Mine-RCNN

Loi Dario (1940849), Marincione Davide (1927757), Barda Benjamin (1805213)

Abstract—Real time object detection has recently been made possible due to steady state-of-the-art advancements in the field [1], [2], these methods propose the use of a Region Proposal Network to identify Regions of Interest (RoIs) in the image and correctly classify them, we aim to reproduce the architecture proposed by [2] applied to a novel environment, that of the popular sandbox Minecraft, both for the ease-of-collection of the required data and for a number of graphical properties possessed by the game that make such a complex problem more approachable in terms of computational resources, moreover, due to the novelty of the environment, we also train the entirety of the network from the ground up, having no pre-trained backbone at our disposal.

Index Terms—Object Detection, Convolutional Neural Network, Sandbox, Region Proposal, Real Time Detection

1 Introduction

Traditional visual recognition algorithms employ the use of algorithms such as SIFT or SURF [3], [4], to extract a feature descriptor from key points in the image, a set of sample images is therefore processed in order to obtain descriptors for each class, inference is then performed by extracting the descriptor from the input image and then comparing it with our samples, using a distance criterion to select the closest class.

While these traditional methods are mostly invariant to scale and translations, issues such as variability in the brightness or rotation of the target object are problematic, moreover, the computational cost of extracting features is prohibitively high, effectively preventing the possibility of deploying the model in a real-time context.

Our Approach: One way to reduce the cost in performance while gaining a higher expressive capacity for our model is to employ modern Deep Learning techniques to build our detector, our aim is therefore to design a scaled-back version of [2], stopping ourselves short of the classification stage, the reason for this will be further clarified in the rest of the paper, our architecture[1] is composed of a FCNN [5], [6] Backbone, this is to keep the number of parameters as low as possible as well as allow the initial layer of our architecture to be applicable to images of any resolution, our Convolutional Backbone must be trained from the ground up in the task of simply classifying the classes that we intend to detect, the classifying layer is then removed from the model and we're then left with its feature map output which we then pre-process by *splashing* anchors through a sliding window approach.

After this, both feature maps and anchors are then passed to our region proposal layers, which in turn is composed of a twin ensemble of Neural Networks that perform Regression and Classification on the bounding boxes, the outcome is a set of *positive* and *negative* bounding boxes, the boxes that are classified as positive go through NMS [7] using Intersection over Union (IoU) [8] as a metric for suppression in order to reduce the amount of individual boxes that classify the same object.

Our Environment: In order to keep our project in the realms of feasibility, especially considering the constraints imposed on our team in the realms of time and processing capabilities of our hardware, we chose to detect objects in a virtual environment, this is because the possibility of having complete control over the environment allows us to artificially create scenarios from which we can gather samples of a large quantity, for this reason, we chose the popular sandbox game Minecraft [9], other than the advantage of control over the environment the game is well known for its simplistic approach to Voxel Graphics, rendering the world as a set of blocks of cubic shape, the game also possesses a simplistic lightning model that simply shifts the brightness of blocks' textures when in the vicinity of a light source, thus being void of effects such as bloom, specular components on objects, reflections and other complex techniques that introduce sources of unexpected noise or variability in our samples, this, in turn, allows our model to work in a context that, while still posing a significant challenge, is of suitable difficulty to the scope of this project.

2 Method

2.1 The Dataset

We first started by building a dataset. We recorded one-minutes long videos of Minecraft using commercial screen capture software, we then loaded those shorts into python using the OpenCV library, we then arbitrarily sampled frames from the videos at a rate of 1 frame-per-second, as we assume this rate to be sufficient to provide a good number of samples while also ensuring a good variability in-between our frames.

1. see Figure 1 for a diagram of our architecture

Our Architectures

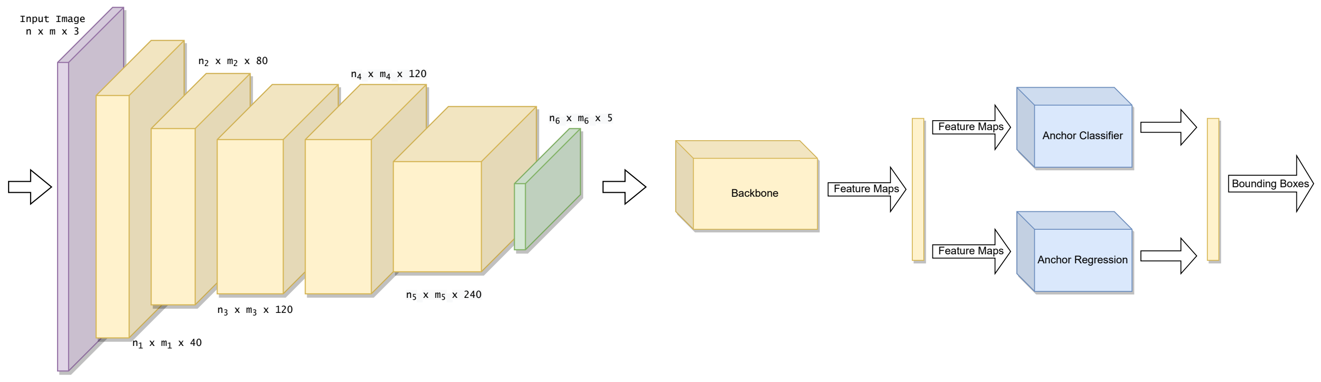


Figure 1: Our FCNN backbone (a) employs a series of convolutions to obtain a feature map to pass to the rest of the network as displayed in (b), the feature maps from our Backbone are first pre-processed by a sliding-window approach that spreads our anchors over the image, this is then passed to a twin neural network, one part of the network performs classification, determining if each bounding box is containing an object or not, the other part performs regression on our bounding box in order to make it better fit an eventual target object.

Figure 2: A Screenshot of the world of Minecraft, note the simplistic graphics and the absence of Bloom (except the illusion thereof provided by the skybox) even though we are staring directly at the sun, note also the presence of a *pig*, one of the entities that we are going to detect in our model.

At first we downsampled the images from 1920×1080 to 426×240 to reduce the memory footprint of the dataset in order to both be able to share it with ease and load it in GPU memory even when the hardware at hand is a common, low capacity commercial card. But then, by convention, we decided it would be better for the image to be at a 1:1 aspect ratio; in order to try and preserve the same amount of information reached beforehand, we decided to choose a size that would keep the area almost constant, 318×318:

$$\begin{aligned} A &= w * h \\ &= 426 \cdot 240 \\ &= 102'240 \\ l &= 318 \approx \sqrt{w * h} \\ \rightsquigarrow l^2 &= 101'124 \approx A \end{aligned} \tag{1}$$

At this time we opted to limit ourselves to only five classes for: Zombies, Creepers, Pigs, Sheep, Nothing.

The first iteration of the Dataset was comprised of images that contained multiple classes in the same scene, this was because our initial idea was to diverge from the implementation described in [2] and implement a particular form of loss called *Triplet Loss* [10], this proved problematic since when training the backbone we were presented with problems of exploding gradients and general failure of the network to converge, this idea was then discarded on the ground of practicality and the focus of the project shifted back to training a traditional classifier that uses Cross-Entropy as a loss function, for this reason the Dataset was also re-done from the ground up to only have a single object in the frame at a time. The next step consisted in manually labelling each one of the sampled frames. We developed a simple but effective tool that allowed us to draw bounding boxes[2] and assign to each one of them a label corresponding to one of the classes mentioned above. During this process we pruned the images that we considered unfit to be part of the dataset (e.g. frames inside the game menu or outside of the game). After standardizing the coordinates of BBoxes we saved them into JSONs files, after having our JSONS files ready we grouped them into a single .dtst[3] file for better integration with `PyTorch`.

Data Augmentation: Due to the reduced amount of resources, manpower, and time available to us when compared to professional teams, we considered *a priori* the size of our dataset to be small[4], and therefore prone to overfitting: we pre-emptively took approaches in order to stave it off, one of those being the applying of a set of random transformations on each input image in order to increase the variability

2. BBoxes for brevity.
3. Our own extension, stands for *dataset*.
4. Funnily enough, the final result of $\approx$ 4000 images is quite a sizeable dataset considering the complexity of our environment.

Table 1: ZG

Layer	Kernel size	IN/OUT channels	padding	stride	Layer Structure
1	7×7	3 / 40	replicate(1)	2	BATCHNORM(3) + CONV + MISH + DROPOUT(.2) + BATCHNORM(40)
2	7×7	40 / 80	replicate(1)	2	CONV + MISH + DROPOUT(.2) + BATCHNORM(80)
3	3×3	80 / 120	replicate(1)	2	CONV + MISH + DROPOUT(.2) + BATCHNORM(80)
4	3×3	120/120	None	1	CONV + MISH + DROPOUT(.2) + BATCHNORM(120)
5	3×3	120/240	replicate(1)	2	CONV + MISH + DROPOUT(.2) + BATCHNORM(240)
6	1×1	240/ 5	None	1	CONV + BATCHNORM(5)
7	$w \times h$	5 / 5	None	None	ADAPTIVEMAXPOOL(1×1)

and artificially inflate the size of our dataset, these were, for the purpose of classification:

1) Reflecting the image horizontally with $p = 0.5$
2) Scaling the image's brightness, contrast and saturation with factors (b, c, s) chosen uniformly over the intervals: $b \in [0.60, 1.40], c \in [0.95, 1.05], s \in [0.99, 1.01]$
3) Randomly downscaling the sharpness of the image with a factor $k_1 = 1.25, p = 0.2$
4) Randomly upscaling the sharpness of the image with a factor $k_1 = 0.75, p = 0.2$
5) Applying a random rotation in the interval $\theta \in [-15°, 15°]$

Whereas, during the training of the full network, only the color-related transformations were kept due to the others requiring non-trivial customizations in order to also be correctly applied to each image's set of ground-truth bounding boxes.

2.2 The Models

Our architecture is divided into two main modules: The Backbone, which we decided to call the *Ziggurat* (ZG), and the Region Proposal Network (RPN).

The Backbone: Ziggurat: The ZG is a Fully convolutional neural network consisting of 6 (plus 1, non learnable) layers as described in table 1, the sixth layer of which will be dropped after the pre-training phase. The purpose of this network is to extract feature maps from the input image for the RPN that follows, it is important then to pre-train this network so that it learns the general embedding in order to reduce the complexity of training the RPN. We decided to add dropout layers as they provide a useful guard against overfitting [11], and, again, considering the dimensions of our dataset we decided to add it to each layer except the last one. For the activation function we decided to use MISH opposed to ReLU used in the original Faster-RCNN as it is already widely accepted as superior in terms of stability and accuracy, due to the smoothness of its first derivative and the lack of vanishing gradient problems that plague ReLU, the weights were initialized using the Kaiming method as proposed by [12].

Anchors: Before diving into the region proposal architecture, we need to pay extra attention to the anchors, which are the core of our whole process. An anchor is defined as rectangle with an associated center, base size and transformations of both its scale and aspect ratio. If we then denote with R the set of ratios and with S the set of scales, the cardinality of the set of anchors is $A = |R| * |S|$.
For our implementation we used $R = \{0.25, 0.5, 1, 2\}$, $S = \{0.5, 1, 2\}$ and a base size of 40, thus having $A = 12$.

After processing the image through the ZG we obtain what is our base feature map (BFM) of size H, W that will be processed further by the RPN, on each feature of the BFM we apply a set of anchors as described above, centered on that feature, this allows us to easily map from the feature map to the original image by computing the total stride of the ZG up to the fifth layer and using that to re-project the anchors from the feature map to the original input. At the end of this process, which we like to call the *Splashing of the Anchors* or *The Splash* in short, we will have a grand total of $H * W * A$ anchors.

RPN: The RPN is the last part of our model. We went through many iterations for its architecture while trying to find a solution that would do good in terms of both accuracy, performance, and size. While the specific architectures have varied a lot during the build of the project the core idea behind all of them stayed the same: after retrieving the feature map and splashing the anchors, we pass the feature map into an additional convolutional layer, we then have two siblings layers, one for classification and one for regression.

The classification layer needs to predict the *probability*[5] of being an object or not for each splashed anchor based on the Intersection over union (IoU) with any ground truth box. The regression layer, on the other hand, tries to predict a set of offsets for each anchor such that, when applied to the corresponding one, will give us our Region Of Interest (RoI).

As for the specific architecture, we first tried to only use convolutional layers for both regression and classification, using 1×1 convolutions over the BFM:

5. The given number is more of a *Score* than a probability due to its distribution not summing up to one, however, we allow this slight abuse of notation as we feel that it describes its purpose better.

while getting *acceptable* results on the classification task, the regression proved to be too complex of a task for the small capacity of our network, so we decided to opt for a linear layer instead.

The architecture chosen is reported in 2. For the classification we used a Sigmoid activation function as the scoring is given in the $[0, 1]$ range. A crucial aspect in this model is how to deal with overlapping anchors: It is easy to see that the amount of anchors for each image would be very large even for a small BFM (e.g. for a 19×19 BFM and $A = 12$ we would have around $4'000$ anchors), moreover most of them would be classified as non object which as we will see do not contribute to the regression task at all. For those reasons we perform non maximum suppression (NMS) over the transformed anchors, allowing us to prune overlapping anchors, keeping only the ones with the highest predicted score.

Table 2: RPN

Layer	Size IN / OUT	Layer Structure
Classification Layer	$C * H * W$ / 4332	Linear + Sigmoid
Regression Layer	$C * H * W$ / 4332 * 4	Linear

Training: We pre-trained ZG on a classification task over the 5 labels mentioned above, using just a portion of our dataset splitting in 60 - 20 - 20 for training validation and testing respectively using frames where only one mob or less of one kind was present. For training we used Cross-Entropy loss function in conjunction with an AdamW optimizer using the AMSGrad variant of the algorithm with a fixed learning rate set to $\gamma = 0.0002$.

Having our backbone pre-trained we cropped the last layer and attached the untrained RPN; it's worth noting that, already at this point, even with no additional training the RoIs proposed were already somewhat giving *interesting* results with respect to the real target.

For the loss function we implemented the one described in [2]:

$$L(\{p_i\}, \{t_i\}) = \frac{1}{N}\sum_i L_{\text{cls}}(p_i, p_i^*) + \frac{\lambda}{N}\sum_i p_i^* L_{\text{reg}}(t_i, t_i^*) \tag{2}$$

where i are the indexes of the anchors in the image, p_i is the predicted score for that anchor and p_i^* is a binary label: we assign 1 to those anchors that have an IoU with respect to any GT BBox greater than 0.45, and 0 to those with IoU smaller than 0.05 we ignore the anchors that falls in between. For L_{cls} we use a BCE loss over the two classes of being an object or not.

On the other hand t_i and t_i^* are vectors containing the parametric coordinates of the predicted positive anchors, and that of the ground truth associated to that anchor respectively, λ is a normalization constant that we set to 10 to prevent that the classification would overwhelm the regression in the objective function. As [2] suggests, we do further normalization based on the number of given samples N: we would like to remind though that doing otherwise would not have changed the curvature of the loss function and therefore the position of their local minima. The coordinates are parametrized as described in [2] :

$$\begin{aligned}
t_x &= (x - x_a)/w_a & t_y &= (y - y_a)/h_a \\
t_w &= log(w/w_a) & t_h &= log(h/h_a) \\
t_x^* &= (x^* - x_a)/w_a & t_x^* &= (y^* - y_a)/h_a \\
t_w^* &= log(w^*/w_a) & t_h^* &= log(h^*/h_a)
\end{aligned} \tag{3}$$

Where x, y, w, h represent the center's coordinates, height, and width of the predicted box, anchor box, and ground truth (e.g. x, x_a, x^*). We decided to split the dataset into 80% training and 20% validation; We trained the network using SGD as suggested in [2] with a fixed learning rate of 10^-5 and a momentum of 0.9

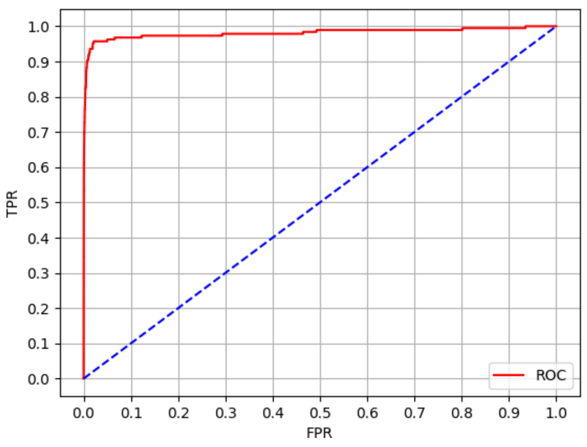
3 Results

Given the inexperience, the difficulty of the task and the (inadequate) hardware at hand, we think to have reached positive results, the system shows signs of being able to recognize traits of the mobs we've trained it on, even though sometimes they are just cases of pareidolia.

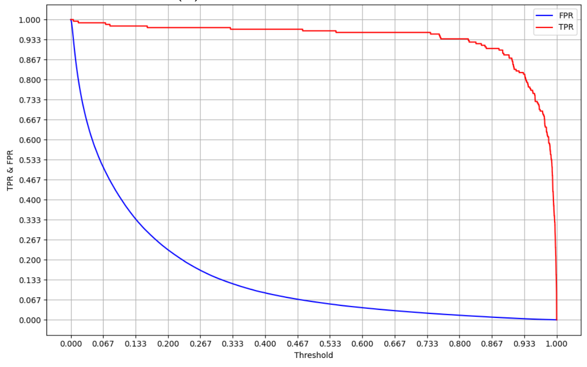
Decision threshold: Before giving some manner of statistics over its capabilities we would like to point out an important decision: that of the threshold for deciding whether, given a score, the anchor for which it is related to is actually a positive one or not. To decide this fundamental hyperparameter we resolved in sampling five hundred (500) images from our training dataset and, after letting the system apply non-maximum suppression, recovering the scores for all the remaining anchors and pairing them to their true labels. Once this pre-processing was done we plotted the ROC together with another graph, showing the decrease of the *true positive ratio* (TPR) and that of the *false positive ratio* (FPR) as the threshold increased (Figure 3)

From both these graphs we notice our network is indeed able to separate positive and negative anchors with very clear-cut definition. Furthermore, we decided that a good compromise between (TPR) and (FPR) could be achieved by choosing a threshold between 0.8 and 0.9: since we didn't want to excessively remove positive anchors, we put ours at 0.81. In order to avoid false positives as much as possible, 0.9 should work fine too.

Test dataset and stats: After this, in our opinion, fundamental decision was made: we resolved to open Pandora's box and create a test set of around 250 images with multiple mobs in the same shot. Unfortunately, as said previously, we didn't develop our



(a) Linear RPN's ROC



(b) Linear RPN's TPR & FPR as threshold changes

Figure 3: Here we show our RPN's ROC (a) and True Positive/False Positive Ratio (b) as we change the threshold, based on (b) we determined a good threshold value to be set around 0.81 by elbow method.

network up to classification of the boxes, and thus a full confusion matrix of those is out of the question. At any rate we show the misclassification table for the anchors (after non-maximum-suppression) scored by our network over the whole test set (threshold= 0.81):

Table 3: Misclassification matrix

	Label pos.	Label neg.
Pred. pos.	62	3545
Pred. neg.	17	127789

Telling us that, indeed, our threshold works, since the TPR is 0.78 and the FPR is 0.03. We would like to point out that, since the labelling of the anchors is itself hyperparameter driven (given the choice of anchors and that of the IoU thresholds used to label them) and that we think to have chosen a combination of these parameters that biases the labels towards the non-object side and, furthermore, given this label by itself already dominates the distribution: it is only normal for the number of positive labels to be so low in the set. If we look at the actual images with the positive region proposals added, we'll appreciate much more positive-looking (that may not actually be labelled as positive) proposals than the anchor labelling would actually suggest.

Going fully convolutional: As said above, we also tried fitting a fully convolutional version of the RPN, unfortunately though the results were pretty mediocre, its ROC pretty clearly showing it:

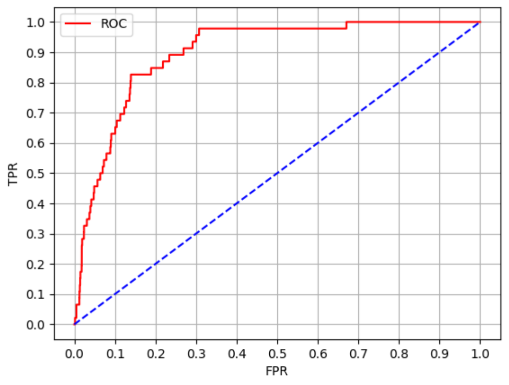


Figure 4: Fully convolutional RPN's ROC

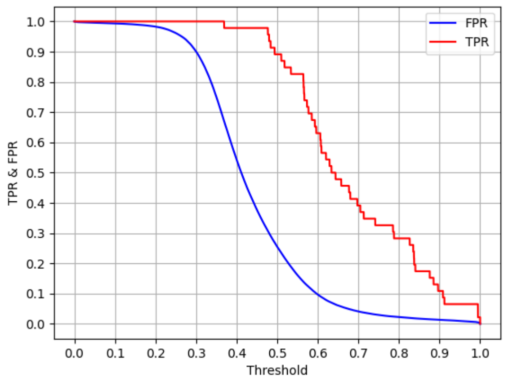


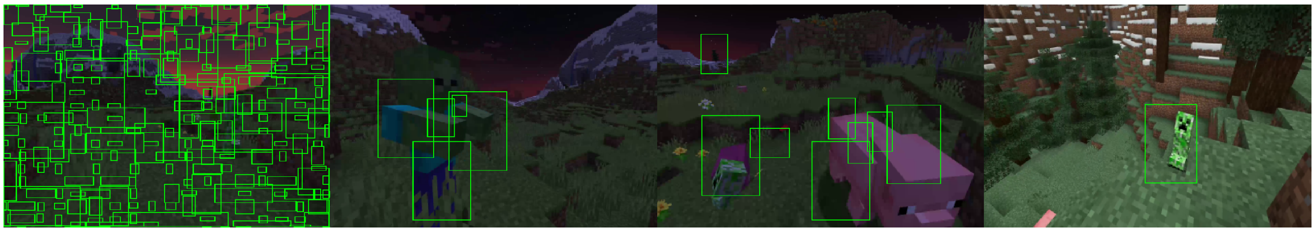
Figure 5: Fully convolutional RPN's TPR & FPR

Indeed this version is unable to sufficiently separate positive and negative samples, leading to unacceptable predictions. . . a shame, since it would've been several magnitudes lighter (weighing just 3.5 mbs) and faster (both to execute and to train) than its linear counterpart.

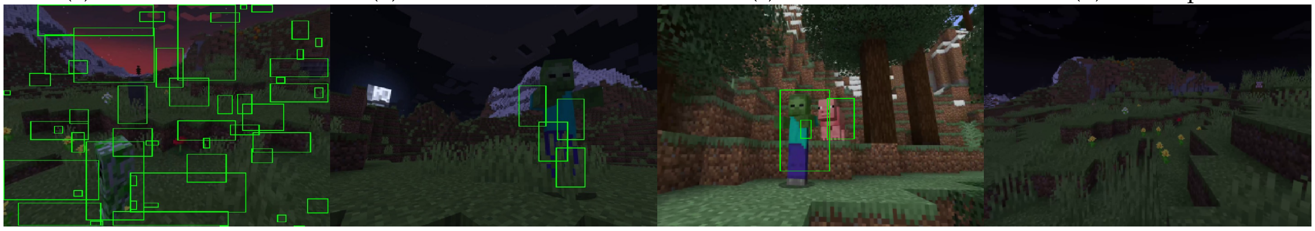
Looking at results: Let's now take a look at some particular behaviours formulated by our network so we can discuss both of its successes and failures (Figure 6 and 7).

Final Conclusions: In light of these agreeable and interesting results, we think to have reached our goals, not without hiccups nor errors that is. But the network seems to be, generally, well behaved and relatively correct whenever it is not presented with unexpected data.

Some of our Results



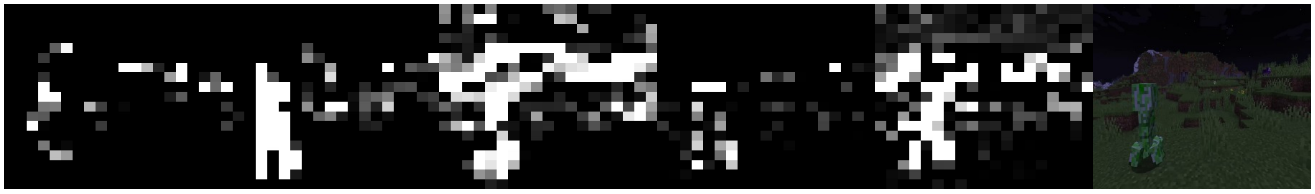
(a) A Weird Sunset (b) A *Piecewise* Zombie (c) Pareidolia (d) A Creeper



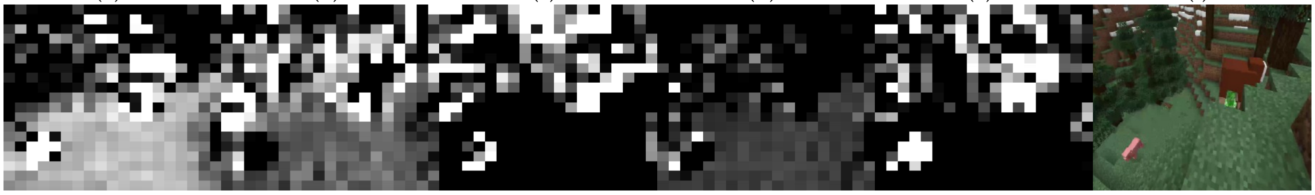
(e) Another Weird Sunset (f) Another Sundered Zombie (g) A Zombie and a Pig (h) Spot the Sheep!

Figure 6: Let us look and comment our results: First of all, we notice a very weird behaviour in output (a), (e), this behaviour is consistent to every image that is taken at that particular time of day, therefore, we speculate that the weird hue of the sky throws our model's predictions completely off, the reason is probably that in our limited dataset we captured an insufficient number of sunsets and therefore this comes of as a completely unexpected input to our network, which responds with this hysterical placing of anchors.
Going further, we can see some pretty promising results in images (b), (f), (g), (c) and especially (d), in all of those the objects are almost always recognized by multiple rectangles at once, in our opinion this is due to the fact that, due to how the problem is mathematically formulated, an anchor does not have to cover an object in its entirety to be labelled as positive. Even though the 12 possible anchors may *theoretically* regress to any size, they do not generalize all possible boxes due to IoU cutoff, making their spreading over a single object a desirable goal, since ensembles can generalize much better.
At last, we can notice a number of misclassifications in (c), (d) and especially (h), where the network is unable to spot the sheep, perhaps due to the low brightness and the fact that the sheep's green color blends in with the surrounding grass, we can also notice the occurrence of *Pareidolia*, where the network mistakes a pine in the background in image (c) for a target object and incorrectly classifies it.

Some Feature Maps



(a) (b) (c) (d) (e) (f)



(g) (h) (i) (j) (k) (l)

Figure 7: We now look at five feature maps that we randomly chose from the *Ziggurat*'s sixth output layer, as we can see, in the first row of filters, we get some clear activations around the area of the Creeper, especially in (b), (c), and (d). Whereas, if we look at our second row, we get reasonable activations only in the case of (i) and (k); (h) and (j) seem to have much more timid activations that blend the two animals' contours with the surrounding grass.
We do not draw any conclusions from such observations since trying to determine the inner workings of a Deep Neural Network is a fool's errand, nonetheless we thought it would be interesting (and, we won't lie, aesthetically pleasing) to have a look at the inner workings of its brain.

References

[1] R. B. Girshick, "Fast R-CNN," *CoRR*, vol. abs/1504.08083, 2015. [Online]. Available: http://arxiv.org/abs/1504.08083

[2] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," *CoRR*, vol. abs/1506.01497, 2015. [Online]. Available: http://arxiv.org/abs/1506.01497

[3] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004. [Online]. Available: http://dx.doi.org/10.1023/B:VISI.0000029664.99615.94

[4] H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool, "Speeded-up robust features (surf)," *Computer Vision and Image Understanding*, vol. 110, no. 3, pp. 346 – 359, 2008, similarity Matching in Computer Vision and Multimedia. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1077314207001555

[5] Q. Chen, J. Xu, and V. Koltun, "Fast image processing with fully-convolutional networks," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[6] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014. [Online]. Available: https://arxiv.org/abs/1409.1556

[7] A. Neubeck and L. Van Gool, "Efficient non-maximum suppression," in *18th International Conference on Pattern Recognition (ICPR'06)*, vol. 3. IEEE, 2006, pp. 850–855.

[8] H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid, and S. Savarese, "Generalized intersection over union: A metric and a loss for bounding box regression," 2019. [Online]. Available: https://arxiv.org/abs/1902.09630

[9] Mojang, "Minecraft," https://www.minecraft.net/en-us, Nov. 2011.

[10] G. Chechik, V. Sharma, U. Shalit, and S. Bengio, "Large scale online learning of image similarity through ranking," *J. Mach. Learn. Res.*, vol. 11, p. 1109–1135, mar 2010.

[11] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.