WHITE PAPER

Transportation
Artificial Intelligence
Intel AI® Builders

# Pedestrian Detection Using TensorFlow* on Intel® Architecture

### Table of Contents

## Abstract

This paper explains the process to train and infer the pedestrian detection problem using the TensorFlow* deep learning framework on Intel® architecture. A transfer learning approach was used by taking the frozen weights from a Single Shot MultiBox Detector model with Inception* v2 topology trained on the Microsoft Common Objects in Context* (COCO) dataset, and then using those weights on a Caltech pedestrian dataset to train and validate. The trained model was used for inference on traffic videos to detect pedestrians. The experiments were run on Intel® Xeon® Gold processor-powered systems. Improved model detection performance was observed by creating a new dataset from the Caltech images, and then selectively filtering based on the ratio of image size to object size and training the model on this new dataset.

## Introduction

With the world becoming more vulnerable to pronounced security threats, intelligent video surveillance systems are becoming increasingly significant. Video monitoring in public areas is now common; prominent examples of its use include the provision of security in urban centers and the monitoring of transportation systems. These systems can monitor and detect many elements, such as pedestrians, in a given interval of time. Detecting a pedestrian is an essential and significant task in any intelligent video surveillance system, as it provides fundamental information for semantic understanding of the video footages. This information has an obvious extension to automotive applications due to its potential for improving safety systems.

Continued research in the deep learning space has resulted in the evolution of many frameworks to solve the complex problem of image classification, detection, and segmentation. These frameworks have been optimized specific to the hardware on which they are run in order to achieve better accuracy, reduced loss, and increased speed. Intel has optimized the TensorFlow* library for better performance on its Intel® Xeon® Scalable processors.

This paper discusses the training and inferencing pedestrian detection problem that was built using the Inception* v2 topology with the TensorFlow framework on an Intel® processor-powered cluster. A transfer learning approach was used by taking the weights for the Inception v2 topology on the Microsoft Common Objects in Context* (COCO) dataset and using those weights on a Caltech dataset to train and validate. Inference was done using traffic videos to detect the pedestrians.

## Train and Infer Procedures

This section describes in detail the steps we used to train and infer the pedestrian detection problem.

### Choosing the Environment

#### Hardware Configuration

Intel has launched its new and faster Intel® Xeon® Gold processor powered system and the experiments are performed on this system.

The details of the hardware for the Intel Xeon Gold processor powered system used for the experiments are listed in the following Table 1:

| | |
|---|---|
| Architecture | x86_64 |
| CPU op-mode(s) | 32 bit, 64 bit |
| Byte order | Little endian |
| CPU(s) | 24 |
| Core(s) per socket | 6 |
| Socket(s) | 2 |
| CPU family | 6 |
| Model | 85 |
| Model name | Intel® Xeon® Gold 6128 processor @ 3.40 GHz |
| RAM | 92 GB |

**Table 1.** Intel® Xeon® Scalable Gold processor configuration.

#### Software Configuration

The TensorFlow framework optimized for Intel® architecture and the Intel® Distribution for Python* were used as the software configuration, as shown in Table 2.

| | |
|---|---|
| TensorFlow* | 1.3.0 (Intel® optimized) |
| Python* | 3.5.3 (Intel distributed) |

**Table 2**. Software configuration for the Intel® Xeon® Gold processor.

The software configurations are available on the hardware environments chosen, and no source build for TensorFlow*AI  was necessary.

#### TensorFlow Object Detection API

The TensorFlow Object Detection API was used, which an open source framework is built on top of TensorFlow that makes it easy to construct, train, and deploy object detection models. This API was used for the experiments on the pedestrian detection problem.

### Dataset

We chose the Caltech Pedestrian Dataset[1] for training and validation. This dataset consisted of approximately 10 hours of 640x480 30-Hz video that was taken from a vehicle driving through regular traffic in an urban environment. To accommodate multiple scenarios, about 250,000 frames (in approximately 137 one-minute-long segments) with a total of 350,000 bounding boxes and 2,300 unique pedestrians were annotated.

The dataset consisted of the following elements:

- A list of bounding boxes for the image. Each bounding box contained:
    - Bounding box coordinates (with the origin in the upper-left corner) defined by four floating point numbers such as, [ymin, xmin, ymax, xmax]. We stored the normalized coordinates (x / width, y / height) in the TFRecord dataset.
    - The class of the object in the bounding box.
- The dataset was organized into six training sets and five test sets.
- Each set consisted of 6–13 one-minute-long .seq files with annotations in .vbb file format.
- An RGB image was encoded for the dataset as jpeg.

## Topology

The Inception architecture was built with the intent of improving the use of computing resources inside a deep neural network. The main idea behind Inception is the ability to approximate a sparse structure with spatially repeated dense components and use dimension reduction like those used in a “network in network” architecture to keep the computational complexity in bounds, but only when required. The computational cost of Inception is also much lower than that of other topologies. More information on Inception is given in this paper[2]. Figure 1 shows the Inception architecture.

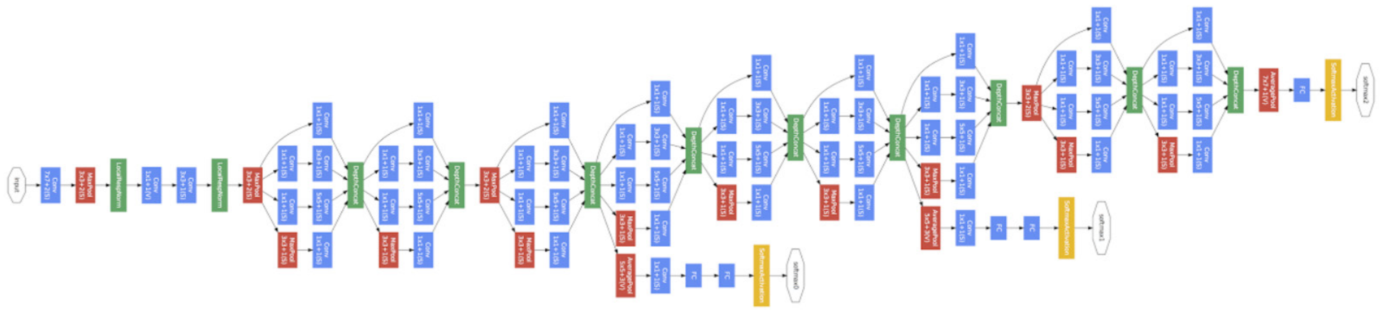

**Figure 1.** GoogLeNet* Inception* model.[3]

Inception v2 has a slight structural change in the Inception module. Figure 2 shows the Inception v2 module structure.

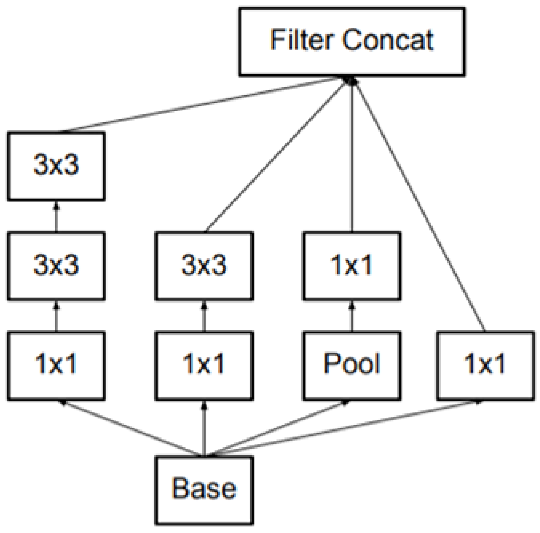

**Figure 2.** Inception* v2 module.[3]

To accelerate the training process, we applied a transfer learning technique by using the pretrained Inception v2 model from GoogLeNet* on the COCO dataset. The pretrained model had already learned the knowledge on the data and stored that in the form of weights. These weights were directly used as initial weights and readjusted when the model was retrained on the Caltech dataset.

The pretrained model (265MB) was downloaded from the following link : http://download.tensorflow.org/models/object_detection/ssd_inception_v2_coco_2017_11_17.tar.gz

## Methodology

This section covers the steps we followed to train and infer pedestrian detection on Intel architecture.

These steps included:

- Preparing the input
- Training the model
- Experimental runs and inference

### Preparing the Input

#### TFRecord Format

To use the pedestrian dataset in TensorFlow Object Detection API, it must be converted into the TFRecord file format. Reading data from the TFRecord file is much faster in TensorFlow than reading from other image formats.

The Caltech dataset consisted of images in the jpg format and their corresponding annotations in XML format.

To convert the dataset into TFRecord format, we did the following:

1. Images from the .seq files were extracted into an Images folder.
2. The annotations from the corresponding .vbb files were extracted into an annotations folder.

The following code was used to convert the Caltech dataset into TFRecord format:

```
DATASET_DIR=./CALTECH/train/
OUTPUT_DIR=./tfrecords
python tf_convert_data.py \
    --dataset_name=caltech \
    --dataset_dir=${DATASET_DIR} \
    --output_name=caltech_tfrecord \
    --output_dir=${OUTPUT_DIR}
```

#### Label Map

Each dataset is required to have an associated label map. This label map defines a mapping from string class names to integer class Ids. The label created for pedestrian was as follows.

```
item {
 id: 1
 name: 'person'
}
```

#### Configuring Training Pipeline

The TensorFlow Object Detection API uses protobuf files to configure the training and evaluation process. The configuration file is structured into five sections. The required sections were used as appropriate. The changes to be done in each section are as below.

1. **model** section, set the num_classes to one (`num_classes: 1`). For the pedestrian detection only one class has to be detected.
2. **train_config** section, set the checkpoint file with the path. (fine_tune_checkpoint: " `~/research/object_detection/models/model/ssd_inception_v2_coco_2017_11_08/model.ckpt`")
3. **train_input_reader** section, set the input_path (input_path: "`~/caltech/cal_tfrecord/caltech_train_000.tfrecord`") and label_map_path (label_map_path: "`~/research/object_detection/data/ped_label_map.pbtxt`"). The paths given are examples. Paths can be set as per the location of the files on individual systems.
4. **eval_config** section, number of samples to be evaluated.
5. **eval_input_reader** section and also the label_map_path is set the same as train_input_reader. The input_path is set to point to the evaluation dataset (input_path: "`~/caltech/cal_tfrecord/caltech_train_001.tfrecord`").

### Training the Model

After making the necessary changes as listed in the previous section, experimental runs were done to retrain the model on the Intel Xeon Scalable Gold processor. Different parameter values for environment options and finally it was found that the following combinations works the best.

```
"OMP_NUM_THREADS = "8" or "6"
"KMP_BLOCKTIME" = "0"
"KMP_SETTINGS" = "1"
"KMP_AFFINITY"= "granularity=fine, verbose, compact, 1, 0"
'inter_op' = 1
'intra_op' = 8 or 6
```

Values of both 6 and 8 gave a per-step execution time that varied between 2 and 4 seconds.

## Experimental Runs and Inference

### On the Intel Xeon Scalable Gold processor (AI DevCloud Cluster)

To run the training on the AI DevCloud, we used the following command to submit the training job:

```
qsub ped_train.sh -l walltime=24:00:00
```

On this cluster, there is a restriction on walltime for six hours to execute a job. There is a maximum value that can be set to 24 hours. As shown in the qsub command, the walltime is set to 24 hours.

The job script ped_train.sh has the following code:

```
#PBS -l nodes=1:skl
cd $PBS_O_WORKDIR
protoc object_detection/protos/*.proto --python_out=.
export PYTHONPATH=$PYTHONPATH:`pwd`:`pwd`/slim
numactl --interleave=all python ~/research/object_detection/train.py --logtostderr \
--pipeline_config_path=~/research/object_detection/models/model/ssd_inception_v2_caltech.config
--train_dir=~/research/object_detection/models/model/ckpt_train
```

Table 3 lists the details of the run iterations.

| Run | Iteration Count | Batch Size | Loss |
|---|---|---|---|
| 1 | 3600 | 24 | 4.3007 |
| 2 | 8509 | 64 | 2.5600 |
| 3 | 10441 | 24 | 2.7100 |
| 4 | 15501 | 24 | 3.6926 |
| 5 | 21304 | 24 | 1.9600 |
| 6 | 28406 | 24 | 2.4090 |
| 7 | 35940 | 24 | 2.2240 |

**Table 3.** Run iteration details.

### On Intel Xeon Scalable Gold processor dedicated cluster

To run on the dedicated cluster, the walltime setting is not required. The other part of the code as listed under the DevCloud cluster section above remains the same.

The training was done for 190K+ iterations and the variation of loss is shown in Figure 3.

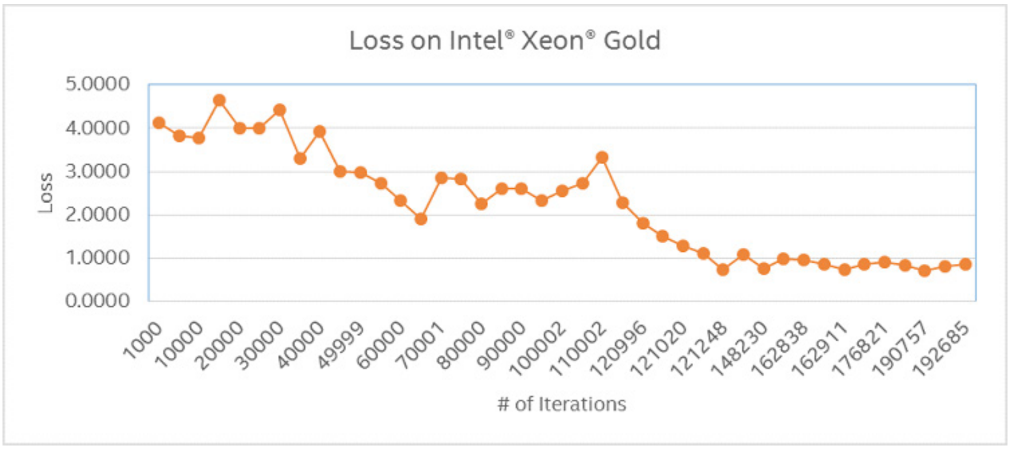

**Figure 3.** Variation of loss on an Intel® Xeon® Scalable Gold processor.

After the model was trained, we exported it to a TensorFlow graph proto. The checkpoint will typically consist of three files:

- `model.ckpt-${CHECKPOINT_NUMBER}.data-00000-of-00001`
- `model.ckpt-${CHECKPOINT_NUMBER}.index`
- `model.ckpt-${CHECKPOINT_NUMBER}.meta`

After identifying a candidate checkpoint, we used the following script to export the trained model file for inference:

```
#PBS -l nodes=1:skl
cd $PBS_O_WORKDIR
protoc object_detection/protos/*.proto --python_out=.
export PYTHONPATH=$PYTHONPATH:`pwd`:`pwd`/slim
python object_detection/export_inference_graph.py  --input_type=image_tensor  --
pipeline_config_path=~/research/object_detection/models/model/ssd_inception_v2_caltech.config  --
trained_checkpoint_prefix=~/research/object_detection/models/model/ckpt_train/model.ckpt-34150  -
-output_directory=~/research/object_detection/models/model/output_inference_graph
```

Figure 4 shows the inference output for the model.

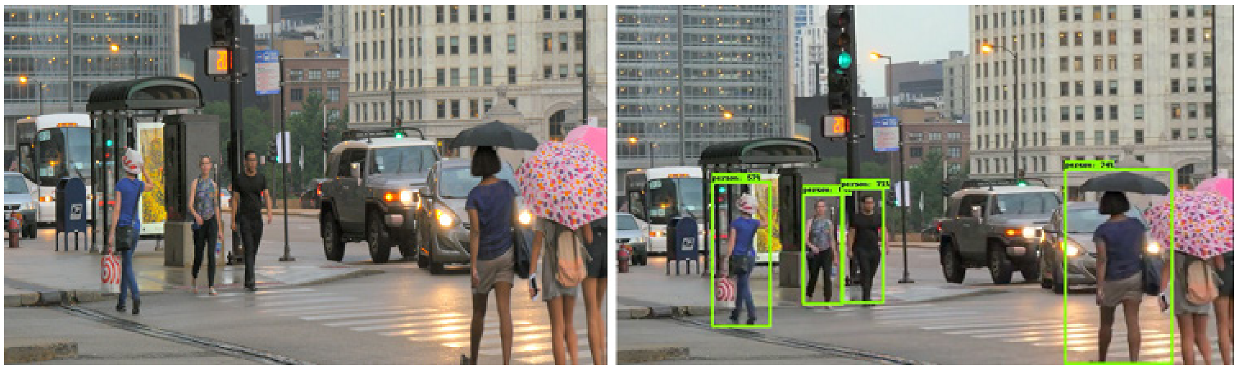

**Figure 4.** Raw[6] and inferenced frames on the Intel® Xeon® Gold processor.

## Results and Improvement

The inference runs on the Intel Xeon Scalable Gold processor resulted in a Mean Average Precision (mAP) close to ~30 percent. To boost the accuracy we looked at other options to treat the training data.

To achieve better detection performance, the size of the image and objects within the image need to be tracked and adjusted.

The Caltech dataset consists of a dominant set of images where the pedestrian objects are ~50 to ~70 in pixel size, which is less than 15 percent of the image height. The presence of too many small-scale objects in the images could potentially result in underperformance on pedestrian detection by the model when trained on this dataset. Treating the dataset could help improve the detection performance of the model.

## Data Treatment, Training, and Inference

The following steps were performed on the Caltech data to selectively choose the right data:

1. Filter those images where the size of the objects in any image is less than 5 percent of the size of the image. This forms a new dataset.
2. From the newly created dataset in step 1, filter those images where the size of the objects in any image is less than 10 percent of the image size.
3. From the set created in step 2, filter those images where the size of the objects in any image is less than 15 percent of the image size.
4. Remove the dataset created in step 2 from the one created in step 1.

All the datasets were converted into TFRecord format for training and inference.

The dataset created in step 1 was used for training, while the ones in steps 2 and 3 were used for testing.

Table 4 summarizes the counts of the datasets created.

| Caltech database | 5% Object Size Filtering (A) | 10% Object Size Filtering (B) | 15% Object Size Filtering (C) | Training (A-B) | Inference1 (B) | Inference2 (C) |
|---|---|---|---|---|---|---|
| 60,000 | 6,279 | 1,270 | 270 | 5,000 | 1,270 | 270 |

**Table 4.** New treated dataset details.

The model was run for 33K iterations using the new training dataset of 5,000 images. Table 5 details the training performed.

| Iteration count | Batch Size | Loss |
|---|---|---|
| 33,103 | 24 | 1.3527 |

**Table 5.** New run iteration details.

Inference was run on 1,270 and 270 count datasets. Table 6 shows the results of the inference.

| Inference # | Image Count | mAP |
|---|---|---|
| 1 | 1,270 | 46% |
| 2 | 270 | 73% |

**Table 6.** Inference results.

Figure 5 shows the inference output for the model.

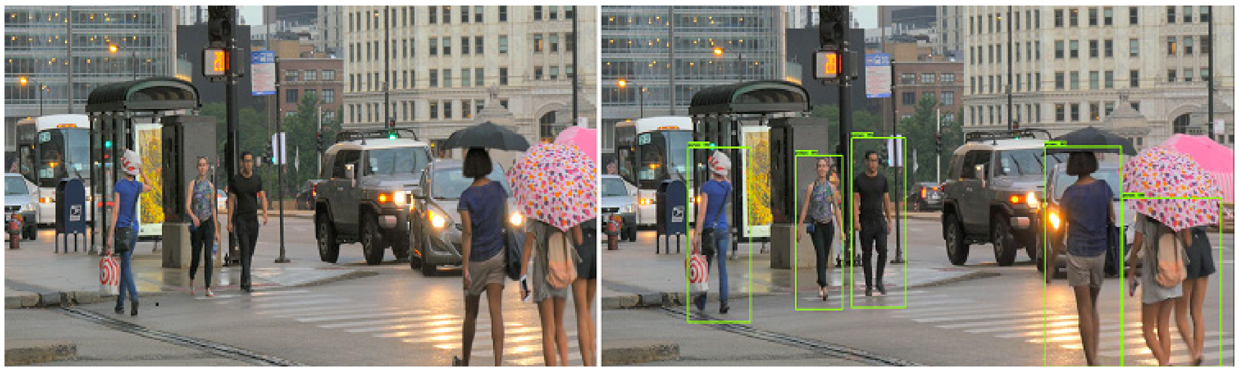

**Figure 5.** Raw[6] and inferenced frames trained on a treated dataset on the Intel® Xeon® Gold processor.

Comparing the results, the model detection was better on the treated dataset.

## Summary

In this paper, we discussed training and inferencing a pedestrian detection problem built using the Inception v2 topology with the TensorFlow framework on Intel architecture applying the transfer learning technique. The weights from the model trained on the COCO dataset were used as initial weights on the Inception v2 topology. These weights were readjusted when the model was retrained using the Caltech dataset on the Intel Xeon Scalable Gold processor powered environment. The model was better trained as the iterations increased on both systems. The mAP was observed to be low. From the Caltech dataset, by selectively filtering the images, where the pedestrian object sizes were less than 5 percent of the image size and training the model on this new dataset, improved the mAP. As a next step, more generalization of the model can be achieved by creating custom pedestrian datasets with varied object sizes and training on those datasets to improve the model detection performance.

## About the Author(s)

Ajit Kumar Pookalangara, Rajeswari Ponnuru, and Ravi Keron Nidamarty are part of the Intel and Tata Consultancy Services relationship team, working to evangelize artificial intelligence.

## References

1. Caltech dataset for training:

http://www.vision.caltech.edu/Image_Datasets/CaltechPedestrians/

2. Going deeper with convolutions:

https://arxiv.org/pdf/1409.4842v1.pdf

3. Rethinking the Inception Architecture for Computer Vision:

https://arxiv.org/pdf/1512.00567v3.pdf

4. TensorFlow Object Detection API:

https://github.com/tensorflow/models/tree/master/research/object_detection

5. Single Shot MultiBox Detector in Tensorflow:

https://github.com/balancap/SSD-Tensorflow

6. Traffic Video:

https://www.videvo.net/video/traffic-in-downtown-chicago/5069/voluptatia doluptatquae num etur simus eariossimpor moluptatur?Summary

## Related Resources

SSD: Single Shot MultiBox Detector: https://arxiv.org/abs/1512.02325

TensorFlow* Optimizations on Modern Intel® Architecture: https://software.intel.com/en-us/articles/tensorflow-optimizations-on-modern-intel-architecture

Build and Install TensorFlow* on Intel® Architecture: https://software.intel.com/en-us/articles/build-and-install-tensorflow-on-intel-architecture

TensorFlow Issue #1907: https://github.com/tensorflow/models/issues/1907