WHITE PAPER

Artificial Intelligence
Deep Learning
Intel® AI Builders

Power System Infrastructure Monitoring Using Deep Learning on Intel® Architecture

Abstract

The work in this paper evaluates the performance of Intel® Xeon® processor powered machines for running deep learning on the GoogleNet* topology (Inception* v3). The functional problem tackled is the identification of power system components such as pylons, conductors, and insulators from the real-world video footage captured by unmanned aerial vehicles (UAVs) or commercially available drones. By conducting multiple experiments we tried to derive the optimal batch size, iteration count, and learning rate for the model to converge.

Introduction

Recent advances in computer-aided visual object recognition, namely the application of deep learning, has made it possible to solve a wide array of real-world problems which previously were impossible. In this work, we present a novel method for detecting the components of power system infrastructure such as pylons, conductor cables, and insulators.

The original implementation of this algorithm took advantage of the power of the NVIDIA* graphics processing unit (GPU) during training and detection. The current work primarily focuses on implementing the algorithm on TensorFlow* CPU mode and executing it over Intel® Xeon® processors.

During execution, we will record performance metrics across the different CPU configurations.Facestibus, idelit rest, sunt.

Environment Setup

Hardware Setup

Intel Xeon processor	Model Name: Intel® Xeon® processor E5-2699 v4 @ 2.20GHz	
	Core(s) Per Socket: 22	RAM (free): 123 GB
	OS: Ubuntu* 16.1	

Table 1. Intel® Xeon® processor configuration.

Software Setup

1. Python* Setup

The experiment is tested on Python* version 2.7.x. Verify the version as follows:

```
$ python --version
```

Table of Contents

2. TensorFlow* Setup

   1. Install TensorFlow using pip: "$ pip install tensorflow". By default, this would install the latest wheel for your CPU architecture. Our experiment is built and tested on TensorFlow3 version 1.0.x.
   2. Verify the installation using the command shown below:

```
$ python -c "import tensorflow; print(tensorflow.__version__)"
```

3. Inception* Model

The experiments detailed in the subsequent sections employ the transfer learning technique to speed up the entire process. For this purpose, we used a pretrained GoogleNet* model, namely Inception* v3. The details of the transfer learning process are explained in the subsequent sections.

Download the Inception v3 model from the following link: http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz

4. TensorBoard*

We use TensorBoard* in our experiments to visualize the progress and the results of individual experiment runs.

TensorBoard is installed along with TensorFlow. After installing TensorFlow, enter the following command from the bash script to ensure that TensorBoard is available:

" `$ tensorboard --help` "

Solution Design

The entire solution is divided into three stages. They are:

- Data Preprocessing
- Model Training
- Inference

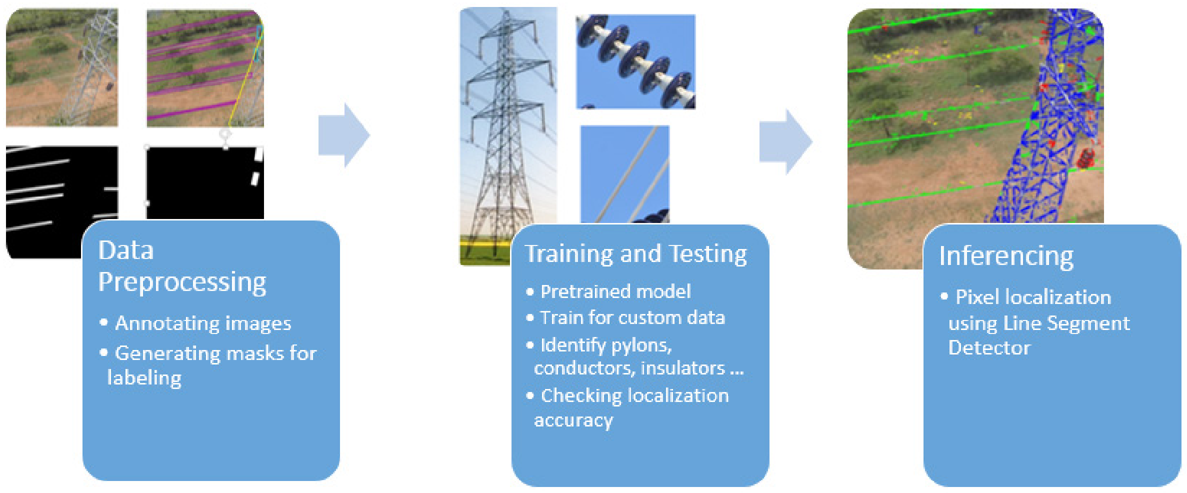
Data Preprocessing
• Annotating images
• Generating masks for labeling
Training and Testing
• Pretrained model
• Train for custom data
• Identify pylons, conductors, insulators ...
• Checking localization accuracy
Inferencing
• Pixel localization using Line Segment Detector

Figure 1. High-level solution design

Data Preprocessing

The images used for training the model are collected through aerial drone missions carried out in the field. The images collected vary in resolution, aspect, and orientation, with respect to the object of interest.

The entire preprocessing pipeline is built using OpenCV* 2 (Python implementation). The high-level objective of preprocessing is to convert the raw, high-resolution drone images into a labeled set of image patches of size 32 x 32, which is used for training the deep learning model.

The various processes involved in the preprocessing pipeline are as follows:si iducillaut eicat enes molorep udicate nos sitaspe rferae quatet planderis quate nihicietus, corporae pratectur si comnimus.

- Image annotation
- Generating binary masks
- Creating labeled image patches

The individual processes involved in the pipeline are detailed in the following steps:

Step 1: Image annotation.

Those experienced in the art of building and training convolutional neural network (CNN) architectures will quickly relate to the image annotation task. It involves manually labeling the objects within your training image set. In our experiment, we relied on the Python tool, LabelImg*4, for annotation. The tool outputs the object coordinates in XML format for further processing.

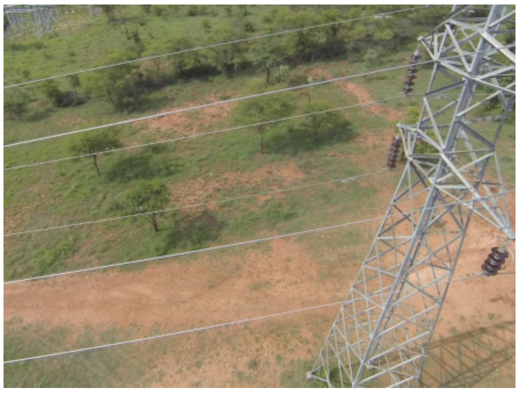

Figure 2. Image without annotation.

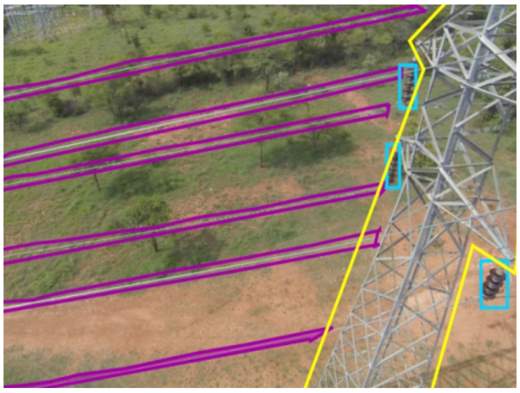

Figure 3. Image with annotation overlay.

The preceding images depict a typical annotation activity carried out on the raw images.

Step 2: Generating binary masks.

Binary masks refer to the mode of image representation where we depict either the presence or absence of an object. Hence, for every raw image, we generate individual binary masks corresponding to each of the labels available. The binary masks so created are used in the steps that follow for actually labeling the image patches. This idea is depicted in the following images. In the current implementation, the mask generation process is developed using Python OpenCV.

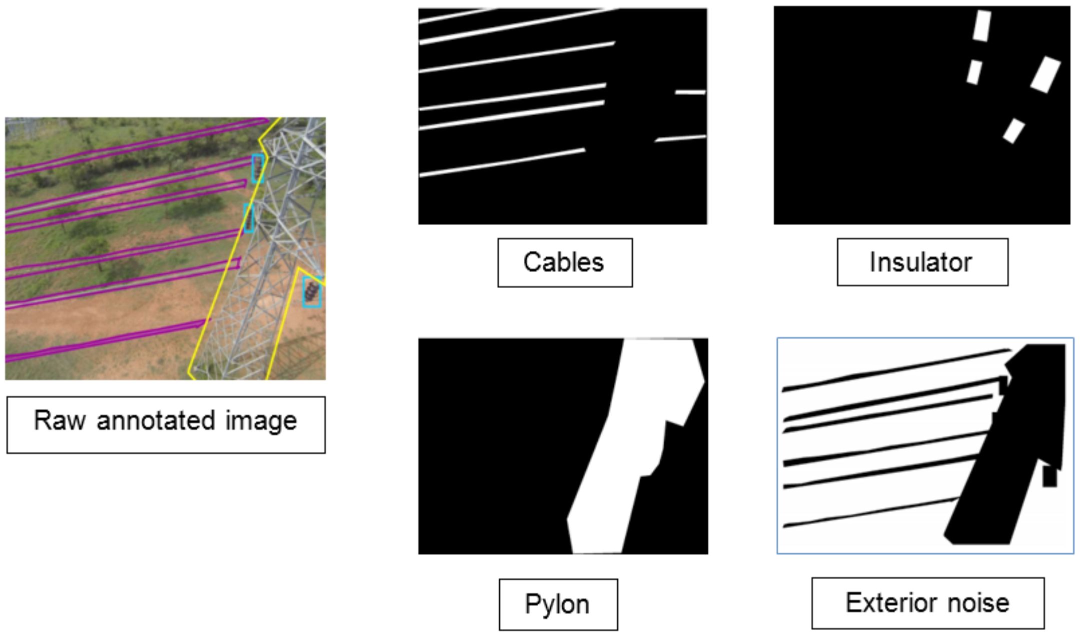

Figure 4. Generating binary masks from the raw image.

Step 3: Creating labeled image patches.

Once the binary mask is generated, we run a 32 x 32 filter over the raw image and compare the activations (white pixel count) obtained in the various masks for the corresponding filter position.

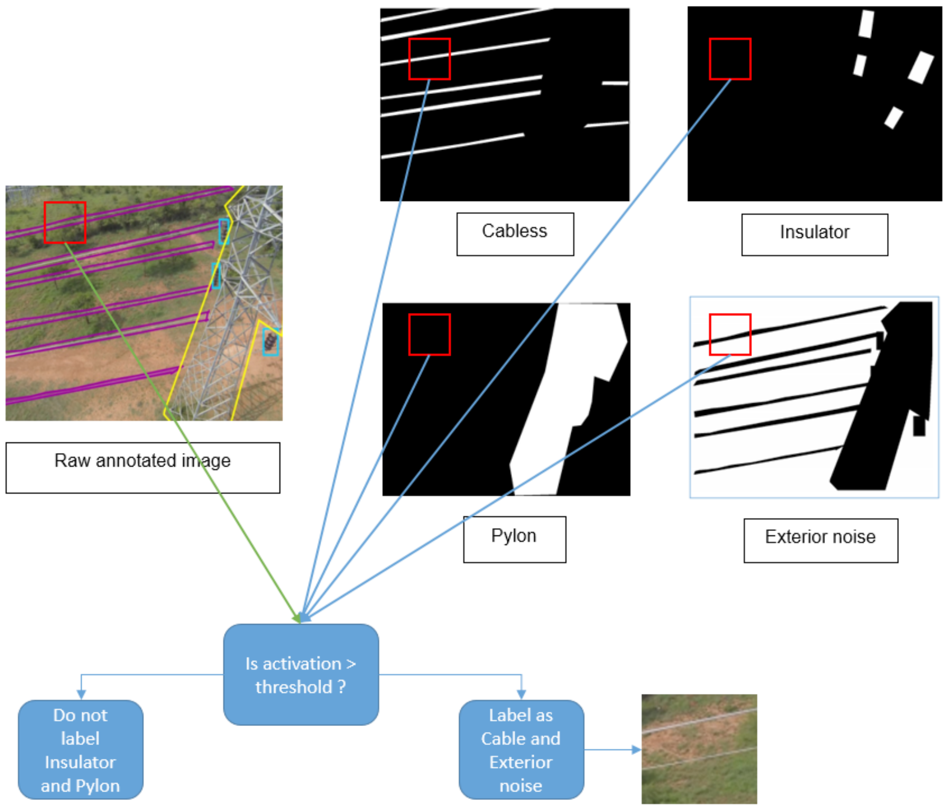
Cabless
Insulator
Raw annotated image
Pylon
Exterior noise
Is activation > threshold ?
Do not label Insulator and Pylon
Label as Cable and Exterior noise

Figure 5. Creating labeled image patches.

If the activation in a particular mask is found to be above the defined threshold of 5 percent of patch area (0.05*32*32), we label the patch to match the mask's label. The output of this activity is a set of 32 x 32 image patches partitioned into multiple directories based on their labels. The forthcoming model training phase of the experiment directly accesses this partitioned directory structure for label-specific training images.

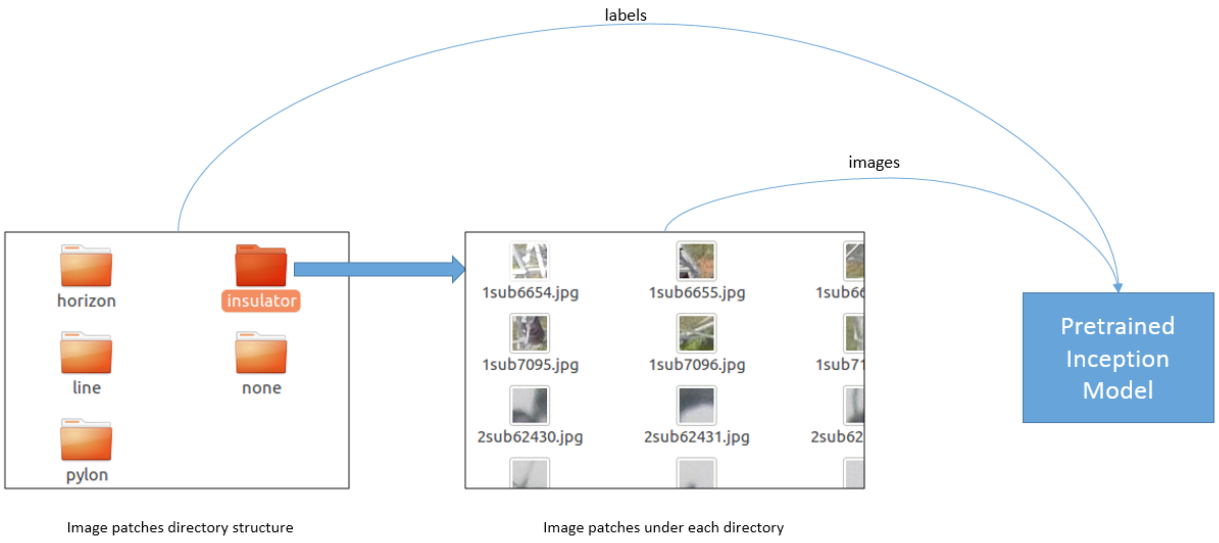

Figure 6. Preprocessing output directory structure.

Please note that in the above-described patch generation process, the total number of patches generated varies, depending on other variables such as size of the filter (32 x 32 in this case), resolution of input images, and the activation threshold, while comparing with binary masks." `$ tensorboard --help` "

Network Topology and Model Training

Inception v3 Model

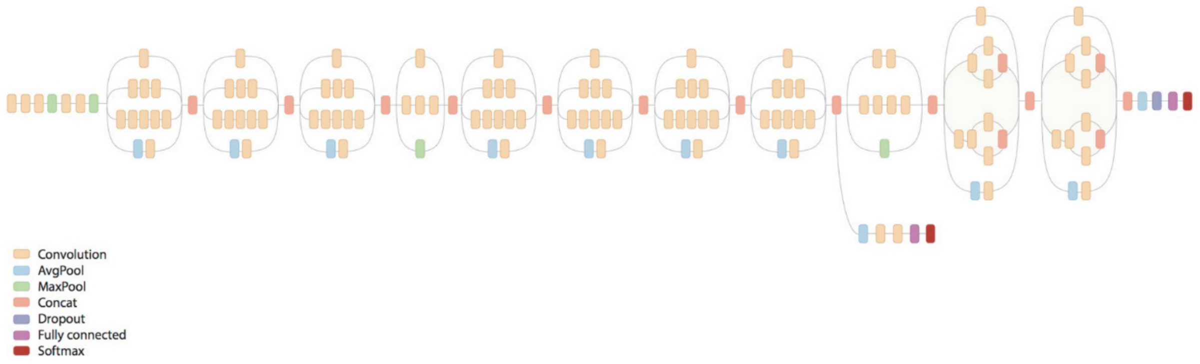

Figure 7. Inception V3 topology.

Inception V3 is a revolutionary deep learning architecture, which achieved state of the art performance in ILSVRC14 (ImageNet* Large Scale Visual Recognition Challenge 2014).

The most striking advantage of Inception over the other topologies is the depth of feature learning achieved, keeping the memory and CPU cost nearly at a par with other topologies. The architecture tries to improve on performance by reducing the effective sparsity of the data structures by converting them into dense matrices through clustering. This sparse-to-dense conversion is achieved architecturally by designing telescopic convolutions (1 x 1 to 3 x 3 to 5 x 5). This is commonly referred to as the network-in-network.

Transfer Learning on Inception

In our experiments we applied transfer learning on a pretrained Inception model (trained on ImageNet data). The transfer learning approach initializes the last fully connected layer with random weights (or zeroes), and when the system is trained for the new data (in our case, the power system infrastructure images), these weights are readjusted. The base concept of transfer learning is that the initial many layers in the topology will have learned some of the base features such as edges and curves, and this learning can be reused for the new problem with the new data. However, the final, fully connected layers would be fine-tuned for the very specific labels that it is trained for. Hence, this needs to be retrained on the new data.

This is achieved through the Python API, as follows:

1. Add new hidden layer, Rectified Linear Unit (ReLU):

```
hidden_units_layer_1 = 1024
  layer_weights_fc1 = tf.Variable(
      tf.truncated_normal([BOTTLENECK_TENSOR_SIZE, hidden_units_layer_1], stddev=0.001),
      name='fc1_weights')
  layer_biases_fc1 = tf.Variable(tf.zeros([hidden_units_layer_1]), name='fc1_biases')
  hidden_layer_1 = tf.nn.relu(tf.matmul(bottleneck_input, layer_weights_fc1,name='fc1_matmul') + layer_biases_fc1)
```

2. Add new softmax function:

```
layer_weights_fc2 = tf.Variable(
      tf.truncated_normal([hidden_units_layer_1, class_count], stddev=0.001),
      name='final_weights')
  layer_biases_fc2 = tf.Variable(tf.zeros([class_count]), name='final_biases')

  logits = tf.matmul(hidden_layer_1, layer_weights_fc2,
                     name='final_matmul') + layer_biases_fc2
  final_tensor = tf.nn.softmax(logits, name=final_tensor_name)
```

Testing and Inference

Testing is done on a 90:10 split on the entire image set. The test images go through the same patch generation process that was invoked during the training phase. The resultant patches are sent for detection on the trained model.

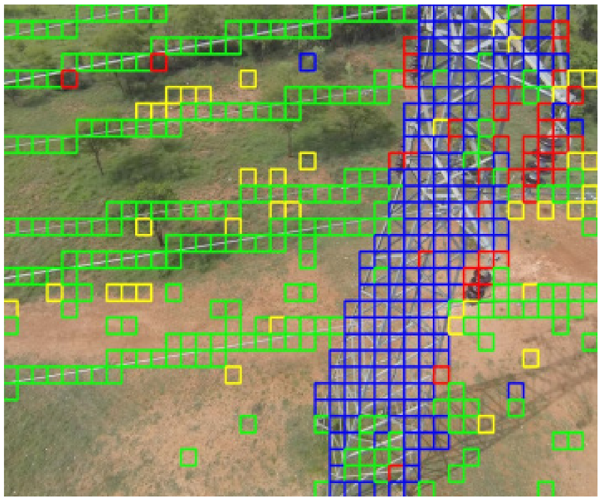

Figure 8. Result of model inference overlaid on raw image.

After detection, the patches are passed through a line segment detector (LSD) for the final localization.

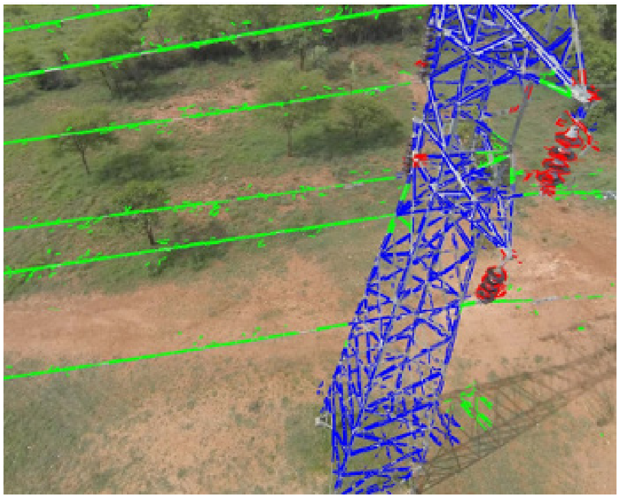

Figure 9. Result of running LSD.

Results

The different iterations of the experiments involve varying batch sizes and iteration counts.

During the experiments, in order to reduce the time consumed during preprocessing, we modified the preprocessing logic. Therefore, the metrics for different variants of the preprocessing logic were also captured.

We also observed that in the inception model, bottleneck tensors are cached during the initial run, so the training time during the subsequent runs would be much less. The final training result for the Intel Xeon processor is as follows.

Sl. No	Experiment ID	Batch Size	Iteration Count	Preprocess Time	Train Time in Seconds	Inference Time (Images Per Second)	Accuracy (Top-1 Test)	Cross-Entropy	Train Images /Sec
1	run1batch100step40k	100	40K	1100	53797	1.80	0.72	0.61	13.9
2	run2batch100step40k Modified pre-processing logic	100	40K	29876	59576	1.79	0.74	0.63	10.54
3	run1batch256step40k cached bottleneck	256	40K	NA	8190	1.90	0.76	0.74	65.34
4	run1batch512step40k cached bottleneck	512	40K	NA	14052	1.84	0.70	0.66	38
5	run1batch1024step40k cached bottleneck	1024	40K	NA	27571	1.84	0.73	0.72	19.4
6	run1batch2048step40k cached bottleneck	2048	40K	NA	55056	1.86	0.71	0.65	9.72

Table 2. Experiment results.

Note: Inference time is inclusive of the preprocessing (patch) operation along with the time for the actual detection on the trained model.

Running the Experiment on Intel® Xeon Phi™ Processors

The next set of experiments are designed for processors falling into Intel® Many Integrated Core Architecture, in this case the Intel Xeon Phi processors. The objective of the experiment is to utilize all the available cores within the processor to speed up the experiment execution time.

The processor specifics are as follows:

Intel® Xeon Phi™ Processor	Model Name: Intel Xeon Phi processor 7210 @ 1.30GHz
	Core(s) Per Socket: 64 CPU:256 RAM (free): 109 GB
	Operating System: CentOS* 7.2

To obtain the proposed performance benefit we need to perform two high-level tasks, as follows:

1. Install TensorFlow that is optimized for the modern Intel® architecture.
   The latest Intel® processors are equipped with registers specialized for vector operations. The instruction set for accessing these registers falls into Intel® Advanced Vector Extensions (Intel® AVX). Specifically, the Intel Xeon Phi processor comes with 512-bit floating point operation registers and corresponding instruction set extension as Intel® Advanced Vector Extensions 512 (Intel® AVX-512). Ideally, our TensorFlow build should utilize these registers through the compiler flag '–mavx512', which would enable the required instruction sets during compilation. But since the Google* TensorFlow repository is still not prepared for Intel AVX-512 compilation, in our experiment we relied on the Intel® Math Kernel Library 2017 for influencing the power of Intel AVX-512.

2. Make necessary changes to the environment and TensorFlow code.
   Environment-related setups:

   a. OMP_NUM_THREADS: This specifies the maximum number of OpenMP* threads that should be spawned. Ideally, this should be kept slightly less than the number of physical CPU cores multiplied by 2, where 2 is the level of multithreading that could be achieved on each core.

   b. KMP_BLOCKTIME: The wait time between thread completion and sleeping.

c. KMP_SETTINGS: When enabled, it prints the runtime library environment variables during session execution.

d. KMP_AFFINITY: Instructs the execution engine to bind threads to same physical cores.

TensorFlow session modifications:

e. intra_op_parallelism_threads: The session configuration parameter that controls the maximum parallelism that could be obtained within a TensorFlow operation. Ideally, this would be the same as the OMP_NUM_THREADS value defined above.

f. inter_op_parallelism_threads : This configuration controls parallelism across TensorFlow operations. Setting this value equal to the number of sockets is recommended.

The performance parameters defined above could be set up by referring to TensorFlow* Optimizations on Modern Intel® Architecture. The parameter values depend on the deep learning topology being experimented, the underlying hardware architecture, etc.

The settings for our Intel Xeon Phi processors are as follows:

```
export OMP_NUM_THREADS='127'
export KMP_BLOCKTIME='30'
export KMP_SETTINGS='1'
export KMP_AFFINITY='granularity=fine,verbose,compact,1,0'
```

Changes made to the TensorFlow script are as follows:

```
..
...
def create_inception_graph():

  config = tf.ConfigProto(inter_op_parallelism_threads=2,intra_op_parallelism_threads=136)
  with tf.Session(config=config) as sess:
    model_filename = os.path.join(
        FLAGS.model_dir, 'classify_image_graph_def.pb')
    with gfile.FastGFile(model_filename, 'rb') as f:
      graph_def = tf.GraphDef()
      ...
      ..
```

Experiment Setup:

All the experiment runs listed below were run for an iteration count of 40K. The primary objective of the experiment was to monitor the training time and train throughput (measured in images per second). Across all the experiments we had observed the test accuracy to be between 0.72 ~ 0.74.

Sl. No	Experiment ID	Preprocess Time		Train Time in Seconds		Train Images/Sec	
		Intel® Xeon® Processor	Intel® Xeon Phi™ Processor	Intel Xeon Processor	Intel Xeon Phi Processor	Intel Xeon Processor	Intel Xeon Phi Processor
1	run2batch100step40k Modified preprocessing logic	X	8.5X	----	-- did not finish --	-----	-- NA --
2	run1batch256step40k Cached bottleneck	NA	NA	5.11X	X	4.17X	X
3	run1batch512step40k Cached bottleneck	NA	NA	4.92X	X	4.19X	X
4	run1batch256patchsize256	NA	NA	4.34X	X	5.24X	X

Table 3. Experiment results for Intel® Xeon® processors versus Intel® Xeon Phi™ processors.

Observations

The experiment pipeline had two major stages, namely data preprocessing and model training. By utilizing the many cores within the Intel Xeon Phi processor, we observed an 8.5X improvement in data preprocessing time.

The model training was found to be slower on Intel Xeon Phi processors, which was attributed to the slower input/output performance and frequent input/output blocks during the bottleneck generation phase in the inception model.

Conclusion and Future Work

The functional use case tackled in this paper involved the detection and localization of power system components. The use case could be further expanded to identifying powersystem components that are damaged.

The training and inference time observed could be further improved by using an Intel optimized version of TensorFlow5.

List of Abbreviations

Abbreviations	Expanded Form
DL	deep learning
LSD	line segment detector
UAV	unmanned aerial vehicle
GPU	graphics processing unit

References and Links

The references and links used to create this paper are as follows:

1. Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna, Rethinking the Inception Architecture for Computer Vision (2015).

2. Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi, Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning (2016).

3. Tensorflow Repository, https://github.com/tensorflow/tensorflow.

4. LabelImg – Python tool for image annotation, https://github.com/tzutalin/labelImg.

5. Optimized Tensorflow – TensorFlow optimizations on Modern Intel Hardware, https://software.intel.com/en-us/articles/tensorflow-optimizations-on-modern-intel-architecture