Logo Detector: Logo Recognition using YOLO

Kai Huang
1469670
kaihuang@ucsc.edu
University of California, Santa Cruz
Santa Cruz, California

Rishita Roy
1487369
riroy@ucsc.edu
University of California, Santa Cruz
Santa Cruz, California

Justin Yu
1456898
jubyu@ucsc.edu
University of California, Santa Cruz
Santa Cruz, California

Figure 1: Blurring image from logo detection using our finalized model

ABSTRACT

The basis of our project is to classify company logos and blur them from an input image. Given an input image, containing different background objects and noise, the machine learning model should locate the potential regions within the image that may contain logos, along with the potential companies these logos may belong to.

This report will talk about how we used a data set of 47 classes, which contained 32 classes and 15 text classes of logos of certain companies, and trained a model to detect these logos in an image and blurred them.

We will discuss different approaches we used to train this model, and why we did certain tests. We will then talk about which approach we used and created a custom model based off of the research we have done to train a model that worked best for our project.

We will show results of our findings, and compare them to decide which approach we want to use in our final product. We will then use benefits from the approaches we used to optimize our final model.

After all of this, we will talk about future plans for this project and how it can be beneficial to everyday use in the real world,

1 MOTIVATION AND OBJECTIVE

When we were deciding on what kind of project we wanted to do, we were motivated by several different factors. The first big one was that we were all fairly interested in doing image processing, as that was an area of machine learning that all three of us had not yet delved into. During the same time, there was big buzz in the news about copyright law and how it has been affecting the internet landscape since virtually anybody can file a copyright claim on

products, videos, and much more. Logos in particular are crucial to a company's identity, and there are significant rules governing where and when logos can be used. This lead to an intersection of interests, where we thought we could create a machine learning model that could potentially identify logos and help remove them automatically without needing human interference, like shown in Figure 1.

While doing research into how to approach our idea, we came across several projects and papers where others proposed their own solution into logo recognition. *DeepLogo: Hitting Logo Recognition with the Deep Neural Network Hammer* is a paper published by the ASPIRE Laboratory at the University of California, Berkeley [3]; it focuses on the usage of logos in advertisements and marketing materials, but still had logo detection at its core. This research also introduced us to the differences between recognition, detection (with and without localization), and localization by itself. This helped us realize that while localization would be sufficient for just removal of the logo, we were also interested in seeing if we could build a network that could identify the companies as well. Thus we were determined to build a model that could detect with localization.

This paper also introduced us to Faster Regions with Convolutional Networks (Faster R-CNN) as of the different kinds of neural networks we could potentially use in our solution. From there we researched more methods that we could use in our implementation, which included R-CNN, Fast R-CNN, YOLO and its variations, and others. We found several existing projects that utilize custom R-CNN models, such as (another) DeepLogo project [10], but we were interested in seeing if we could build a model that was faster than the sliding window approach.

Ultimately we used our own custom network with the YOLO infrastructure to produce fast but accurate results. There are definitely other projects that utilize this approach, but we believe our custom network provides reliable results for our logo detection task.

2 DATASET

The main dataset we used for this project was the FlickrLogos-47 set [9]. This was the final of three datasets we used throughout the course of our project, with the other two being the FlickrLogos-27 [4] and FlickrLogos-32 [9]. All of these sourced images from Flickr's photo gallery and include various images of logo instances in different circumstances; the number appending the each set's title refers to the number of classes available in each set. FlickrLogos-27 is maintained by the National Technical University of Athens and is publicly available to download from their website. The other two sets, FlickrLogos-32 and FlickrLogos-47 are maintained by Multimedia Computing and Computer Vision Lab, Augsburg University, and are only available after personally emailing to request access to the datasets. While they are all similar in terms of what they offer, University of Athens and Augsburg University both disclaim that the sets that they provide will not yield comparable results as there is little overlap between images as well as the specific logo brands available for training and classification. While we were waiting for permission to utilize the FlickrLogos-32 and FlickrLogos-47 datasets, we began preliminary testing and development using FlickrLogos-27. Each of the datasets comes with two main aspects: images and annotation files. We'll discuss more about the contents in the followng sections.

2.1 Preprocessing FlickrLogos-32

Once we acquired access to FlickrLogos-32 and FlickrLogos-47, we decided first to focus on FlickrLogos-32. Both sets have the same 32 logo classes, with FlickrLogos-47 adding 15 additional text based classes to use in conjunction with their matching symbol classes. We were provided bounding boxes of the logos, which is shown in Figure 2. Since there were fewer overall classes, the size of the dataset was considerably smaller, leading to reduced training times and thus allowing us to test out different approaches in a shorter amount of time.

An unexpected side effect to using the smaller dataset was missing or unclear logo positions along with a file naming convention which made it difficult to set up training environments. While we could not solve the issue of inconsistent logo positions, we wrote scripts to rename images to include class names which slightly helped with our annotation problem. FlickrLogos-32 also had a fewer number of overall images, so we increased the dataset by augmenting the preexisting images. This was done by generating random numbers to use as transformation values for x and y shifts and angle rotations, which would then get passed into the scikit-image transform library [1].

Figure 2: Example image and corresponding annotation mask from FlickrLogos-32 and FlickrLogos-47 datasets

2.2 Utilizing FlickrLogos-47

While we ultimately did not use the FlickrLogos-32 dataset for our final model, some of the scripts that we had created became useful FlickrLogos-47 was simply an expanded version with additional classes. Creating our file structure was easy with the same script, and the creators of the dataset solved our missing annotation problem by confirming that FlickrLogos-47 "is built from the same images than the FlickrLogos-32 dataset which have been re-annotated to fix mission annotations and to include more classes."

As mentioned above, the dataset comes with images and corresponding annotation files. For FlickrLogos-47, this meant that each image has a binary mask in form of a one channel PNG image that marks the bounding box, as can be seen in figure 2. In addition to the

image mask, each image also has additional information in the form of the coordinates of each annotation present. The bounding box of the annotation was crucial to our project as we wanted to not only detect the class in the picture, but also the bounding box of the logo.

The images available fall into three disjoint categories: training, validation, and testing. There are 10 images per class available for training, 30 per class for validation, and another 30 for testing. In addition to the 3150 class specific images, there are an addition 3000 images for each validation and testing that contain no logos. This comes out to 9150 images total available to us to use. We were initially concerned by the low number of images for training, which is why we created a script to augment these logo pictures. After reading the accompanying paper to FlickrLogos-32 and FlickrLogos-47 [9] it was clear that this low number was intentional. The training images were all hand selected to show various different angles of the logos. The paper also mentions that the validity of a logo classifier is also heavily dependent on ensuring it isn't falsely detecting the presence of a logo, hence the very large non-logo datasets.

3 MODELS AND ALGORITHMS

Our project had two big phases. Step one was researching various object detection methods and seeing how we could use these preexisting networks to our advantage in solving the logo detection problem. The second phase was reflecting on the results of our experiments from our initial trials and using that knowledge to create our final model.

Additionally, we did mention that our final goal was to both detect logos with localization and then to remove them from the given input. We made the decision to devote majority of our time towards developing our deep learning model. Our initial thought was to approach removing the logo in a content aware fashion, similar to how Adobe Photoshop does using its "Content Aware Fill" feature, but soon realized that this in itself could be another deep learning problem [1]. We instead opted to blur out the portion of the image using the coordinates of the predicted logo in the image. This task was much simpler, as we could accomplish this using libraries such as OpenCV or Python Imaging Library.

3.1 Preliminary Models and Trials

When we were doing research for our project, we were looking at common and well known examples in the realm of image processing as starting points for our project. It became clear to us that we were most likely going to be using deep learning, specifically utilizing convolutional neural networks to approach our task of logo classification.

We came across three main image processing networks that piqued our interest: LeNet-5 [5], VGGNet (Figure 3), and ResNet. These were all appealing to us because they approached similar problems using vastly different networks, and each one had their own strengths and weaknesses. We decided to try out these networks

[1] Adobe solves this using the PatchMatch algorithm, which is actually based a nearest neighbors implementation. Nvidia has recently been showcasing their own competitor to this, called inpainting, which solves the same problem using deep learning. More about these two approaches can be read about here [2] and here [6].

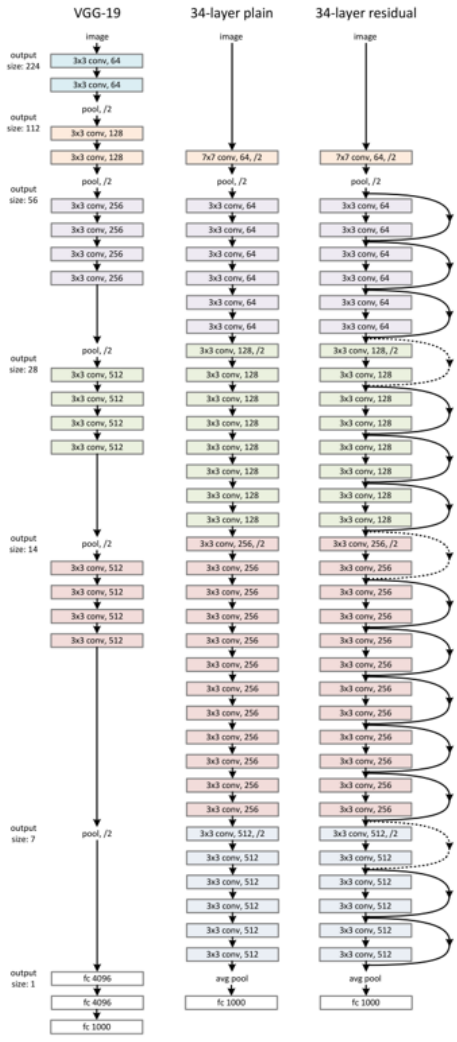

Figure 3: The first architecture is for VGGNet, and the third one is ResNet with Residual layers

using various different techniques to apply our own dataset into these preexisting models. We were unsure at the time how to construct an architecture that would be tailored to logo detection, and this step was important in discovering our final model.

3.1.1 Using LeNet. One of our first CNN models was based off of the LeNet-5 architecture, which was an original convolutional neural network architecture used for handwritten and machine-printed character recognition [5]. This architecture was very straightforward to learn and apply to our project's purpose, which shares its main purpose of image feature detections. LeNet-5 uses two sets of convolutional and average pooling layers, flattening layer, two fully-connected layers, and then a softmax classifier.

Layer 1: convolutional layer with 6 filters of 5x5
Layer 2: average pooling with 2x2

Layer 3: convolutional layer with 16 filters of 5x5
Layer 4: average pooling with 2x2
Layer 5: fully-connected layer with 120 feature maps, connected to 400 nodes
Layer 6: fully-connected layer with 84 units
Layer 7: softmax classification layer

In order to use this architecture, we needed to scale it up to handle higher resolution pictures. So just as LeNet-5 recognized different features from characters to classify them, we were recognizing different features that make up a certain company's logo. Using this model, after all the convolution, pooling, and flattening layers, we could find and locate different features that make up a certain company logo, and then mapped the existence of these features into the input image to a probability of it being a certain company logo. We then used a loss function of Cross-Entropy and Stochastic Gradient Descent as our optimizer for the model.

The implementation based off of LeNet-5 was done using PyTorch. We then used transfer learning to apply our dataset to their pre-trained model, and only modified the output softmax layer to fit our existing labels.

3.1.2 Using ResNet. We also decided to try a different Python library to create our logo detection model. ImageAI provides several "different Machine Learning algorithms trained on the ImageNet-1000 dataset. ImageAI also supports object detection, video detection and object tracking using RetinaNet, YOLOv3 and TinyYOLOv3 trained on COCO dataset" [7]. While it mainly focuses on object recognition, you can also use ImageAI to learn your own dataset onto these networks. We decided to use this approach to train a model using Microsoft's ResNet for its accessibility and ease of use.

ResNet's architecture, as can be seen all the way to the right in figure 3 is based on deep residual learning. This appealed to us as an option since it won many awards for its strength in object detection using its deep network with residual layers. Since we had much fewer classes, we were hoping that this could learn the same amount of detail and grant us with high accuracy for logo detection.

3.1.3 Using VGGNet. Using similar logic to the architectures listed above, we implemented our own version of VGGNet using Keras, and we were also able to repurpose the CNN architecture of VGGNet for logo detection. After some parsing and additional labeling of FlickrLogos-32, our directories of training, testing, and validation image datasets could be used.

3.2 Final Algorithm

From our preliminary models, it would take around 1-2 seconds to get an output prediction for any image. Originally, our working solution was to utilize a sliding window approach by feeding in multiple regions of interest into the model. However, since there will be thousands of regions, it will too inefficient, so identification and classification of logos would take too long. An attempt at a

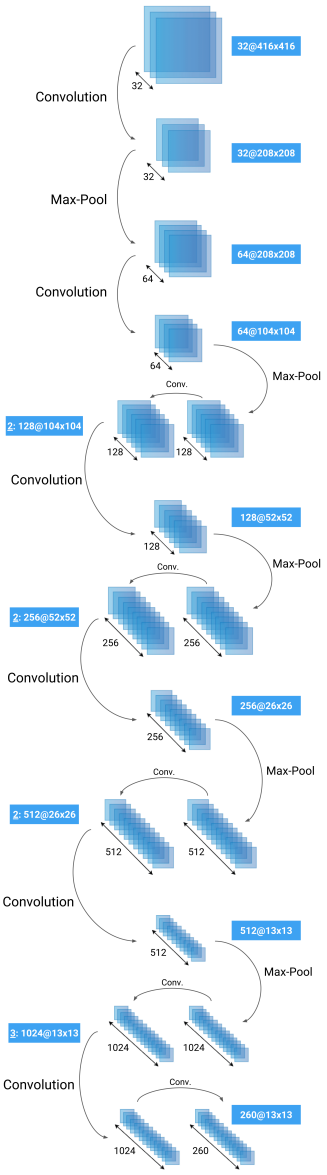

Figure 4: Custom model we created derived from Darknet and VGG convolutional neural network models.

solution, we looked into YOLO (You Only Look Once), a real-time object detection system. When training, there's not too much different between YOLO and other deep learning CNN architectures, however during testing, YOLO will look at the entire image using a single network, which makes it much faster compared to other similar architectures, such as R-CNN and Fast R-CNN [8].

In order to better understand how YOLO worked, we explored the structure of darknet19, which is the CNN architecture used by YOLOv2. So for our purpose of logo detection, we came up with the following two possible approaches utilizing YOLO/darknet19.

Using a model of darknet19 pre-trained on ImageNet (1000 object classes), we can use the features that it has learned from ImageNet objects to further help us identify logos. This is known as transfer learning, and we are freezing the entire network but removing the last fully-connected layer, and then repurpose darknet19 for logo detection.

Create a custom CNN architecture optimizing darknet19. Since darknet19 on YOLOv2 was made for object detection using the ImageNet dataset, it has a deep and wide architecture that has too many unnecessary layers for logo detection. As a solution, we took elements from VGGnet and darknet19 in order to create a lightweight and fast CNN architecture (shown in Figure 4) for the purpose of logo detection.

As stated above, when designing our own network architecture, we based our new design off VGGNet [11] and darknet19 [8], focusing on key features:
Deep Network: Consecutive convolutional layers with filters of 3x3 simulates a larger filter and allows less parameters. Growing depth, but shrinking spatial size, meaning our architecture will be fast and effective, and learn more with less parameters.
Activation: Leaky ReLU activation function is used to fix dying ReLU from large gradients.
Batch Normalization: each convolutional layer is normalized, allowing for independent learning in layers and reduced over-fitting.
Detailed Features: Output feature map dimensions of 13x13, which will match darknet19, so we won't lose quality.
High Resolution Classifier: Training with 416x416 and testing with 832x832 for small logos and objects within the image.
Adaptive Learning Rate: learning rate adjusts itself after 10000 epochs, or whenever loss rate changes.
Multi-Scale Training: trained model using darknet implementation, so input image size changes every 10 batches, varies 320x320 to 608x608.

Evaluation Metric for YOLO-based architectures is based on:

- Intersection over Union (IoU): intersection of actual object bounding box and the detected prediction of object bounding box over the detected prediction of object bounding box.
- Mean Average precision (mAP): intersection of the two bounding boxes over the union of the two object bounding boxes.

Table 1: Testing Accuracies of Preliminary Models

Model	Testing Accuracy
VGG-19	71.57%
LeNet-5	65.83%
ResNet	69.71%

4 RESULTS AND ANALYSIS

4.1 Preliminary Results

The preliminary training phase of this project can be be thought of as our own training phase. There were many different approaches we took, and while we did not end up using any of these finally, they taught us several key things that we ended up using later in our project.

Table 2: Custom Model vs Darknet Performance

	IoU (best)	mAP (best)
Darknet19 (8000 epochs)	46.53%	56.28%
Custom (9900 epochs)	**71.10%**	**77.26%**

One big thing we figured out in this phase is that these models that we were using (ResNet, LeNet) had fairly decent accuracy values, as can be seen in table 1, but unfortunately did not give us a bounding box. Removal of the logo from our input was a major part of our project, and this is when we realized that logo recognition simply was not enough. We had to look into detection and localization algorithms that would solve this portion of the problem. It was at this phase that we started doing research into differences between just CNNs, Regional CNNs (R-CNN), the fast variants on R-CNN, and YOLO. It was this research that pointed us towards YOLO as a viable option for our project, and this is ultimately what we ended up using as well.

We also realized at this point that using online tools such as Colab was difficult to manage with the magnitude of our data. The initial Python notebooks we had were often finicky, and often times the hosted run time would stop without given any reason. This became increasingly problematic as we would be training with epochs that took around two hours long, and suddenly all of our work would disappear due to unknown and unforeseen errors. This lead to a lot of repeated effort, and ultimately made us realize that we would have to perform these computationally expensive trials locally.

4.2 Final Results

At first we started with Darknet19's CNN to train our model for the Flickr Logo set. The problem is that this CNN trains too long, and it was a general training model that we thought was not too geared towards what we wanted in our model. We then also tested VGG training model which was also taking a long time. We decided to make our own custom model that was based off of Darknet19 and VGG but changed some attributes in some of these layers in order to train on things that we felt were more important and removed layers that we felt were unnecessary. We chose these different CNNs to modify for various reasons. We wanted to use Darknet because it allowed us to train for a bounding box, and we also wanted to use VGG because it had the highest accuracy compared to the other architectures we compared to in out preliminary results.

When designing our custom CNN, we created a larger filter so we reduced the amount of parameters, because one original concern we had was the training time it took to train our model. We used a Leaky Rectified Linear Unit because when gradient gets really high we still want to learn something, and if we use normal Rectified Linear Unit we will not learn anything during training

when the gradient is in the negative part. We also use batch normalization so that independent layers can continue to learn while reducing overfitting at the same time. We furthered this by changing the learning rates after every 10,000 epochs or whenever loss rate changes. These methods resulted in feature maps of 13x13 which matches with darknet19. While training we tried iterating through many different different image inputs such as training on 416x416 images and testing on 832x832 images, and changing sizes of input to 320x320 or even 608x608 sized images. With all these things in mind, we made some comparable differences with the darknet19 training model versus our custom model.

Training darknet19 and our custom CNN architecture took an immense amount of time. We trained our models in batches of 64 in 8 mini-batches. This allowed us to efficiently train 64 images every epoch. A single epoch on the MacBook Pro's Intel i5 processor would take around 2 hours. When training on a Nvidia Jetson TX2, each epoch took 4 seconds. This allowed us to train each model for 20,000 epochs, so we can observe the early stopping point and the weights that gave us the best accuracies. YOLO's implementation allowed us to save our weight files every 500 epochs, so we just let it continually train overnight so we can scrap the accuracy in the morning using a script.

We have significant results that show our model works better with our dataset above with a little less than 10,000 epochs. Note that our CNN actually changes learning rate every 10000 epochs, which begs the question why we did not check our IoU and mAP after that. We actually tested it, and we gave the result at 9,900 epochs because that is where out IoU and mAP converges, so after that many epochs not much more learning is gained, even though the loss was still decreasing at a measly rate. We trained up to 20,000 epochs and the accuracy peaked at epoch 9,900. We experimented with running different learning rates our accuracy never got any better. So for our custom CNN architecture on FlickrLogos-47, our model's loss converged at around 11,000 epochs, and our observed early-stopping point was around 9,900.

5 CONTRIBUTION

Kai Huang worked on training and testing on VGG as well as getting the data sets we needed by contacting the owner of the Flickr Dataset. Worked on training Darknet CNN and making custom CNN to train for testing. Did most research and provided solutions to many different approaches when doing this project, and tested them each in order to figure out what was the model we wanted to train on the most.

Rishita Roy worked on training images on Resnet and ImageAI. Trained previous models for image recognition for certain objects in an image so team can understand concepts and strategies we should focus on when we start the project. Created diagrams for our custom CNN, and created script to blur an image given a bounding box which was outputted from our model of a test image.

Justin Yu worked on training vanilla YOLO using Darknet on the COCO dataset, and then changed dataset with our own which required a script to parse images so that Darknet can input the images and train the model. Researched different algorithms we could use, and worked and touched up on team reports as well as poster.

6 FUTURE WORK

As of now we can only blur images. We would like to extend this tool into video recordings. This software has a lot of promising capabilities for a lot of people who want to edit videos and take pictures. Some future implementations are to apply this software in real time so that when users before they take a picture or video record can see what to expect. This is possible with YOLO's algorithm that only checked the image, or each frame of a video, once. We can process the images fast enough so that we allow users to see everything that is supposed to be blurred right away.

We can also have an option where users can submit an image or video and the model would change everything that way as well, and allow users to edit what they want. We can allow these two different modes for users based on preference.

Since our main motivation was to look out for copyright purposes, we can also look into blurring people's faces so. There seems to be a lot of resources that already exist today about blurring people's faces in real time, but we would also want this to be done with YOLO. We would have to train for a face as a class in the training model.

Face recognition with YOLO exists already, and we may consider not using them as a class in our own created training model because we do not want to ruin the training for our logos. We would probably approach this by running YOLO twice, one for logos and one for faces, but that will be a project for another time!

REFERENCES

[1] [n. d.]. Module: Transform. http://scikit-image.org/docs/dev/api/skimage.transform.html#module-skimage.transform
[2] [n. d.]. PatchMatch. https://research.adobe.com/project/patchmatch/
[3] Forrest N. Iandola, Anting Shen, Peter Gao, and Kurt Keutzer. 2015. DeepLogo: Hitting Logo Recognition with the Deep Neural Network Hammer. *CoRR* abs/1510.02131 (2015). arXiv:1510.02131 http://arxiv.org/abs/1510.02131
[4] Y. Kalantidis, LG. Pueyo, M. Trevisiol, R. van Zwol, and Y. Avrithis. 2011. Scalable Triangulation-based Logo Recognition. In *in Proceedings of ACM International Conference on Multimedia Retrieval (ICMR 2011)*. Trento, Italy.
[5] Yann Lecun, LÃĺon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*. 2278–2324.
[6] Guilin Liu, Fitsum A. Reda, Kevin J. Shih, Ting-Chun Wang, Andrew Tao, and Bryan Catanzaro. 2018. Image Inpainting for Irregular Holes Using Partial Convolutions. *CoRR* abs/1804.07723 (2018). arXiv:1804.07723 http://arxiv.org/abs/1804.07723
[7] OlafenwaMoses. 2019. ImageAI. https://github.com/OlafenwaMoses/ImageAI
[8] Joseph Redmon and Ali Farhadi. 2016. YOLO9000: Better, Faster, Stronger. *CoRR* abs/1612.08242 (2016). arXiv:1612.08242 http://arxiv.org/abs/1612.08242
[9] Stefan Romberg, Lluis Garcia Pueyo, Rainer Lienhart, and Roelof van Zwol. 2011. Scalable logo recognition in real-world images. In *Proceedings of the 1st ACM International Conference on Multimedia Retrieval (ICMR '11)*. ACM, New York, NY, USA, Article 25, 8 pages. https://doi.org/10.1145/1991996.1992021
[10] Satojkovic. 2018. satojkovic/DeepLogo. https://github.com/satojkovic/DeepLogo
[11] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014). arXiv:1409.1556 http://arxiv.org/abs/1409.1556