Poe Dameran – Mathematical Review

The Problem

The problem of object detection is as follows. In one image or frame, there are n number of objects or 'entities' in view, where n may be any number greater or equal to 0 (n=0 where there are no objects in view). Our model must be able to detect and classify the majority of objects with accuracy and precision.

The frame may be represented by a grid with P width and Q height, and each point on the grid representing a pixel on the frame. Each entity in the frame can be represented by a bounding box with $\mathrm{B} = (\mathrm{x}, \mathrm{y}, \mathrm{h}, \mathrm{w})$, where:

- (x, y) refers to the x and y co-ordinates of the top-left corner of the bounding box, where $0 \leq x \leq P$ and $0 \leq y \leq Q$.
- h and w refer to the height and width of the bounding box respectively.

Each entity represented by a box can be given a class. The class is represented by the integer c, where $0 \leq c \leq 12$. Class 0 refers to an ignored area, classes 1 to 11 refer to an object located in our dataset, and 12 refers to any object that exists but would not be considered for one of the previous entity classes. The model determines a score s for each entity, which represents the probability that the entity is of the determined class.

To summarise, each entity e can be represented as

$$e = \{B_i, c_i, s_i\}$$

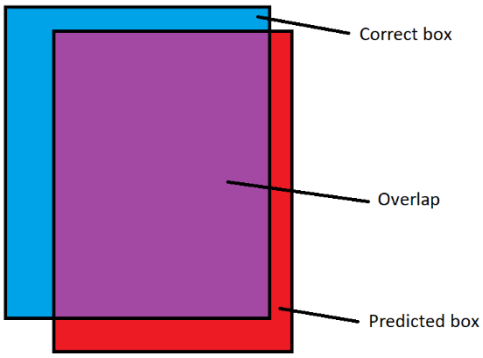
Where B is the entities bounding box, c is the class determined by the model, s is the probability or confidence score, and i is the index of the entity in the total collection of all entities found.

How our solution will be assessed

To evaluate how competent our solution is, we will use the following metrics: Frames per second (FPS), recall, precision, and intersection of union (IoU) of bounding boxes.

FPS is an especially important metric for our model as it our problem domain (as described earlier in the report) sometimes involves real-time detection of entities, so a solution that can process many frames per second is more favourable than a model that achieves a lower number of frames. We calculate the FPS for the model by predicting boxes for a set number of frames and timing how long it takes to predict for the entire set.

Precision and recall are important for most computer vision models and are two of the most useful metrics we can use. Precision refers to the proportion of bounding boxes that are identified as their correct classes, or $P = \frac{(correct\ boxes \cap predicted\ boxes)}{predicted\ boxes}$. To make assessment of the model easier, we determined the average precision for each class (AP), and then determined the mean average precision (mAP). Recall refers to the proportion of correct bounding boxes that are predicted by the model, or $R = \frac{(correct\ boxes \cap predicted\ boxes)}{correct\ boxes}$. Like precision, we calculate the average recall for each class (AR), and the mean average recall (mAR).



The IoU (as by Jaccard, P. [1]) refers to the percentage of the area of a correct bounding box is covered by the corresponding bounding box.

$$IoU = \frac{(Predicted\ Box \cap Correct\ Box)}{(Predicted\ Box \cup Correct\ Box)}$$

This metric is used to evaluate how well the predicted bounding boxes cover the correct boxes.

The model

Our model uses a Faster R-CNN model [2] as its backbone. The backbone was pretrained on the COCO train 2017 dataset before being trained on our model.

RPN Loss function

The loss function used in Faster R-CNN's Regional Proposal Network (RPN) is a multi-task loss function designed to tackle the tasks of classification and box-localisation at the same time.

The loss function is as follows:

$$L(\{p_i\},\{t_i\}) = \frac{1}{N_{cls}}\sum_i L_{cls}(p_i, p_i^*) + \lambda\frac{1}{N_{reg}}\sum_i L_{reg}(t_i, t_i^*)$$

Where:

- i is the index of the anchor in the mini-batch
- p_i is the response probability that anchor i is an object
- p_i^* is the ground truth of the label, being 1 if the anchor is true or 0 if false
- t_i is the coordinates of the centre of the predicted bounding box contained in a vector
- t_i^* is the coordinates of the true bounding box contained in a vector
- N_{cls} and N_{reg} are used to normalize their respective loss functions, where N_{cls} is the mini-batch size, and N_{reg} is the number of anchor locations total
- λ is a balancing parameter used to determine weighted sides the function ($\lambda = 10$ by default.

$L_{cls}(p_i, p_i^*)$ is log loss over the two classes, or

$$L_{cls}(p_i, p_i^*) = -p_i^* \log p_i - (1 - p_i^*)\log(1 - p_i)$$

This is used to translate a multi-class classification to a binary classification (whether an object belongs to one of many classes to whether an object is an object or not). Here, if p_i^* is 0, L_{cls} will be -1 regardless of the value of p_i, whereas when p_i^* is 1, the value of L_{cls} can be between -3.2 and +1.2 depending on the value of p_i.

$L_{reg}(t_i, t_i^*)$ is given by the robust loss function, also called the smooth L_1 function [2] and is used to determine the distance between t_i^* and t_i while being resistant to outliers.

$$L_1^{smooth}(x) = \begin{cases} 0.5x^2, & if\ |x| < 1 \\ |x| - 0.5, & otherwise \end{cases}$$

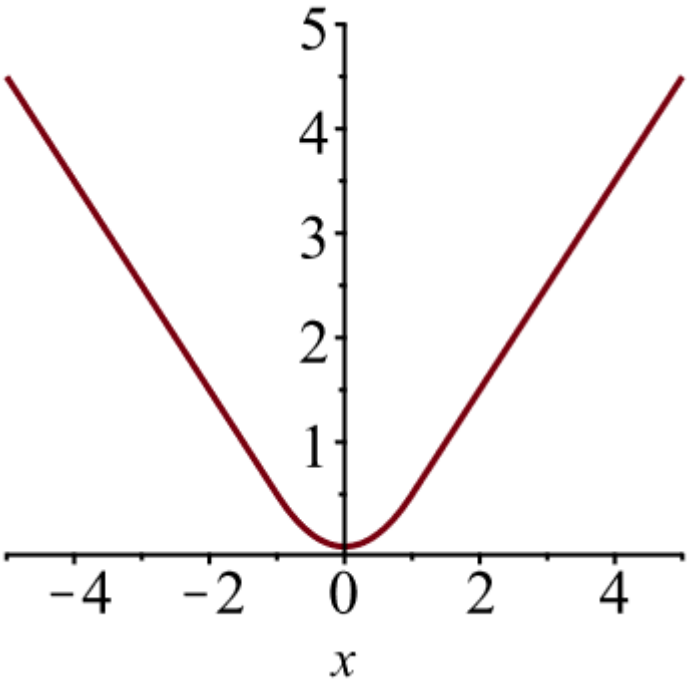


Figure 1 - Plot of y vs. $L_1^{smooth}(x)$, from [2]

The general loss function can be split into two parts which cover each purpose: the classification part (purple) and the box-recognition part (yellow).

$$L(\{p_i\}, \{t_i\}) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i^*) + \lambda \frac{1}{N_{reg}} \sum_i L_{reg}(t_i, t_i^*)$$

The overall, it helps reduce the number of incorrect predictions during training by only assigning positive values to anchor boxes with the highest IoU overlap with true boxes, or any predicted box with 70% overlap with any true box.

A parameter of a model set at the before the training process called the weight decay, which is added to the loss returned from the loss function. The weight decay is multiplied by the sum of the weights of the network squared, which allows the weights to be represented in the loss function (which prevents the weights from becoming very large and thus making the model overly complex) but without making the loss function use overly large value.

Optimisation function

When the Faster R-CNN research was published, it used stochastic gradient descent (SGD) and back-propagation during training, however it has later been implemented to use the Adam [3] optimiser as well.

SGD is a method of stochastic optimisation, which aims to find the smallest average value of an optimisation function by adjusting parameters. Through multiple iterations, the parameters

experience slight changes that adjust them towards being equal to the “true” parameter. Most variations of SGD feature the learning rate parameter. Initially, the learning rate is high to bring the parameters close to the “true” parameter and is gradually reduced so the changes can become more accurate and precise. Each iteration of the output in one instance of SGD can be found using:

$$\theta_{t+1} = \theta_t - \eta_t g_t$$

Where θ is the output, η is the learning rate, g is the optimisation function (for standard SGD, $g_t = \nabla \mathcal{L}_t(\theta_t, z_t)$), and t is the index of the values. [4]

Adam (short for ‘adaptive moment estimation’) is a descendant of the SGD method which includes root mean squared adaptive learning rates that decreases over time and momentum-based methods that increase the rate of descent at points where there is little change to the gradient and decreases the rate of descent at points where there is a large change. These changes prevent the effective learning rate to become too large too early, while reducing the number of overall computations that need to be made.

For the Glie-44 model, a pre-trained network has been selected which includes network weights already trained on a large database. To make the model ready to use in the new context, it must be fine-tuned to the new dataset. The process of fine-tuning requires small, fine changes to the network weights since the pre-trained weights are already close to being appropriate, so the learning rate during this process must be small to reduce the magnitude of the changes to the weights.

We can see this in the overall formula for one iteration of the output of SGD why the learning rate should be low for fine tuning. If f is the index of the point in the iteration after the pre-training process, then:

$$\theta_{f+1} = \theta_f - \eta_f g_f$$

θ_{f+1} should be close to θ_f since the difference between values during fine-tuning is very small, and since g_f will not change very much between iterations, the learning rate η_f needs to decrease.

Figure 2 - Diagram of iteration step vs difference between the parameter's current value and target. The pre-training eventually levels out due to decreasing learning rates, but when the fine-tuning process begins the value experiences larger changes once again, but still not as great as the beginning of the pre-training period.

The learning rate in Glie-44 is changed in steps. A step size and gamma parameters are set at the beginning, then during the learning process, the learning rate is multiplied with the gamma at every epoch equal to the step size. For example, a model using a step size of 10, a gamma of 0.1 and a learning rate of 0.2, would use the current step size for every epoch until after the 10^{th} epoch, where the learning rate becomes 0.02

From epochs	Learning rate from example
1 to 10	0.2
11 to 20	0.02
21 to 30	0.002
31	0.0002

The optimal choices for these parameters can be determined through testing, where the model is tested using a set of values for each parameter, the accuracy and precision for the model using each value is determined and compared to a baseline. Normally, values greater than or less than the target value of each parameter return worse metrics, so the metrics for each value (for example, accuracy) can be plotted on a graph, forming a peak when close to the target value for that parameter.

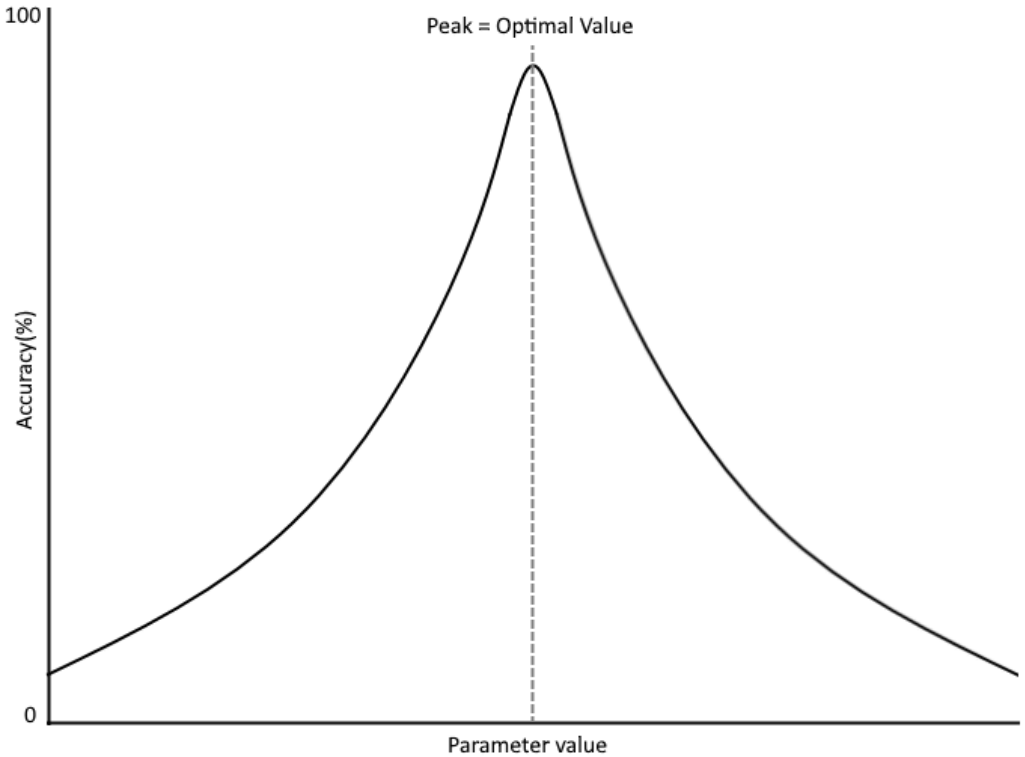


Figure 3 - Parameter value vs Accuracy. The peak represents the value that obtains the highest accuracy so the optimal value for that parameter (Depending on the number of values tested and their differences, graph could be less smooth but will take this overall shape)

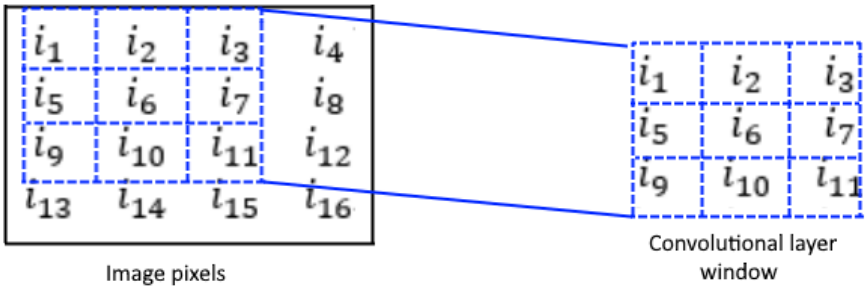
Convolutional layer

In Faster R-CNN, convolutional layers are used to create a condensed feature map of the original image, which can then be more effectively used in later layers of the network.

The convolutional layer uses a kernel vector of weight values that are used to obtain the feature values from sections of the original images. The layer uses a "window" of a fixed size, usually 3x3, that takes the pixel values of a small portion of the original image and multiplies those values with the corresponding kernel weights (the element-wise multiplication of the kernel vector and the window vector). The sum of these intermediate values is taken and added to the corresponding place in the new feature map. The window then moves by a set amount and repeats this process for the entire image. [5]

$$kernel = \begin{pmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{pmatrix}$$

$$image = \begin{pmatrix} i_1 & i_2 & i_3 & i_4 \\ i_5 & i_6 & i_7 & i_8 \\ i_9 & i_{10} & i_{11} & i_{12} \\ i_{13} & i_{14} & i_{15} & i_{16} \end{pmatrix}$$

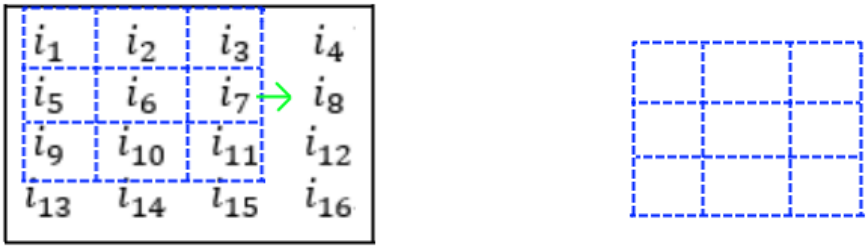


$$s1 = kernel * window = \begin{pmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{pmatrix} * \begin{pmatrix} i_1 & i_2 & i_3 \\ i_5 & i_6 & i_7 \\ i_9 & i_{10} & i_{11} \end{pmatrix}$$

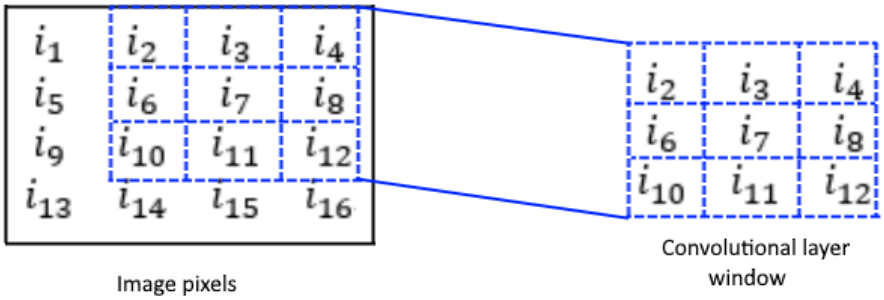
$$s1 = (w_1 * i_1) + (w_2 * i_2) + (w_3 * i_3) + (w_4 * i_5) + (w_5 * i_6) + (w_6 * i_7) + (w_7 * i_9) + (w_8 * i_{10}) + (w_9 * i_{11})$$

$$feature\ map = \begin{pmatrix} s1 & & \\ & & \\ & & \end{pmatrix}$$

Window changes position by an amount equal to the stride size.



Repeats the process until the end of the image is reached and the feature map is filled.



$$feature\ map = \begin{pmatrix} s1 & s2 & s3 \\ s4 & s5 & s6 \\ s7 & s8 & s9 \end{pmatrix}$$

Subsequent convolutional layers can use the feature map generated by the first layer and produce even more condensed feature maps.

Since the convolutional window size is usually a much smaller than the original image, the window may have to adjust its position many times for one image, which leads to many computations that could reduce performance for larger images. To increase the speed, the window size can be increased, or the number of spaces that the window moves each time (the stride) can be adjusted. Most often the image is resized to a portion of the original, which reduces the number of computations needed in one pass and prevents the window from leaving "remainders" when the image size does not completely divide by the window size and stride size.

References

[1] P. Jaccard, "THE DISTRIBUTION OF THE FLORA IN THE ALPINE ZONE," *New Phytologist,* vol. 11, no. 2, pp. 37-50, 1912.

[2] R. Girshick, "Fast R-CNN," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015.

[3] D. P. Kingma and J. L. Ba, "Adam: A Method for Stochastic Optimization," in *ICLR 2015 : International Conference on Learning Representations 2015*, 2015.

[4] K. Patrick Murphy, "Stochastic Gradient Descent," in *Probabilistic Machine Learning: An Introduction*, MIT Press, 2021, pp. 285-293.

[5] M. Umberto, "3. Fundamentals of Convolutional Neural Networks," in *Advanced Applied Deep Learning : Convolutional Neural Networks and Object Detection*, Berkeley, CA, Apress, 2019.

[6] S. Ren, K. He, R. Girshick and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 39, no. 6, pp. 1137-1149, 2017.

[7] R. Girshick, "Smooth L1 Loss," [Online]. Available: file:///C:/Users/Benji/AppData/Local/Temp/SmoothL1Loss.1.pdf. [Accessed 8 May 2021].