Image Localization and Detection

Using Python, Inception, Tensorflow, and Tensorboard

Anthony Hershberger

1 ImageNet

1.1 ImageNet Database

ImageNet is an open source large scale image database containing 14 million hand-annotated images that indicate what objects or classes are in the pictures. A subset of 1.2 million images have been hand-annotated to provide ground truth bounding boxes. The synsets, used to create the class labels, are determined using a data structure of noun labels provided by WordNet, a large lexical database of the English language.

Using the WordNet noun synset, nearly 21,000 synsets can be drawn upon to class label objects, such as "natural object" or "sports", that are also connected by children synsets(nodes) or subclasses. For example, a typical synset would start a tree with the class label "natural object" which would have another child synset of "rock" leading to another child synset "boulder".

1.2 ImageNet Statistics

WordNet Statistics of High Level Categories			
High Level Category	Num. of Synsets	Avg. Num. of images in synset	Total Images
Animal	3822	732	2799K
Device	2385	675	1610K
Person	2035	468	952K
Plant	1666	600	999K

Table 1: Statistics of 4 of the largest categories in ImageNet

1.3 ImageNet Competition

The goal of the competition is to produce an algorithm that will recognize single and/or multiple instances of an object in a given image and draw bounding boxes around that object with a high degree of accuracy. The competition will require the algorithm achieve the following tasks with a high degree of accuracy:

- *Image Classification*: For each of the 1000 object classes the algorithm must predict the absence or presence of at least one of the class objects in the test image. The output is a decreasing rank of five object class confidence scores.

- *Object Detection*: For each of the 1000 object classes the algorithm must predict bounding boxes of each object class for any test image. The bounding box should be output with confidence scores for the detection so that a precision/recall curve can be constructed.

2 Preparing The Data

2.1 Overview

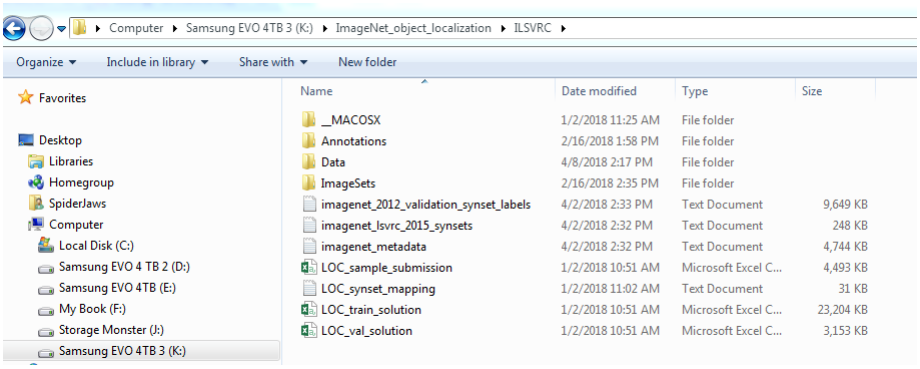
The ImageNet Localization Challenge data set consists of a training and validation set that contains 150,000 annotated images along with 1000 class labels. The image annotations are saved in an XML format and must be parsed using the PASCAL Development Kit or through a custom parser. Annotations are given and ordered by synsets. The following notation is used:

- *wnid*: This ID represents the annotation to the image's synset label. For example, n00015388 would be the synset label for animals. n01905661 would be the label for invertebrate, which is a child synset of animals.

- *PredictionString*: The prediction string is a 5 integer, space delimited string of a class prediction with bounding box coordinates. For example, a string [899 30 50 190 110] would denote a prediction of class label 899 with the bounding box coordinates of $x_{min} = 30, y_{min} = 50, x_{max} = 190, and\ y_{max} = 110$.

2.2 Download the Data

In order to obtain the ImageNet dataset, you must sign up at Imagenet.org to obtain an $ACCESSKEY$ that grants access to the data. Once you download and unzip the files, you should see a directory containing the following files and subfolders.



2.3 Data Files

The following files will be used to create your data set:

- **Annotations Folder** The XML files contain synset id, the integer width and height, and bounding box coordinates in pixels. These files within the 1000 synsets will be used to produce the *process bounding box* csv file used to create the TFRecords.The annotations folder is given in the file structure `<data-dir>/nxxxxxxxx/nxxxxxxxx_YYYY.xml`.
- **Data Folder** This contains the train, validation, and test images.

2.3.1 TXT and CSV files

File	Description
`imagenet_2012_validation_synset_labels.txt`	validation synsets
`imagenet_lsvrc_2015_synsets.txt`	training synsets
`imagenet_metadata.txt`	human-readable descriptions
`process_bounding_box.csv`	Unpacks XML bounding boxes into single csv

2.4 Process Image Validation Data

In order to create the dataset, we must first use the `preprocess_imagenet_validation_data.py` script. The purpose is to remap the data structure of the training data so that we can reuse the `process_bounding_box.csv` on both the training and validation data sets. Using the following steps with Tensorflow-gpu == 1.5 and Python 3.5.5 in the Anaconda Prompt and Tensorflow activated environment, the `preprocess_imagenet_validation_data.py` script can be executed.

1. Open up Anaconda Prompt
2. Change your command line directory to where `preprocess_imagenet_validation_data.py` script resides.

3. Enter `python preprocess_imagenet_validation_data.py`

   `imagenet_2012_validation_synset_labels.txt <data-dir>` into the command line. Your data directory should be the directory where your validation images reside in disk.

4. Upon completion of the preprocessed validation data, your validation images should have the same file structure as your training data which is `<data-dir>/nxxxxxxx/ILSVRC2012_val_YYYY.jpg`

2.5 Processing Bounding Boxes

After processing your validation data to have the same file structure as your training set, you can now process the XML files that contain two points that specify the lower left and upperH-right corners of the rectangular bounding box. The bounding box coordinates are given in integer pixel values that have a relative relation to the the image size. These values must be converted to floating point integers on a [0,1] scale to store the information in a TFRecord format. The following steps can be followed to process the XML bounding box files.

1. Open up Anaconda Prompt
2. Change your command line directory to point where the `process_bounding_boxes.py` file resides.
3. Call the process bounding box script by entering `python process_bounding_boxes.py <data-dir> [training-synsets] > process_bounding_box.csv`
4. The `> process_bounding_box.csv` will redirect the STDOUT of the python script to a single CSV containing 5 columns: `<JPEG filename>`, `<xmin>, <ymin>, <xmax>, <ymax>.`
5. Repeat using the validation synset labels.

Note: The `process_bounding_boxes.csv` file will be used as a data input for building the TFRecord database in `build_imagenet_data.py` and to provide bounding box information in the pre-processing of the neural network when random color and image augmentation will be applied.

2.6 Building the Imagenet Database

2.6.1 Data Parsing

In order to deal with the large and complex data that will stress the limits of computational bandwidth and speed, we will implement at our data parsing stage protocol buffers, the TFRecords file format, and use TensorFlow commands to make our model training as efficient as possible.

- **Protocol Buffers** are a proprietary Google technology to serialize large structured data sets. They were designed to automate and efficiently read and write files into a program by defining the data structure *only once* and then generating source code to execute functions in Python or TensorFlow. For larger datasets, protocol buffers handle TFRecord formats faster than XML and CSV by reducing the byte sizes of command line code and reading records that are streamlined for data parsing by CPU's or GPU's. Protocol buffers can be identified by their .proto file extensions.

- **TFRecords** are a standard TensorFlow file format that allows users to take advantage of the protocol buffers. TFRecords consists of a set of sharded files where each record is serialized into a $tf.Example$ proto containing a JPEG encoded image and the corresponding label and bounding box metadata. By converting your data into the TFRecords file format, your code will become easier to read, adding data from other datasets will be easier, and pasrsing tools in Tensorflow can be utilized for faster data parsing and image decoding.

- **TensorFlow Commands**: Python was the chosen language to express and control the training of models using TensorFlow commands. The choice of Python to implement TensorFlow commands was due to Python's strong roots and efficient C++ backend, as well as, its popularity among machine learning experts and data scientists. Python allows for the creation of extension modules, namely Numpy, that can be written in C++ and interfaced with native Python code. The goal is to have Python give you an easy to use environment to program while using Tensorflow to pipeline computationally expensive tasks, such as matrix multiplication, to modules written in C++ or CUDA languages in order to take advantage of coding efficiencies or GPU paralellism.

2.6.2 Building the ImageNet Database

After processing the training and validation bounding box XML files into a single CSV file called `process_bounding_box.csv` , the images, their appropriate synset text files, and bounding box information can be converted into a TFRecord. The script `build_imagenet_data.py` produces $N = \frac{Number of Shards}{Number of Threads}$ shards per thread where the `num_shards` and `num_threads` variables are user defined inputs tailored to the number of threads the user's computer can handle. A processor capable of handling 12 threads with a user defined `num_shards`=132 would produce 11 shards or TFRecord partitions.

The shards will contain individual TFRecords which are serialized example protos. The `tf.train.Example{features=tf.train.Features(dict{inputs}}` function creates each individual record with the following features:

- **Height/Width** - 64 bit signed integer, pixel height and pixel width.

- **Colorspace** - utf-8 encoded string for 'RGB'

- **Bounding Box** - 64 bit floating point digits for `<xmin> <ymin> <xmax> <ymax>`
- **Synset** - utf-8 encoded string for the synset
- **Text** - utf-8 encoded string for human-readable label
- **Filename** - utf-8 encoded string for `<JPEG file path>`
- **Image Format** - utf-8 encoded string for `<image format>`

2.6.3 Processing the TFRecords

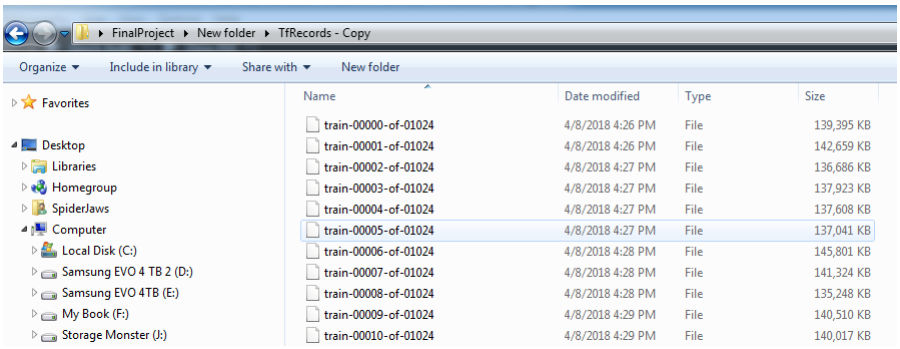
The following files must be used as inputs into the `build_imagenet_data.py` script: `imagenet_lsvrc_2015_synsets.txt` , `imagenet_metadata.txt` , `process_bounding_box.csv`. The 2015 ImageNet synsets text file is used to map each synset to a numerical category in alphabetical order beginning at 1. The zero label is reserved as a background category that is used when no class label can be assigned to an image. The ImageNet metadata text file is used to map human-readable class labels to the TFRecords. The synset label n01930112 would translate to nematode, nematode worm, or roundworm.

While the TFRecords are being created, the `build_imagenet_data.py` script will also convert all the files that are not JPEG with RGB colorspace to the correct format. The most common conversions are PNG images to JPEG images and CMYK to RGB colorspace. Moreover, for each TFRecord created, the image order will be shuffled to ensure that the images are randomly inserted with respect to their class labels.

After the correct files have been input into the script and user-defined inputs set, the `build_imagenet_data.py` can be executed using the following steps:

1. Open up the script in your Python editor
2. Enter in the `num_shards`, `num_threads`, the imagenet metadata text file, and `process_bounding_box.csv` file and save it. You may choose to run this script in your Python interpreter.
3. If not, open Anaconda Prompt
4. Change your command line directory to the directory with your `build_imagenet_data.py` script
5. Execute the script by entering in `python build_imagenet_data.py`
6. Depending on your computer hardware, this could take hours or even a day to run.

When your script has finished your data directory should look like the following screenshot.



2.7 Computational Complexity of Raw Files

Since the data is scraped from the web, the raw files of the dataset reside in a variety of image file formats. The image resolutions vary by color gamut, image size, files type and quality. The large size, variations in image resolution, miss annotated labels, and complex data structuring of the labels present many challenges. The sheer size of the dataset will require the implementation of parallel processing with graphic processing units to handle the I/O complexity, the large memory management of large batches of images, and the numerous matrix calculations. GPU's are better suited to handle these tasks due to the sheer number of CUDA cores, wide memory width, and fast GB/s throughput.

2.8 Image Pre-processing

2.8.1 Image Processing Parameters

Image pre-processing before training your convolutional neural network is critical for training accuracy and robustness. The input data parameters must be carefully considered when designing a network architecture. The following input parameters must be considered in your network design:

- Number of images in a batch
- Image width and height (image size)
- Number of image color channels
- Pixel intensity range (i.e 0 - 255)

2.8.2 Processing Steps

Image processing is done on each individual image, in parallel, and then the processed images are then grouped together to create a training batch. The following steps can be used as a framework to processing images for batch inputting into a convolutional neural network.

1. After parsing the data into TFRecords, bring an individual image from the disk.
2. Check for RGB color channel, image dimensions, and JPEG file format.
3. Decode the JPEG image file into a tensor.
4. Apply cropping, distortions, and padding (Image Augmentation).
5. Apply random color augmentations (Color Augmentation).
6. Collect images in a batch until the batch size is met.

2.8.3 RGB, Image Dimensions, and JPEG

Since the ImageNet data is web-scraped from a wide range of sources, it is important to check the number of color channels in an image. There are 13 images that have a subtractive CMYK, or 4 color channel feature. Since our convolutional neural network will work on a fixed square size matrix computation, these 13 images must be converted to RGB or discarded. Failure to do so will cause input errors when trying to feed forward their incompatible tensor sizes.

It is also important to check the file type of the image to ensure that it follows the JPEG extension. The ImageNet data contains a number of PNG files that must be converted to JPEG's or discarded. PNG's appear to have the RGB color space but include an extra alpha channel that gives the image a feature to control transparency. This extra transparency channel will cause an input error when feeding the file forward through the network.

Since our convolutional neural network will be programmed to accept fixed sizes of width and height, it is very challenging when working with image dimensions of various sizes. The ImageNet data contains images that are as large as 1280x800 and as small as 300x300. More importantly, there are a variety of aspect ratios associated with the various image dimensions which is challenging when our input aspect ratio must be a square. Data augmentation helps us to deal with dimension size while attempting to improve the robustness of our network.

2.8.4 Image Augmentation

Image augmentation occurs when we create new images from modifications on existing images. We can augment images by mirroring, rotating, flipping, or zooming in or out on an image. The benefit to applying image augmentation is that it can add noise to your model and prevent over-fitting by adding modified or distorted images to your data set.

For example: If an entire class of elephants were all right-facing, a convolutional neural network may perform poorly when faced with a left-facing elephant and fail to classify it properly. In order to improve the image classification of

left-facing elephants, we can simply mirror the right-facing elephants and create another image to represent a left-facing elephant.

Image augmentation can also help us to resize our images to a square aspect ratio by implementing three techniques: random crop, center square crop, and distorted bounding box method. When using random crop, we can produce a number of new images with a square aspect ratio by cropping out a new image from the various corners of the image. The center square crop method will pick a center point of the image and crop a square aspect ratio that leaves a user-defined percentage of the image remaining. The last method, using a distorted bounding box, will choose one of the ground truth bounding boxes and distort it based on an acceptable range of distortion. With the newly distorted bounding box, the image will be cropped according to its dimensions leaving a distorted image that correctly fits the bounding box and has a square aspect ratio.

2.8.5 Color Augmentation

One of the most powerful aspects of image augmentation involves randomly applying color distortions to the image. Unlike a wedding photographer who would like to create a pristine and beautifully colored image for every picture of a wedding party, the goal of color augmentation is to provide a convolutional neural network with a variety of suboptimal colorings that will reduce overfitting and provide invariance to aspects of an image in order to boost its robustness on classification prediction.

Color augmentation is a non-commutative and destructive process meaning that the ordering of the color operations matter and those operations can not be undone unless the process is started over. Given unlimited computational power, the ideal application of color augmentation would be to randomly permute over all color operations. Since random permutations of color operations is computationally too expensive, the better method would be to choose three sets of coloring operations and randomly choose between the three.

In our convolutional neural network, we will use the following image plugins to apply color augmentation:

- **Unsharp Mask**: A technique for increasing the acutance of an image making it appear sharper by increasing contrast among the edges in an image.
- **Saturation**: An image effect that uniformly increases/decreases the intensity of a color based on the percieved light. A color that is fully saturated produces its pure color tone. A color that has zero saturation is on a grayscale.
- **Hue**: The hue is simply the gradation of color in your image.
- **Brightness**: The brightness is the relative pixel intensity of visible light. In the RBG color space, the pixel intensities scale from 0 to 255. If we increase the brightness by 10 percent, every pixel intensity would increase uniformly by 10 percent.

- **Contrast**: Contrast is the range of tonal difference. In the RBG color space, contrast specifically defines how sharply colors differ from one another.

Example of Pre-Processed Image

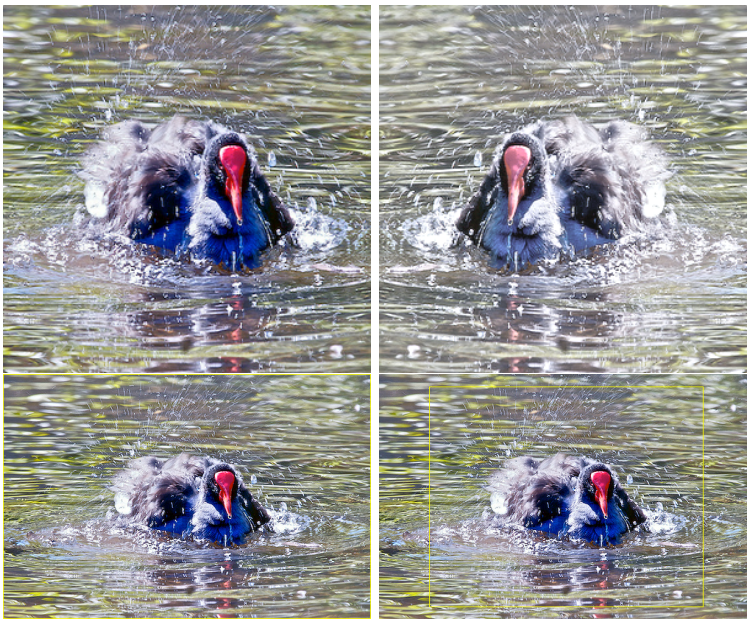


Figure 1: **Top Left:** Cropped and Resized **Top Right:** Flipped with random adjustments to brightness and hue **Bottom Left:** Stretched image with redrawn bounding box around edges **Bottom Right:** Resized image with resized bounding box

2.8.6 Pre-Processing Code

Although the `inception_preprocessing.py` script is included in the folder, it is not run directly in the terminal. However, the script will be accessed and executed when running the `train_image_classfier.py` script that will be discussed in the next section. The `inception_preprocessing.py` script file is important for a variety of reasons. For one, when `train_image_classfier.py` is executed in the terminal, it will construct a computational graph that is formed by implementing the pre-processing steps in the `inception_preprocessing.py` script and by constructing the layers of the neural network as designed in the `inception_v3.py` script. The preprocessing task will produce the following name scopes within the **Distort Image** name scope for the creation of the computational graph:

- **Convert Image:** During the training and evaluation phase, this name scope will be used to call on the function *preprocess image* to make the appropriate image format and rescaling conversions.
- **Distort Bounding Boxes:** This name scope inputs a properly converted image, applies random cropping and flipping techniques, and draws a newly distorted bounding box based. The newly formed images will then be output for additional color processing.
- **Distort Color:** This name scope inputs a randomly selected image, randomly chooses 1 of 4 predefined color augmentation orderings, and outputs a tensor of the image for batching.

3 Convolutional Neural Network Implementation

3.1 Setting up your Network

Before training your model, there are a number of important considerations to make before beginning to train your convolutional neural network. It cannot be understated how computationally efficient your decisions must be when programming your code, choosing your image dimensions, thinking about your color depth(RGB, grayscale, 4-channel?), and understanding your hardware limitations. Given the most efficient training algorithms, a simple doubling of your image dimensions can add hours, days, or even weeks to your training, so understanding how your parameters stress your computer is very important.

When implementing the network, the image queue that governs the batch of images will be run entirely on the CPU and the CPU random access memory. This means that all resizing, image and color augmentation, and batching will be calculated using the CPU and CPU memory. As a batch of images is built, the GPU sends dequeue operations to the CPU to instruct it to release a batch of images. If the batch size is set too high, the GPU memory will be exceeded causing your system to stop training. The optimal batch size is the size that utilizes the most GPU bandwidth without halting the process of training. By running your pre-queue operations on the CPU, the GPU is free to perform gradient descent, compute intensive matrix calculations, and output your realtime training performance measures.

3.2 Training your network

In order to train your convolutional neural network, you will execute the `train_image_classifier.py` script. The training script calls on 4 modules that must be contained in the same folder or defined with an absolute path to the modules directory. The 4 modules include *model deploy, preprocessing factory, dataset factory, and nets factory.* After verifying the modules are properly placed in the folder or defined by an absolute path, there are a number of flags,

or user defined parameters, that need to be considered before executing your script. The following flags include but are not limited to:

- **Batch Size**: If the batch size flag is not set, the default will be a batch size of 32. When choosing a custom batch size, it is generally good practice to choose sizes with a base of 2 (i.e 32 ,64, 128, 256).
- **Train Directory**: The directory where the event logs and checkpoints are written.
- **Dataset Directory**: The directory where the TFRecords for the training and validation data are stored.
- **Dataset Split Name** For the training data, we will set this flag to "train". For the evaluation phase, we will set this to "validation".
- **Checkpoint Path** If we want to train our neural network from a previously saved checkpoint, we will set this directory to point to a saved checkpoint. If there is no path defined, this flag will default to the training folders latest checkpoint. If no checkpoints are found, the model will start from scratch.
- **Checkpoint Exclude Scopes** This flag instructs the training phase to not implement certain aspects of your model training. In order to exclude a variable or name scope, we must use this flag by including a comma seperated list of variable and name scopes to exclude. When restoring parameters from a checkpoint it is often the case that you may want to exclude the last layer and retrain it with more or less classes.
- **Trainable Scopes** This flag instructs the CNN to train on only the variables defined by a comma separated list. If this flag is not used, the default is to train the CNN on all variables.
- **Number of Clones** This flag instructs how many instances of the convolutional neural network that you would like to run when running in parallel. If using multiple GPU's you would set this flag to the number of GPU's that you have in your model. For a CPU-only model, this number would default to 1.
- **Clone-on-CPU** This flag allows you to run the model on the CPU version of Tensorflow. If you have the GPU version of Tensorflow, you could use clone-on-cpu to create the instance of the model on the CPU and perform only matrix computation on your GPU's. This option gives better performance when run-time errors prevent completion of model training and evaluation.

In order to run the network we can use the following steps:

3.3 Steps to Execute the Training Script

1. Open up Anaconda Prompt

2. Set the directory to point to where the `train_image_classifer.py` script resides. Make sure the modules for *model deploy, preprocessing factory, dataset factory, and nets factory* are appropriately defined.

3. Execute the script by typing in `python train_image_classifier.py` and entering in the appropriate flags for `--train_dir --dataset_dir --clone_on_cpu`. These flags are the only essential flags to train the model from scratch using pre-defined hyperparameters.

4. If we would like to save time and not train the model from scratch, we can use the `--checkpoint_path` flag to define a saved check point. Google provides checkpoints to train the last layer of all their Inception models.

5. If you would like to use user-defined hyper-parameters and optimization methods you can use a variety of flags that are annotated in the Inception source code.

6. After executing the script, you should begin to see the model train by initializing the global step parameter and outputting the global steps per second. As the model trains, the model will periodically save checkpoints to mitigate downtime from interruptions caused by vanishing gradients, data corruption, or software/hardware hiccups. Depending on your hardware setup this could take weeks or months to train. It is recommended to begin from a predefined checkpoint to ensure model training can be completed.

3.4 Executing the Training Script

In order to train the Imagenet data set, we can use the following code:

```
python train_image_classifier.py

--train_dir {training directory}
--dataset_dir {dataset directory}
    --checkpoint_path {checkpoint path}
    --checkpoint_exclude_scopes=InceptionV3/Logits,InceptionV3/AuxLogits
    --trainable_scopes=InceptionV3/Logits,InceptionV3/AuxLogits
    --clone_on_cpu=True
    --optimizer=rmsprop
    --rmsprop_momentum=.9
    --rmsprop_decay=.9
    --learning_rate=.01
    --ending_learning_rate=.0001
```

If we have executed the `train_image_classifier.py` script appropriately, then the model should begin training and our terminal should produce the following output:

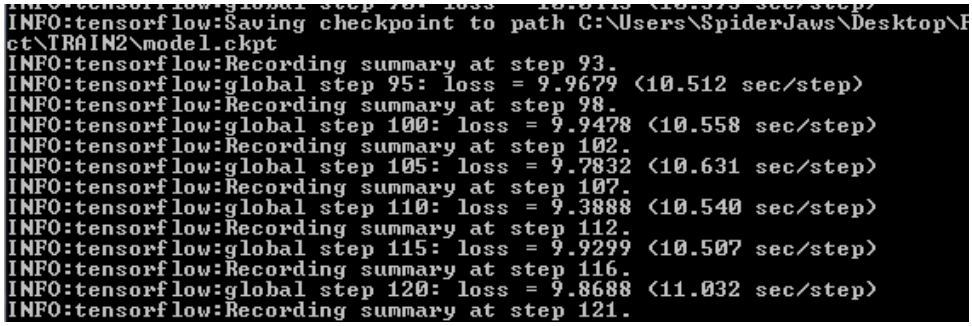


Figure 2: Training ImageNet.

3.5 Files Produced By The Training Script

File	Description
`events.out.tfevents.{Serial #}.{Computer Name}-PC`	Event Files for Tensorboard
`checkpoint.ckpt`	Saved checkpoint
`graph.pbtxt`	Computational Graph Definition
`model.ckpt-{step #}.data-###-of-####`	Data of all training images already captured at latest step

3.5.1 Tensorflow Event Files

The Tensorboard event files correspond to the summary operations, such as `tf.matmul` or `tf.nn.relu`, that are just tensors that contain serialized protobuffers. These protobuffers allow summary operations, such as `tf.summary.scalar` or `tf.summary.histogram`, to collect the summary data. In order to write the summary data to disk, `tf.summary.FileWriter` is used to direct the data to a specified directory using the `--logdir` flag. Tensorboard uses the event files to display the following visualization dashboards:

- **tf.summary.scalar**: Scalar statistics are network metrics that vary over time or by step. The scalar dashboard will contain metrics for the network's learning rate, model loss, and global steps per second.
- **tf.summary.image**: The image dashboard contains the image at each step and their various bounding box distortions and color augmentations.

- **tf.summary.histogram** The histogram dashboard displays information regarding how the distribution of tensors vary over time or by step.

Running Tensorboard In Browser

After your network has finished or during network training, you may view all your summary operations by using the following steps:

1. Open up your Anaconda Prompt and activate the Tensorflow environment.
2. Initialize Tensorboard by executing the following command `tensorboard --logdir={Training Directory}`
3. The command line will return `Tensorboard 1.5.1 http://localhost:6006`
4. Open up your browser and enter local host address: `http://localhost:6006`

If we have successfully accessed the Tensorflow event files, the browser should display the dashboard below:

Tensorboard In Browser

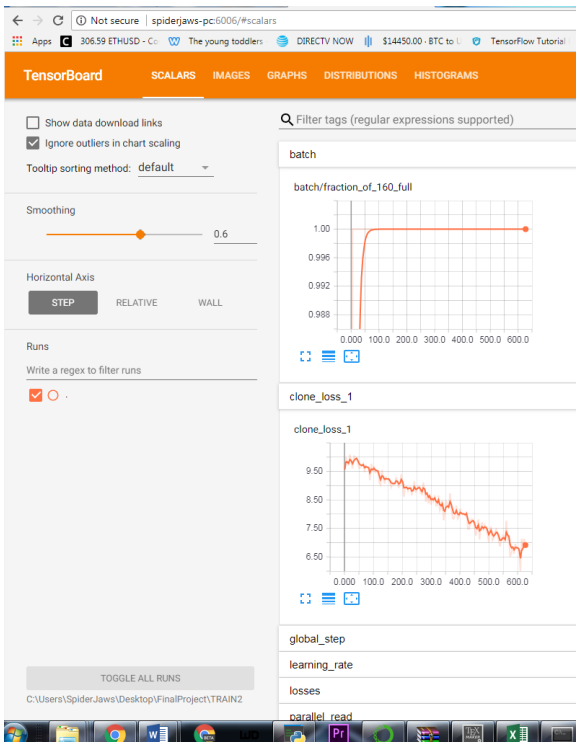


Figure 3: Tensorflow output constructed from TF Events

3.6 Defining the Computational Graph with Inception V3

In order to train the convolutional neural network, the `train_image_classifier.py` script will call the `inception_v3.py` module that defines the computational graph to Google's Inception V3 deep convolutional neural network. The structure of the the network is written to disk as a `graph.pbtxt` file and called upon at the execution of the `train_image_classifier.py` script.

The structure of Inception V3 begins with its 22 parameterized layers, 5 pooling layers, and its inception modules that bring the total layer count to 100. The purpose of the inception modules was to define a number of convolutional operations, such as 1x1 3x3 or 5x5 filter sizes, and let the module stack all the possible convolutions. Using the stack of all possible convolutions, the model can now recover the local features associated with smaller convolutions and the higher abstractions associated with the larger convolutions.

One of the most groundbreaking concepts of the inception modules is the clever way that it will reduce the dimensionality of the entire network. When a previous layer passes its outputs to an inception module, the module will first calculate the 1x1 convolution filter. This 1x1 filter is then placed in front of the larger convolutional filters. The effect is subtle but the simple placing of the 1x1 filter in front of the larger convolutions allows the network to retain a lot of flexibility in recovering feature information while reducing the total number of operations in the network.

Inception Module

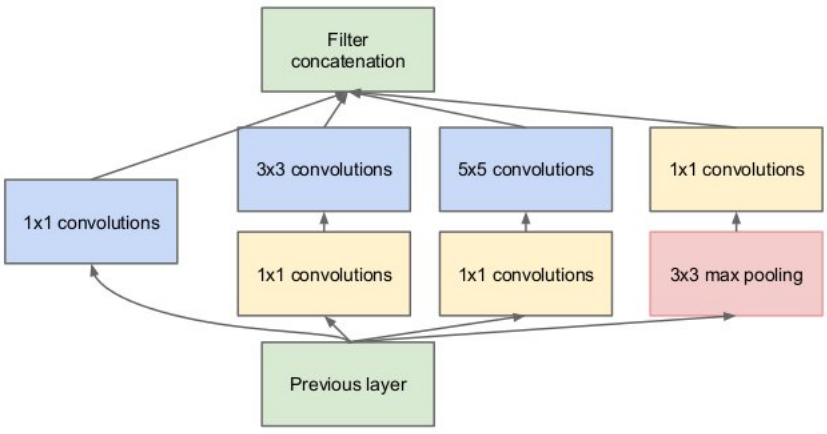


Figure 4: The inception module allows for multiple convolutional filters to retain both local and high abstraction features while reducing the dimensionality of the network.

Complete Inception Architecture

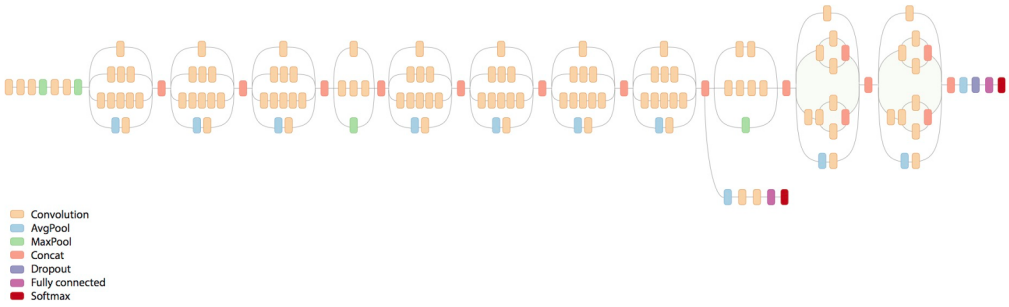


Figure 5: The inception module allows for multiple convolutional filters to retain both local and high abstraction features while reducing the dimensionality of the network.

Table of Inception Layers

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Table 1: GoogLeNet incarnation of the Inception architecture

3.6.1 Calculating the Total Number of Operations

In order to demonstrate how to calculate the total number of parameters and the dimensionality reduction employed by the inception module, we can

employ the following formula:

$$Number of Operations = F \cdot D \cdot I \cdot M$$

where F is the convolutional filter size, D is the matrix dimension of the input, I is the the number of feature map inputs and M is the number of feature map outputs.

Operations Example

For a $5x5$ convolutional filter, with $D = 28$ x 28, $I = 192$, $M = 32$, the total number of operations would be

$$(5 \cdot 5)(28 \cdot 28)(192)(32) = 120,422,000 \quad operations$$

For a $1x1$ convolutional filter, with $D = 28$ x 28, $I = 192$ and $M = 16$

$$(1 \cdot 1)(28 \cdot 28)(192)(16) = 2,408,448 \quad operations$$

If we were to place the $1x1$ convolutional filter in front of the $5x5$ convolutional filter, and calculate the same size feature map as the aforementioned 5x5 filter, we would reduce the number of feature map inputs to 16 giving the following calculation:

$$(1 \cdot 1)(28 \cdot 28)(192)(16) + ((5 \cdot 5)(28 \cdot 28)(16)(32)$$

$$2,408,448 + 10,035,200 = 12,443,648 \quad operations$$

By placing the $1x1$ convolutional filter in front of the $5x5$ convolutional filter, the we maintain the same 28x28x32 output volume but only using $\frac{12,443,648}{120,422,000} =$ .10333% of the original $5x5$ filter operations.

3.7 Viewing the Network in TensorFlow

As mentioned before, the Inception V3 architecture is defined by the Tensorflow commands, functions, and operations laid out in the `inception_v3.py` script. When executing the `train_image_classifier.py` script, the computational graph is built and the `graph.pbtxt` file is recorded in your training directory. The computational graph can be viewed in your Tensorboard's *Graphs* dashboard. Due to the large number of layers and operations in the network, the following screenshots will give a breakdown of the major parts of the computational graph as viewed in Tensorboard.

3.7.1 Parallel Read Module

Ideally, the placement of the TFRecords created in the `build_imagenet_data.py` would be spread across multiple SSD's, multiple raid 0 HDD's, or NAS storage. The parallel read module would then shuffle through a list of textitfilenames that would be sent to individual reader operations set up for each hard drive. These reader operations would then use the randomly shuffled file names to send to the common queue.

Parallel Read Module

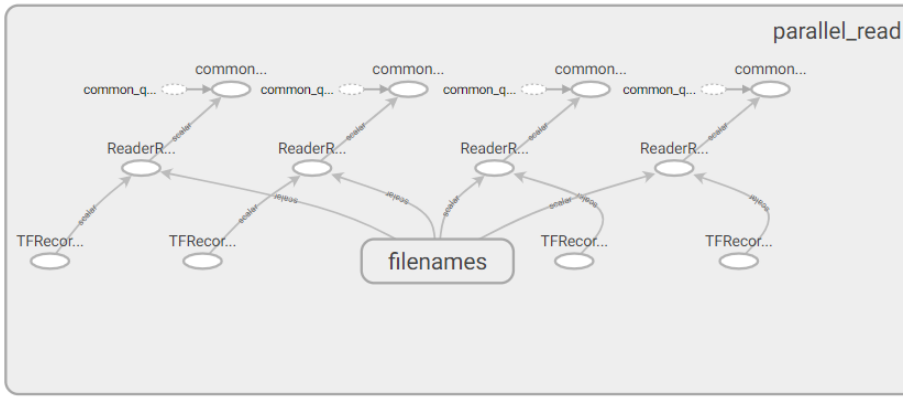


Figure 6: Snap shot of the parsed TFRecords to be shuffled into the common queue and dequeueed by the *ParseSingleExample* module for pre-processing.

CPU Operations

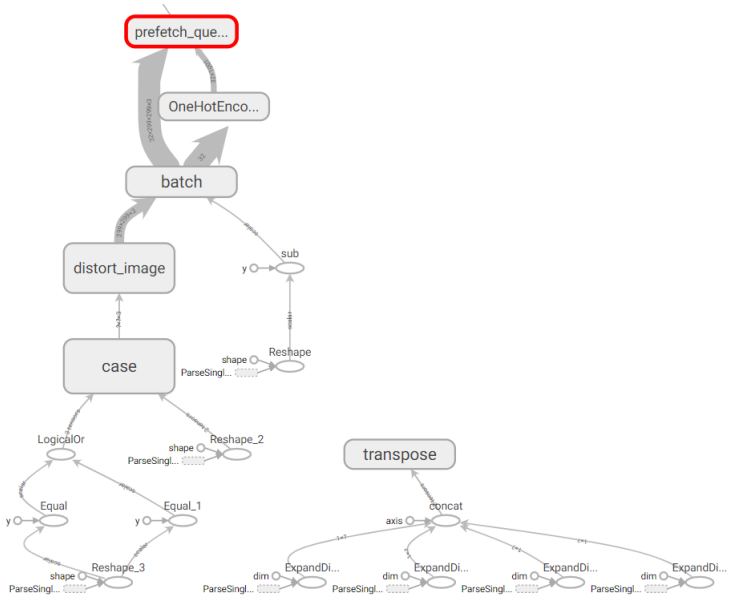


Figure 7: The JPG images to be trained will enter the pre-processing phase at the *ParseSingleExample* input module, follow the pre-processing algorithm, batched, and entered into the prefetch queue to be dequeueed by the GPU.

GPU Operations

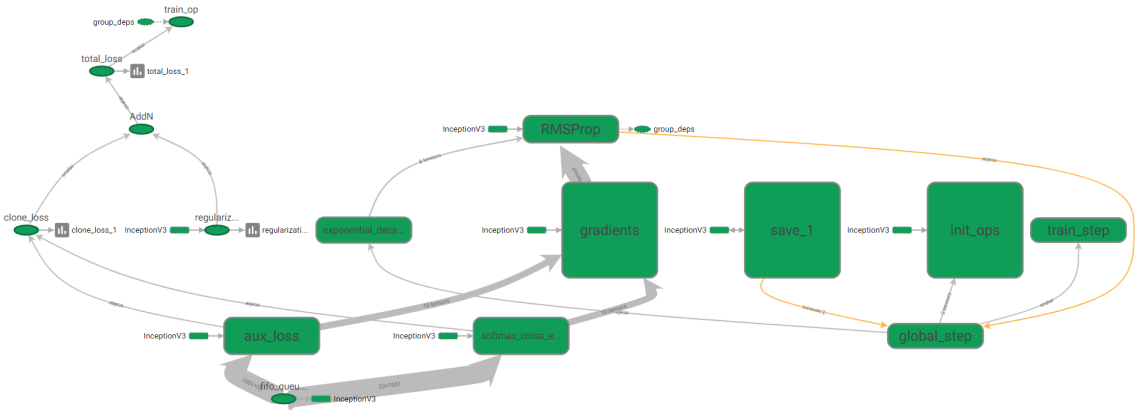


Figure 8: The JPG images to be trained will enter the pre-processing phase at the *ParseSingleExample* input module, follow the pre-processing algorithm, batched, and entered into the prefetch queue to be dequeueed by the GPU.

3.8 Hyperparameter Tuning

When working on a large data set that could take days or even weeks to train, training speed is a crucial factor and is heavily dictated by the hardware specifications. Given your specific hardware, the learning rate and batch size schedule play a huge role in balancing CPU speed, available GPU memory, and overall overhead of the model computations. Larger batch sizes pave the way for more efficient computation of gradients but require more GPU memory to do so. The following parameters will need to be set efficiently to implement the network:

- **Number of Epochs**: Within the context of an iterative machine learning process, an epoch can be described as one full training cycle through the entire training set. Although we would like to train our network with a large number of epochs, it may be more computationally efficient to choose a convergence criteria or set a reasonable maximum given hardware constraints.
- **Initial Learning Rate**: The learning rate controls how much we adjust the weights per each iteration with respect to the gradient. A low learning rate will make small steps, capturing more of the local minima values but could take a very long time to converge. A learning rate that is too high can cause the gradient descent to overshoot the minimum, which could cause the network to not converge or even diverge.

- **Batch Size**: The batch size refers to the number of images processed by the network in one iteration. The batch size can take on three modes: batch, mini-batch, and stochastic batch. The most practical example would be mini-batch with a batch size set to 2^c where c is any constant greater than 0 (i.e 64, 128, 256).
- **Learning Rate Decay Factor**: Learning rate schedules are predefined reductions in the learning rate during training. In most networks, the learning rate is set to a constant amount but more adaptive methods are being used to balance the choice between training speed and the optimal gradient descent step size. The learning rate factor is applied to adaptive learning rate schedules to instruct the gradient descent algorithm how to reduce its step size. Adaptive methods for learning rate schedule include step decay, time-based decay, and exponential decay. In our model, we will implement Adam Optimization.

3.9 Activation Functions

The choice of which activation function to use at each layer of a convolutional neural network is critical to the training performance and robustness of the model at test time. Although, many functions including the sigmoid, the hyperbolic tan, or the exponential linear unit(ELU), have shown acceptable results, none has been used more in practice than the Rectified Linear Unit (ReLU).

3.9.1 Rectified Linear Unit (ReLU)

The ReLU activation function is a thresholded function that activates and becomes its identity above 0. In mathematical terms, it is

$$f(x) = max(0, x)$$

This activation function has been used in nearly all of the state-of-the-art image detection networks since 2010. A number of reasons for this trend can be summed up by the following:

- Since the ReLU activation function has a simple derivative calculation of 1 when the unit is active, it has been shown to substantially increase optimization times. The faster convergence could be a result of the less expensive compute operations compared with the computation of exponentials of sigmoid activation neurons. The ReLU activation also suffers dramatically less from the exploding or vanishing gradient problem due to its on and off linear/non-saturating characteristics.

3.9.2 Softmax Activation Function

The softmax activation function can be described as

$$y_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

The purpose of the softmax activation function is to take the continuous values from the last hidden layer and scale them to probability predictions. Since our softmax function gives us probabilities for each class label, the sum of all the predicted class probabilities should sum to one.

3.9.3 Cross Entropy Loss Function

The cross entropy loss function can be described as

$$H_{y'}(y) = -\sum_{i} y^{'} log(y_i)$$

The cross entropy loss, or negative log-likelihood loss, is a metric of classification performance on probability outputs between 0 and 1. As the predicted probability diverges from its actual class label, the cross entropy will increase in magnitude. A model with 100 percent prediction accuracy would have a negative log-liklihood loss of zero.

3.10 Batch Normalization

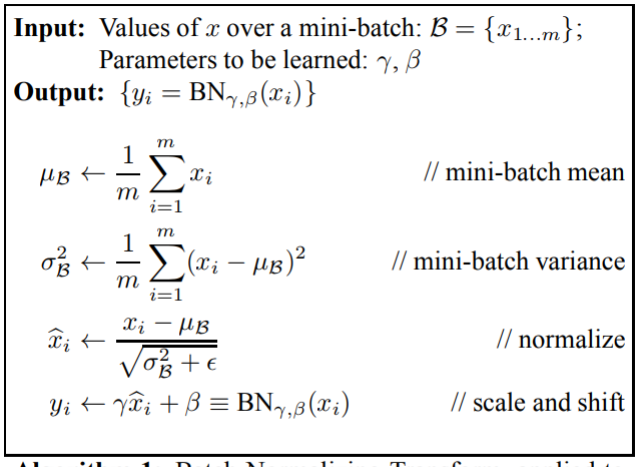
Batch Normalization is the process of normalizaing or standardizaing the inputs or outputs of a network layer. It is used to increase the stability of calculations of a convolutional neural network by shifting and scaling activation inputs or outputs. Without batch normalization, deep convolutional neural networks can suffer from covariate shift, or changes in distributions of layer inputs. This change in distribution is a result of saturated neurons that are stuck by heavy weights in the extreme region of the non-linear activation functions. Batch normalization solves this problem by addressing the layer means and standard deviations. The benefits of batch normalization include the following:

- **Higher Learning Rates**: When working with normalized data, a convolutional neural network is less susceptible to exploding or vanishing gradients allowing the learning rate to be increased. The instability of gradient calculations is largely due to choosing a learning rate that is too small or too large. If the learning rate can be increased, model training can be accelerated.

- **Eliminate Dropout(or greatly reduce strength**): Since batch normalization adds noise to the results by changing the nature of the cost function, batch normalization can effectively add a regularizing effect to the model which can eliminate or greatly reduce the need for dropout.

3.10.1 Batch Normalization For Inception

Batch Normalization Algorithm



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Figure 9: Inception V3 utilizes the Batch Normalization algorithm from the paper *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* by Ioffe and Szegedy, 2015.

3.11 Backpropogation

3.12 Asynchronous Stochastic Gradient Descent

In a distributed network, multiple GPU, or multi-core cpu architecture, Tensorflow allows the user to choose to implement gradient descent using asynchronous or synchronous calculations. When using synchronous stochastic gradient descent, the host machine will employ local workers to perform gradient computations on their own individual mini-batches, and calculate the global gradients. Although this process may be more accurate and stable, it is very slow since all local workers must complete their mini batch gradient calculations before the global gradient is calculated.

To improve the speed of training our convolutional neural network, our network employed asynchronous stochastic gradient descent(ASGD). ASGD has the advantage of allowing local workers to continually train mini-batches without having to wait for other local workers to complete their own mini-batches. The disadvantage of ASGD is that gradient updating may regress at times due to local workers updating the weights and biases on stale parameters.

3.13 Adam Optimization

Adaptive Moment Estimation for Lower-Order Moments

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Figure 10: Inception uses the ADAM algorithm from the *ADAM: A Method For Stochastic Optimization* by Kingma and Ba, 2015.

4 Evaluation

4.1 Notation

For each photo input, the algorithm will output 5 class labels denoted $c_i, i = 1, 2, ...5$ in decreasing order of confidence. The algorithm will also output 5 bounding boxes denoted $b_i, i = 1, 2, ..., 5$ for each class label. The ground truth labels are for the image $C_k, k = 1, 2, ..., n$ class labels.

4.2 What is the function we want to minimize?

The error of the algorithm for each individual image for classification-only is given by the following:

$$e = \frac{1}{n}\sum(min(d(c_i, C_k))$$

The function that we are trying to minimize for classification with localization is given by the following:

$$e = \frac{1}{n}\sum(min_i min_m max(d(c_i, C_k), f(b_i, B_{km})))$$

where d is 0 if your algorithm predicts the same class as the bounded class label and $f(b_i, B_{km}) = 0$ if your algorithm's bounding box overlaps the ground truth bounding box by more than 50 percent.

4.3 What is considered a correct detection?

A detection is considered true or false positives based on an area of overlap. For the ImageNet Object Detection challenge, the area of overlap between my algorithms predicted bounding box and the ground truth bounding box must exceed 50 percent to be considered a correct detection. The formula to calculate the area is given by

$$a_o = \frac{area(B_p \cap B_{gt})}{area(B_p \cup B_{gt})}$$

For a given image, lets say there are three ground truth bounding boxes denoted by some vector $(B^0_{gt}, B^1_{gt}, B^2_{gt})$. If your algorithm predicts a vector of four bounding boxes $(B^0_p, B^1_p, B^2_p, B^3_p)$, your algorithm must iterate over the predicted bounding boxes to discover a **match** with the ground truth bounding box and class. If the conditions of a match are met, the minimum error for a given image will result in a zero, otherwise, the minimum error is 1.

4.4 What is considered a match?

ImageNet strictly defines a *match* to be the following:

- the predicted class label and its bounding box are the exact same as the ground truth label and bounding box, AND
- the predicted bounding box has more than 50 percent overlap with the ground truth bounding box

4.5 How are multiple detections of the same instance treated?

If your algorithm predicts multiple bounding boxes for the same object instance then only one will be counted as the correct detection provided that at least one of them meet the match criteria. For example, if your algorithm predicts 6 detections of the same object and all your predictions meet the match criteria, then 1 will be counted as a correct prediction and 5 will be false detections. False detections will hurt overall accuracy.

4.6 Algorithm for Minimimum Error

In order to iterate over all your predicted bounding boxes, the following code must be used to determine the minimum error:

```
min_error_list = []
for prediction_box in [p_0, p_1, p_2]:
    max_error_list = []
    for ground_truth_box in [g_0, g_1]:
        if label(prediction_box) == label(ground_truth_box):
            d = 0
        else:
            d = 1
        if overlap(prediction_box, ground_truth_box) > 0.5:
            f = 0
        else:
            f = 1
        max_error_list.append(max(d,f))   # the first max
    min_error_list.append(min(max_list))  # the first min

return min(min_error_list)  # the second min
```

4.7 Evaluating the ImageNet Validation Data

Output From Evaluating the Validation Data

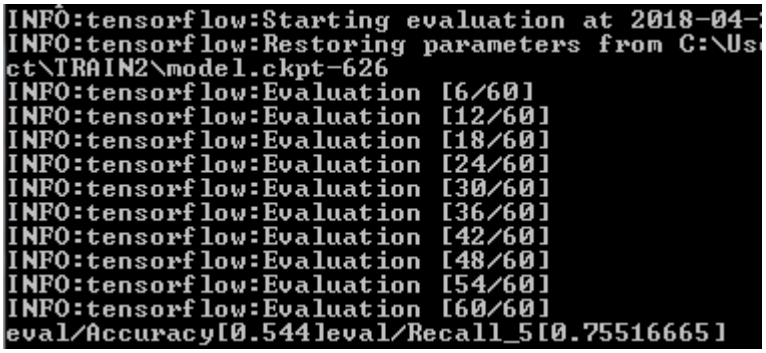


Figure 11: The accuracy measures the percentage correct of the Top-1 classification. The Recall 5 measures the accuracy that the correct class was within the Top 5 predictions.

5 Image Classifier Output

```
swing (score = 0.19141)
shovel (score = 0.15934)
parallel bars, bars (score = 0.15499)
horizontal bar, high bar (score = 0.03545)
chime, bell, gong (score = 0.03447)
```

```
apron (score = 0.92674)
paintbrush (score = 0.02684)
pajama, pyjama, pj's, jammies (score = 0.01278)
spatula (score = 0.00921)
handkerchief, hankie, hanky, hankey (score = 0.00231)
```

```
home theater, home theatre (score = 0.64070)
entertainment center (score = 0.05868)
television, television system (score = 0.05760)
tricycle, trike, velocipede (score = 0.01467)
vacuum, vacuum cleaner (score = 0.01444)
```

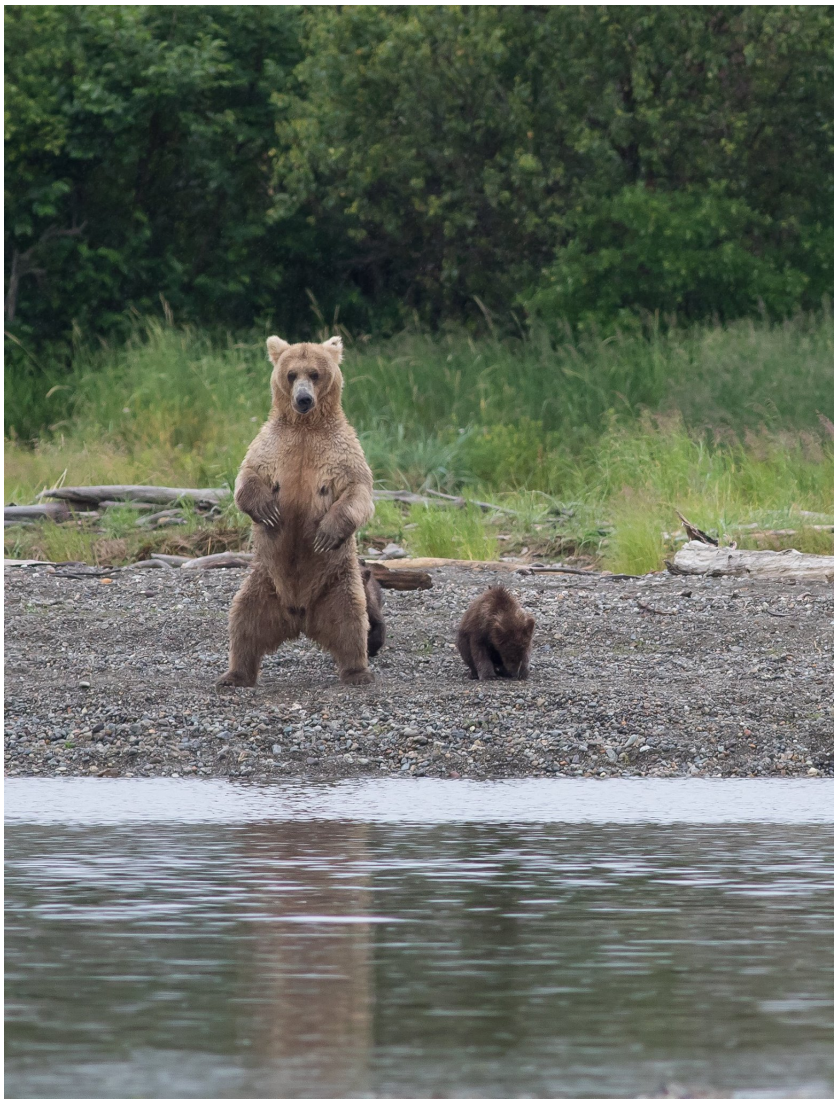
```
seashore, coast, seacoast, sea-coast (score = 0.41154)
umbrella (score = 0.05807)
sarong (score = 0.04080)
maillot, tank suit (score = 0.03249)
maillot (score = 0.02926)
```

```
stage (score = 0.25885)
loudspeaker, speaker, speaker unit, loudspeaker system, speaker system (score = 0.06455)
laptop, laptop computer (score = 0.05895)
computer keyboard, keypad (score = 0.04785)
mouse, computer mouse (score = 0.04469)
```

```
horse cart, horse-cart (score = 0.93709)
oxcart (score = 0.00455)
barn (score = 0.00231)
thresher, thrasher, threshing machine (score = 0.00176)
ox (score = 0.00145)
```

```
vulture (score = 0.65422)
little blue heron, Egretta caerulea (score = 0.05282)
European gallinule, Porphyrio porphyrio (score = 0.04539)
black grouse (score = 0.02519)
bald eagle, American eagle, Haliaeetus leucocephalus (score = 0.01601)
```

```
bald eagle, American eagle, Haliaeetus leucocephalus (score = 0.91173)
kite (score = 0.01935)
vulture (score = 0.00074)
velvet (score = 0.00050)
black grouse (score = 0.00048)
```



```
brown bear, bruin, Ursus arctos (score = 0.95635)
Airedale, Airedale terrier (score = 0.00609)
Irish water spaniel (score = 0.00138)
teddy, teddy bear (score = 0.00125)
Irish terrier (score = 0.00077)
```

```
brown bear, bruin, Ursus arctos (score = 0.95426)
bison (score = 0.00335)
bottlecap (score = 0.00117)
lesser panda, red panda, panda, bear cat, cat bear, Ailurus fulgens (score = 0.00078)
hyena, hyaena (score = 0.00043)
```

```
brown bear, bruin, Ursus arctos (score = 0.98223)
ice bear, polar bear, Ursus Maritimus, Thalarctos maritimus (score = 0.00084)
Irish water spaniel (score = 0.00032)
bottlecap (score = 0.00025)
American black bear, black bear, Ursus americanus, Euarctos americanus (score = 0.00025)
```

```
brown bear, bruin, Ursus arctos (score = 0.29495)
lesser panda, red panda, panda, bear cat, cat bear, Ailurus fulgens (score = 0.09876)
Norwich terrier (score = 0.02791)
Australian terrier (score = 0.01975)
chow, chow chow (score = 0.01757)
```

```
Lunatic (score = 1)
Windsor tie (score = 0)
academic gown, academic robe, judge's robe (score = 0)
suit, suit of clothes (score = 0)
mortarboard (score = 0)
```