposals are passed to the fully connected layer and then to the softmax layer\textsuperscript{15} to classify the object, and linear regression layer for the bounding boxes around the object.

We have chosen Faster RCNN with ResNet-101\textsuperscript{16} topology as the feature extractor to detect different types in the extracted video frame from the inspection video footage and to draw a bounding box around them. The input image frame is run through the ResNet101 network to get the feature map. It then passes through the region proposal network and produces the output of bounding boxes for the anomalies and associated class labels in the output layer.

ResNet101\textsuperscript{17} Residual Neural Network, efficiently trained network with 101 layers, won 1st place in the ILSVRC 2015\textsuperscript{18} classification competition with top-5 error rate of 3.57%. It also won the 1st place in the ILSVRC and COCO 2015\textsuperscript{18} competition in ImageNet Detection, ImageNet localization, coco detection and coco segmentation. They observed a relative improvement of 28\% by replacing the VGG-16\textsuperscript{19} layers in Faster R-CNN with ResNet-101. When network depth increases, accuracy gets saturated and then degrades rapidly. ResNet topology solves this degradation problem in deeper networks.
Solution Approach

This solution approach follows six phases (Figure 6):
1. Get images with potential anomalies.
2. Annotate and label the dataset.
3. Convert dataset to the format needed by the TensorFlow framework.
4. Train the model with Faster R-CNN ResNet-101 architecture.
5. Compute evaluation metrics on the validation data set.

Dataset Acquisition

A good dataset is very important in order to train a high-quality model. To collect such a dataset, assessment videos of sewer pipes with all targeted anomalies must be collected. Frames are extracted at a constant skip rate, for example 3 to 5 frames per second, and saved as image files. A sequence of computer vision algorithms was used to filter out frames that are not likely to contain anomalies.

Annotations and Labelling

Nearly 800 videos were used to extract 17,796 images, constituting all sub-categories of Cracks, such as Crack Longitudinal (CL), Crack Circumferential (CC), Crack Hinge (CH), Crack Spiral (CS) and Crack Multiple (CM). Table 1 lists the number of objects annotated of all sub-categories from 17,796 images. The solution is further enhanced with additional anomaly classes. For more information, refer to the Addendum below.

<table>
<thead>
<tr>
<th>Name of Anomaly</th>
<th>Symbol</th>
<th>Number of Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crack Longitudinal</td>
<td>CL</td>
<td>14865</td>
</tr>
<tr>
<td>Crack Circumferential</td>
<td>CC</td>
<td>4041</td>
</tr>
<tr>
<td>Crack Hinge</td>
<td>CH</td>
<td>942</td>
</tr>
<tr>
<td>Crack Spiral</td>
<td>CS</td>
<td>342</td>
</tr>
<tr>
<td>Crack Multiple</td>
<td>CM</td>
<td>3846</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>24036</strong></td>
</tr>
</tbody>
</table>

Table 1. Annotations of Anomalies in Dataset
All images in the dataset were annotated with all sub-categories by drawing bounding boxes around each anomaly found and labeled accordingly. We define bounding box as the minimal rectangle enclosing the whole anomaly, as shown in the Figure 7. Multiple bounding boxes might be used for anomalies having larger size to cover most of the anomaly features, rather than covering the entire image to accommodate the anomaly object in a single bounding box. Annotation information for each image was stored in separate xml files.

**Generating TF Records**

In a TensorFlow training framework, the data is read from its native binary format, TFRecord. The dataset was converted to the binary TFRecord file format, which consolidates all images and their respective labels into a single file. The TFRecord format serializes the data and annotations, and it helps reduce the time taken for data import and preprocessing by allowing TensorFlow to read the data linearly. This is especially helpful for datasets that are too large to be held in memory, as only the data that is needed at the time (e.g., a batch) is loaded from disk.

**Training the Model**

To train the model, we randomly partitioned the dataset into 70% for training, and 30% for evaluation. To help with model training, we used a transfer learning\textsuperscript{21} approach—a method wherein a model developed for a related task is used as the starting point for training a model on a second task. We initialized weights from a Faster R-CNN ResNet101 model pre-trained on the COCO dataset\textsuperscript{22}—a large-scale object detection, segmentation, and captioning dataset, followed by retraining on all layers with our pipeline anomaly image dataset. As our dataset is completely different from the COCO dataset that was used in the pre-trained model, we did not freeze any layer parameters and continued to re-train all the layers. Some model parameters, like the number of anomaly classes (which in our case is 5), image size (width and height), the number of steps for which training must be iterated, and learning rate were modified for training on our dataset. We used TensorFlow’s truncated\textsubscript{normal}\textunderscore initializer to initialize the weights. The initial learning rate was set to the default 0.000300000014249 and the momentum optimizer was used to train. Training continued until 200,000 steps were reached using our image dataset of 17,796 images. Various training parameters such as loss, accuracy, mAP across validation set are to be monitored for determining the status of the training. This is very important to avoid overfitting of the model.

After training, the model was converted into a frozen graph using TensorFlow’s freeze\_graph function, which we used for inference.

![Figure 7. Sample Non-Crack, Crack and Annotated Images](image-url)
Validating the Model

We evaluated the model using the hold-out validation data set with respect to several performance metrics.

Model detection recall is calculated based on the below formula. Usually an object detection solution is evaluated based on its mean average precision value, but here, our focus is on calculating the percentage of the actual anomalies/cracks found against the total anomalies/cracks, and not the precision of the anomaly/crack detected.

The steps for calculating the detection recall form the Confusion Matrix based on the predictions, which gives values of TP, TN, FP, FN.

Where, A True Positive (TP) is an outcome where the model correctly predicts the positive class. Similarly, a True Negative (TN) is an outcome where the model correctly predicts the negative class. A False Positive (FP) is an outcome where the model incorrectly predicts the positive class. A False Negative (FN) is an outcome where the model incorrectly predicts the negative class. Then Recall is calculated by the following formula. Our model achieved ~84% of detection recall.

\[
\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}
\]

Optimization Using OpenVINO

Inference Optimization

Upon re-training of the model, it was optimized using Model Optimizer and used for inferencing on the Intel® CPU architecture. Inference performance optimization for the CPU was done using the Intel® Distribution of OpenVINO™ toolkit (version R4.420).

The Intel Distribution of OpenVINO toolkit is a comprehensive toolkit that you can use to develop and deploy vision-oriented solutions on Intel® platforms. Vision-oriented means the solutions use images or videos to perform specific tasks. It enables CNN-based deep learning inference on the edge to support heterogeneous execution across Intel® Vision Products using a common API for the Intel® processors, Intel® Integrated Graphics, Intel® Movidius™ Neural Compute Stick (NCS), Intel® Neural Compute Stick 2, Intel® Vision Accelerator Design with Intel® Movidius™ VPUs and Intel® FPGAs. It accelerates time-to-market through an easy-to-use library of computer vision (CV) functions and pre-optimized kernels.

Following is a summary of the steps for optimizing and deploying a trained model (Figure 8):

1. Save the trained model as a frozen graph, which can be used as input to the Model Optimizer.
2. Configure the Model Optimizer and convert the frozen graph to IR format, an optimized Intermediate Representation of the trained model.
3. Test the model in the Intermediate Representation format with the test images using the Inference Engine in the target environment via provided Inference Engine validation application or sample applications.
4. Integrate the Inference Engine into the application to deploy the model in the target environment.

Figure 8. OpenVINO™ toolkit workflow for optimizing and deploying a trained deep learning model
The Model Optimizer is a cross-platform command line tool that facilitates the transition between the training and deployment environment. It performs static model analysis and simplifies the network topology by removing any operations not needed for inference, or by combining adjacent operations where possible. The result of this is that the optimized model is smaller and more efficient, decreasing inference time. It produces an Intermediate Representation (IR) of the network, contained in two files:

- “.xml”: Describes the network topology.
- “.bin”: Contains the weights and biases binary data.

The Inference Engine loads the IR and stages it for inference and offers a unified API across supported Intel platforms. It enables deploying your network model trained with any of supported deep learning frameworks: Caffe, TensorFlow, MXNet, Kaldi, or ONNX.

For this use case, the Model Optimizer is used with the default optimization techniques enabled, like Linear Operation Fusing, ResNet Optimization, and etc., to produce the IR files. Then it was evaluated with new images for performance measurement. The average Inference time taken for one image with non-optimized models and with OpenVINO-optimized models on various Intel processor-based platforms are shown in Figure 9. As you can see, inference time was improved by using the OpenVINO-optimized model Wipro observed a significant improvement in inference time in Intel Xeon processors by using OpenVINO toolkit with negligible loss in model precision accuracy.

Addendum

The model has been enhanced from the previous phase covering more classes. It has been trained with eight more classes, making the problem a 13-class classification task.

Dataset Distribution

Table 2 shows detailed information about number of objects annotated of all sub-categories from 26,600 images. All images in the dataset were annotated with all sub-categories by drawing bounding boxes around each anomaly found and labeled accordingly (Figure 10).

<table>
<thead>
<tr>
<th>Name of Anomaly</th>
<th>Symbol</th>
<th>Number of Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crack Longitudinal</td>
<td>CL</td>
<td>15407</td>
</tr>
<tr>
<td>Crack Circumferential</td>
<td>CC</td>
<td>4179</td>
</tr>
<tr>
<td>Crack Hinge</td>
<td>CH</td>
<td>994</td>
</tr>
<tr>
<td>Crack Spiral</td>
<td>CS</td>
<td>353</td>
</tr>
<tr>
<td>Crack Multiple</td>
<td>CM</td>
<td>4039</td>
</tr>
<tr>
<td>Deposit Attached Encrustation</td>
<td>DAE</td>
<td>1899</td>
</tr>
<tr>
<td>Deposit Attached Grease</td>
<td>DAGS</td>
<td>1899</td>
</tr>
<tr>
<td>Deposit Attached Others</td>
<td>DAZ</td>
<td>1934</td>
</tr>
<tr>
<td>Deposit Settled Others</td>
<td>DSZ</td>
<td>1225</td>
</tr>
<tr>
<td>Root Ball</td>
<td>RB</td>
<td>564</td>
</tr>
<tr>
<td>Root Medium</td>
<td>RM</td>
<td>1415</td>
</tr>
<tr>
<td>Root Fine</td>
<td>RF</td>
<td>1802</td>
</tr>
<tr>
<td>Collapse</td>
<td>X</td>
<td>361</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>36071</strong></td>
</tr>
</tbody>
</table>

Table 2. Annotations of Anomalies in Dataset

![Figure 9. Performance comparison](image)

Figure 9. Performance metrics for OpenVINO™ optimized vs Non-optimized inference on Intel processors

![Sample Anomalies and Annotated Images](image)

Figure 10. Sample Anomalies and Annotated Images
Model Training
To train the model, we randomly partitioned the dataset into 80% for training, and 20% for evaluation, and then generated TF records for it (see Generating TF Records section). We used TensorFlow’s truncated_normal_initializer to initialize the weights. The initial learning rate was set to the default 0.000300000014249 and the momentum optimizer was used to train. Training continued until 300,000 steps were reached using our image dataset of 26,600 images. Various training parameters such as loss, accuracy, mAP across validation set, are to be monitored for determining the status of the training. This is very important to avoid overfitting of the model.

Model Evaluation
We evaluated the model using the hold-out validation data set with respect to several performance metrics. Model detection recall is calculated based on the formula shown in the Validating the Model section.

The model achieved a Recall score of 80.11% on the test dataset.

Model Inference
The trained model is converted to a TensorFlow frozen graph. It is then optimized using OpenVINO (see Inference Optimization section) for inference, and the inference results were captured on the latest Intel® architectures (Figure 11).

Conclusion
The objective of this work was to enable CNN-based deep learning inferencing on edge devices with Intel CPUs, for pipeline condition assessment. From the chart in Figure 11, it is clear that we can perform the automated pipe condition assessment from pipeline inspection videos in various Intel processor-based platforms with acceptable performance.

Based on this experiment, optimizing the deep learning model using OpenVINO for Intel architecture-specific hardware results in real-time performance for this use case.

![Performance comparison](image)

*Figure 11. Performance metrics for OpenVINO™ optimized vs Non-optimized inference on Intel processors*
Authors

**Kuljeet Singh** has worked extensively in the IoT, mobility and embedded systems domains for 17 years. Presently a Solutions Architect (IoT & AI) with Wipro’s Industrial and Engineering Services division, he is working on the development and deployment of AI/ML based solutions for IoT domain. He is also working with the technology teams of a leading semiconductor customer on the next generation of IoT solutions. For more information, contact him at kuljeet.singh@wipro.com.

**Sundar Varadarajan** is an industry expert in the field of analytics, machine learning and AI, having ideated, architected and implemented innovative AI solutions across various industry verticals. Currently, Sundar is a Consulting Partner at Wipro on AI/ML, and plays an advisory role on various leading edge AI and Machine Learning solutions. Sundar can be reached at sundar.varadarajan@wipro.com.

**Subin Guruvayurappan** is a Software Engineer at Wipro with 3.5 years of experience in various domains like machine learning, deep learning, computer vision, DevOps and MEAN stack development. He is also interested in AI/ML and IoT. For more information, contact him at subin.guruvayurappan@wipro.com.

**Anubhav Anand** is a Software Engineer at Wipro who is involved in the development of applications in Predictive Analytics, Computer Vision and NLP. He is interested in building the AI-powered SaaS products. For more information, contact him at anubhav.anand1@wipro.com.
Pipe Sleuth is a member of the Intel® AI Builders Program, an ecosystem of industry-leading independent software vendors (ISVs), system integrators (SIs), original equipment manufacturers (OEMs), and enterprise end users, which have a shared mission to accelerate the adoption of artificial intelligence across Intel® platforms.

4. https://www.tensorflow.org/deploy/distributed
6. https://www.nassco.org/content/pipeline-assessment-pacp

Performance results are based on testing as of May 2019 and may not reflect all publicly available security updates. No product can be absolutely secure. See configuration disclosure for details.

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on non-Intel microprocessors. Developers may obtain the available optimization by using the compilation options that may be available for their optimization software.

Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations, and functions. Any changes to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of the product when combined with other products. For more complete information visit: http://www.intel.com/performance.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of the product when combined with other products. For more complete information visit, http://www.intel.com/benchmarks.

Performance results are based on testing as of May 2019 and may not reflect all publicly available security updates. See configuration disclosure for details.

Intel technologies’ features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at https://www.intel.com.