Image Localization and Detection

Using Python, 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 The 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 Raw Files

The raw files of the dataset reside in JPEG file format. The image resolutions vary by color gamut, image size, 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.

2.3 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.

2.4 Image Pre-processing

2.4.1 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)

2.4.2 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 (Image Augmentation).
6. Collect images in a batch until the batch size is met.

2.4.3 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.

2.4.4 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.

2.4.5 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.

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 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.

3.3 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).

3.3.1 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.

3.3.2 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.

3.3.3 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.

3.4 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.

3.4.1 Batch Normalization Mathematics

$$\mu = \frac{1}{m}\sum_{i=1} z^i$$

$$\sigma^2 = \sum_i (z_i - \mu)^2$$

$$z^i_{norm} = \frac{z^i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

3.5 Backpropogation

X: Tensor pairs of inputs of x_i and class labels y_i

x^ℓ_i: Input x of node i at layer ℓ

y^ℓ_i: Output y of node i at layer ℓ

w^k_{ij}: The weight of node j in layer ℓ_k coming from node i

b^k_i: The bias of node i in layer k

θ: the grouping of parameters, w and b

$y^{'}_i$: the actual predicted class label

α: the learning rate of our update function

3.6 Asynchronous Stochastic Gradient Descent

When training a convolutional neural network using the gradient descent algorithm, the network requires gradient computations of the chosen error function $w.r.t$ the weights w^k_{ij} and biases b^k_i commonly grouped together as θ. Gradient descent will iteratively update the weight and bias at each node and its speed is set by the hyperparamter α, which is the learning rate. The function to iteratively update the weights and bias is:

$$\theta^{t+1} = theta^t - \alpha \frac{dE(X, \theta^t)}{d\theta}$$

3.7 Adam Optimization

Adam Optimization is an optimization method that combines the features of RMSprop and momentum.

3.8 Non-Maximum Surpression

Omitted.

3.9 Anchor Boxes

Omitted.

4 Complications That May Arise

4.1 Multiple Object Detection Dilemma

When predicting the bounding box given a correct prediction of a single class object, it is foreseeable that the algorithm will produce multiple bounding boxes for the same instance of the object class. Since convolutional neural networks rely on the construction of w x h grids through the implementation of smaller dimension convolution filters, objects could span across many grid cells creating a dilemma for where to place the anchor box midpoints.

4.2 Exploding or Vanishing Gradients

Gradient-based learning methods allow for computers to apply iterative methods to minimize objective functions through a process called back-propogation. In any given layer a weight must be updated proportional to the gradient of the error function with respect to each iteration of the current weight. When using non-linear functions such as the hyperbolic tangent, the gradient could be stuck in the very extreme tails of the function causing the gradient to be very small or very large. If the gradient is very small, this would prevent further updates to the weights on any given layer and in the worst case, halt the model from training at all.

4.3 ReLU Dead Neuron Dilemma

The same on and off activation threshold that make the ReLU activation function very

5 Evaluation

$\vec{y} = \langle P_c, b_x, b_y, b_h, b_w, c_1, c_2, c_2, c_3 \rangle$

5.1 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.

5.2 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.

5.3 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.

5.4 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

5.5 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.

5.6 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
```