Labeling Paths with Convolutional Neural Networks

Kyle Wuerch and Sean Wallace

***Abstract*— With the increasing development of autonomous vehicles, being able to detect driveable paths in arbitrary environments has become a prevalent problem in multiple industries. This project explores a technique which utilizes a discretized output map that is used to color an image based on the confidence that each block is a driveable path. This was done using a generalized convolutional neural network that was trained on a set of 3000 images taken from the perspective of a robot along with matching masks marking which portion of the image was a driveable path. The techniques used allowed for a labeling accuracy of over 95%.**

I. INTRODUCTION

The goal of this project was to create a reliable method for predicting a path around the Cal Poly computer science building using a hand-labeled dataset distinguishing geometric boundaries. We created a graphical user interface that assisted in the process for hand-labeling data and used the dataset to generate a machine learning model by training a convolutional neural network.

A. *Problem Statement*

The goal of this senior project was to develop a convolutional neural network that could quickly and effectively label portions of an image as either driveable or not driveable. As a starting point, the project was provided with a set of 3000 labeled images collected around the Cal Poly Computer Science building.

II. DATA

One of the major components of this project was the data used for training and testing our neural network's ability to recognize driveable paths.

A. *Data Collection*

A set of over 3000 images were collected from the viewpoint of a robot by our advisor. An example image is shown in fig. 1. The data was collected using a small cart with an attached HDR camera and laptop. Images were collected around the Computer Science Building at Cal Poly in a variety of weather conditions.

After the images were collected, pixels in the image were labeled as being either driveable or not driveable. This was done by drawing polygon masks over the image to cover the path. The mask was saved as a set of points stored in a an XML file, which is visualized by fig. 2. This data formed the ground truth data for training of our neural network. This process was also completed by our advisor.

B. *Preprocessing*

The raw data went through multiple steps to simplify the process of training our neural network.

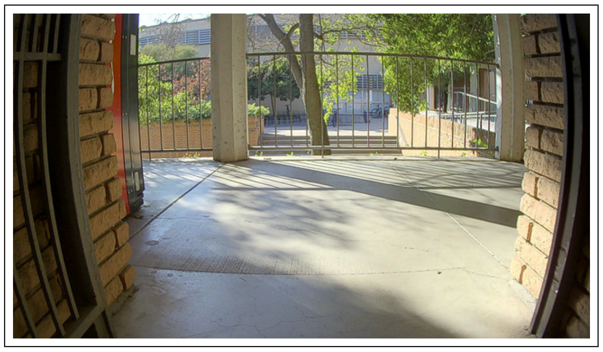
Fig. 1. Example image utilized in training set

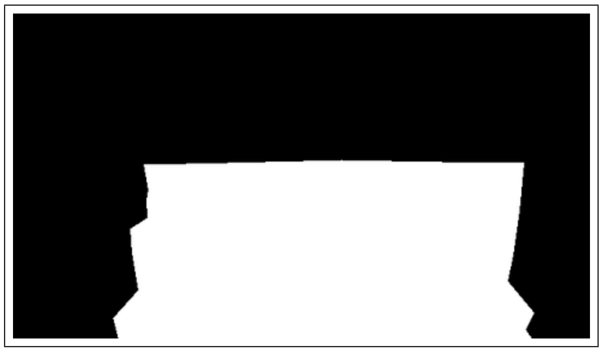
Fig. 2. Mask polygon utilized as ground truth data for fig. 1

1) XML to Masks: The raw XML files representing the ground truth for each individual image were parsed via a Python script, then drawn out to a PNG image using OpenCV. This allowed us to visualize the data, and it made it easy to augment the masks in the same fashion as the input images.

2) Image Size Conversion: The collected image set contained images of size 480x270 and 640x360. A program was written to convert all of the images to 480x270. These resized images were then used in the training process. This simplified the training process by not requiring the resizing of images during training.

3) Data Augmentation: The image set was augmented to a set of 10000 images. The augmentations used are shown in table I.

Additionally, the data was split into three sections: training, validation, and testing with percentages 80%, 10%, and 10% respectively. This split ensured that data in the training set were not utilized when verifying the accuracy of our model.

TABLE I
DATA AUGMENTATION TECHNIQUES APPLIED TO SET OF 3000 IMAGES

Image Modification	Arguments	Probability
Skew	None	10%
Rotate	Max Left Rotation = 10 Max Right Rotation = 10	20%
Zoom	Min Factor = 1.1 Max Factor = 1.2	20%
Left Right Flip	None	20%

III. MODEL

In our project, we solved the problem of path labeling by using a convolutional neural network (CNN). The CNN takes in an image as input and outputs a mask of ones and zeros that represent which part of the input image is driveable, and which part is driveable. The CNN is generic enough that the output size can be modified to be multiple scalar outputs of the original input image. For example, the CNN, with an input image of 480 x 270, can be trained to output a mask with shape 16 x 9, 32 x 18, 64 x 36, etc. The output space can go as high as the original input image resolution of 480 x 270. Most of the testing of this CNN was done at either an output space of 480 x 270 or of 160 x 90. The full model that we developed is included in table II. Additionally, the section of Python code describing the model is included in section VI-D.

TABLE II
MODEL USED FOR DRIVEABLE PATH LABELING

#	Layer Function	Argument	Value
1	MaxPool2D	input shape padding pool size	(270, 480) same 3
2	Conv2D	filters kernel size activation	8 5 relu
3	Dropout	percentage	10%
4	MaxPool2D	pool size padding	3 same
5	Conv2D	filters kernel size activation	8 5 relu
6	Dropout	percentage	25%
7	Flatten		
8	Dropout	percentage	25%
9	Dense	number outputs activation kernel initializer	512 relu glorot uniform
10	Dropout	percentage	50%
11	Dense	number outputs activation kernel initializer	256 relu glorot uniform
12	Dropout	percentage	50%
13	Dense	number outputs activation kernel initializer	128 relu glorot uniform
14	Dense	number outputs	factor of (270 x 480)

A. Development

When we were initially posed with the challenge of labeling the driveable portions of paths, we focused on determining the inputs and outputs to our solution. For us, based of the data set we were provided, the input was a 480 x 270 color image, but we were not sure on exactly what the output would be. In order to solve the problem, it would need to accurately represent what parts of the image were driveable, but it would also need to be discrete. We thus spent time discussing possible ways of doing this.

One of our initial thoughts was to use a spline that would encompass the area that was driveable. This would be possible since our dataset had contiguous driveable paths. The only challenge with this solution is that a spline does not have a discrete set of coefficients that describe any spline.

We then came up with the idea of splitting the image up into a set of equal sized subsections that could then be labeled as driveable or not driveable. Increasing the number of subsection would then allow us to increase or decrease the resolution of our output with little extra work. Since this idea would work for the input image and still have a discrete output, we moved forward with it.

We were already planning on using a CNN to take our input and convert it to the output space, and thus at this point, we needed to design the CNN such that it would accurately do so. Since this was our first time working with CNNs we used a Tensorflow tutorial with the MNIST data set a starting point for our neural network. As we continued working on modifying the tutorial to match our input and output we quickly found that Tensorflow had a much simpler library called Keras that greatly simplified the process for implementing CNNs. We then modified our code to utilize Keras.

Our initial model utilized two convolutional layers a hidden layer and a 16x9 output layer. When initially attempting to train the model on the dataset, we had trouble approaching any reasonable accuracy. Regardless of the number of epochs trained on, the loss and accuracy barely changed. This issue turned out to be an issue with the learning rate. A couple of attempts with different learning rates led us to settle at a learning rate of 1e-4. After training for 50 epochs, this initial model produced a training accuracy of 99.09% and a validation accuracy of 93/56%, but utilized 25,000,000 trainable parameters to achieve such results. From this point, be began working to improve the model.

B. Design Decisions

Ultimately, there were two factors driving the design decisions for the model used in this project: accuracy and efficiency.

1) Accuracy: In this project, accuracy was measured as the average of the difference from the confidence of each output of the neural network from the actual ground truth value for the output. Thus, a perfectly accurate model would predict the same output as our labeled ground truth. To drive a higher accuracy for the model, multiple techniques were used.

The first strategy used for improving accuracy was changing the data input that the neural network was trained on. This included a variety of augmentations to the dataset to produce a more general neural network. Using the augmentation listed in table I when compared to the unaugmented image set on an output space of 160x90 and training for 200 epochs led to a validation accuracy of 95.15% compared to a 94.81% validation accuracy. Additionally, this lowered the training accuracy from 96.15% to 95.44%, showing reduced effects of overfitting.

The second strategy used for improving accuracy was adding one dropout layer after each convolutional or dense layer. Using the dropout percentages listed in table II and augmented data as explained in table I with an output space of 160x90 had a validation accuracy of 95.15% and a training accuracy of 95.44% after 200 epochs. Using only one dropout layer of 40% before the last dense layer of the network led to a validation accuracy of only 94.81% but with a training accuracy of 97.24% after 200 epochs. While the validation accuracy was not significantly different, dropout appeared to minimize the effects of overfitting, since the training accuracy more closely matched the validation accuracy after 200 epochs for the model with more dropouot layers.

2) Efficiency: To achieve the maximum efficiency, the solution for this project needed to minimize the size of the model used for path labeling while still maintaining a high accuracy. The process of achieving this included multiple tests using various layer size and mostly consisted of a guess and check strategy where certain layers would be reduced or removed from our original model. The measurement used for model size in this project was total parameters, while the accuracy used was validation accuracy.

The most effective improvement came from decreasing the size of the dense layers used in the model. The first model used two convolutional layers and 512 neuron hidden layer before a 144 output layer. This relatively simple model for a 16x9 output space contained 25,261,648 parameters and only produced a validation accuracy 93.56% with significant overfitting. By halving the number of filters on the convolutional layers and splitting the hidden layer into two 128 neuron dense layers, the total number of parameters was decreased down to 350,064 trainable parameters with a validation accuracy of 91.27%.

As we desired to increase the output space of our neural network to 160x90 and 480x270, we needed to add more hidden layers to account for the additional complexity of problem. This was done by using three hidden dense layers of size 128, 256, and 512 placed after the convolutional layers and before the output layer. This ultimately increased the size of the model, but also allowed for higher accuracy when a large output space was used. With a 480x270 output space the model achieved a validation accuracy of 95.26%, and a validation accuracy of 95.12% for and output space of 160x90.

The final model as shown in table II had a variable number of trainable parameters depending on the specified output size. For a 160x90 output space, the model had 3,205,408 trainable parameters, and for 480x270, the model had 18,066,208.

An example of a successful run of the neural network with an output space of 480x270 is included in fig. 3.

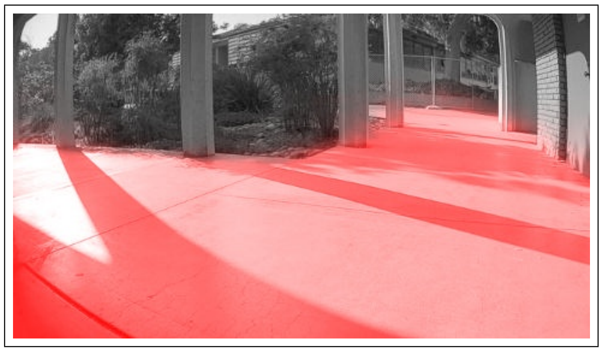

Fig. 3. A generated path mask for a given input image passed to the neural network with output space 480x270

C. Generality Testing

The dataset used for training the neural network in the project was significantly limited in that it only contained images from a specific part of campus. In order to test the effectiveness of our model in a more generalized environment, we took pictures at some different locations on campus and ran them through the same neural network. A semi-successful example of these tests can be seen in fig. 5 while an unsuccessful one is shown in fig. 4.

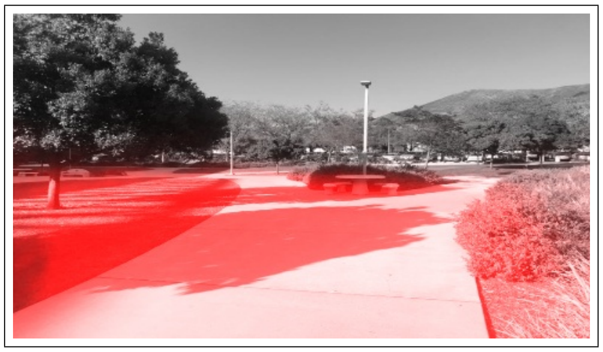

Fig. 4. Path mask applied to image significantly different from image set with signifcant error

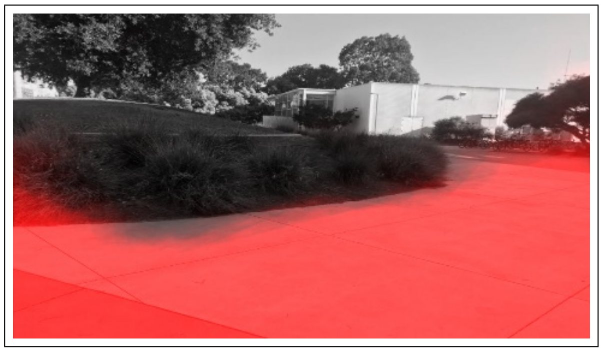

Fig. 5. Path mask effectively applied to image from outside of image set

From these images, we believe that the network correctly was able to identify objects that rise out of the image as being non-driveable, but has difficult when the objects are the same height as the driveable path. Thus, in order to make this network more effective, additional data containing flat non-driveable path next to a driveable path would need to be collected and used in the training of the network to improve the overall generality of this network.

IV. Graphical User Interface

The ability to visualize the output of our neural network turned out to extremely valuable to the development of our neural network. We made revisions to the model iteratively. Initially, we would make a code change and observe during training the metrics for loss and accuracy. While these heuristics were sufficient for determining whether a specific iteration was effective, these simple metrics didn't allow us to gain any further insight into the behavior of the network to understand why it labelled a certain portion of the image the way it did. We needed to visualize the output of our network in order to understand what kinds of images that our network had difficulty with, and the development of a lightweight graphical user interface allowed us to better understand the effectiveness of the network such that we could make better decisions in keeping or discarding revisions to our network.

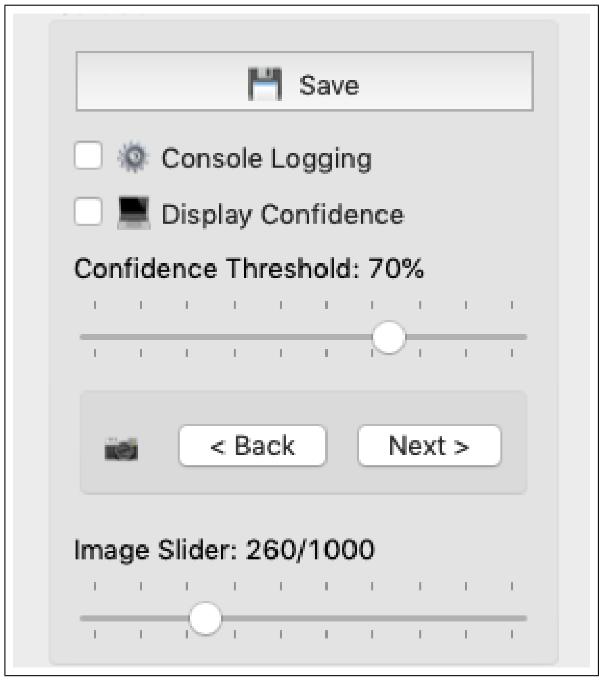

Fig. 6. An overview of our graphical user interface's control panel.

A. Grid-based interface

Initially, we wanted to simplify our neural network in order to both improve our understanding of its behavior as well as create a consistent baseline that we could improve from. Instead of working to classify parts of an image that could be a path from the get-go, we started small and in our first milestone we aimed to classify subsections of an image that were either red or black with a grid-like level of granularity.

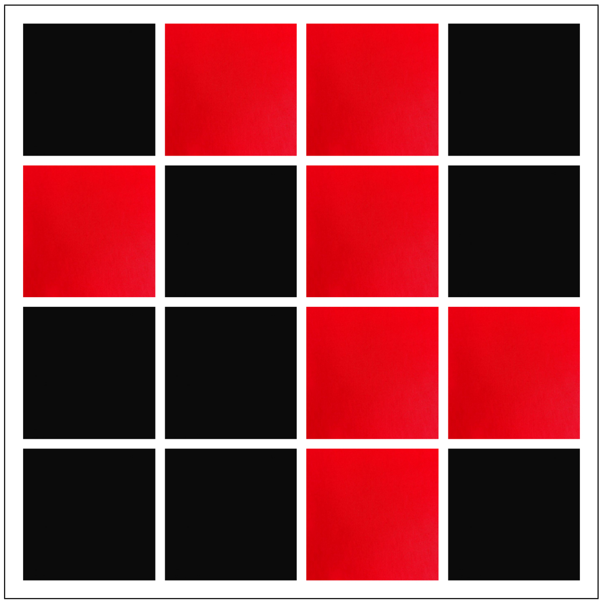

Fig. 7. An example of one of the earliest images used in our early neural network to distinguish between black and red in a grid-like fashion.

After we created red-black grid images such as the one shown in fig. 7, it was then necessary to hand label them to describe which areas of the grid would be considered red blocks. We developed our network until we could confirm that it could learn to identify a specific feature in an image and create a generalized model to output correct predictions in our desired grid-like granularity.

Once we were happy with the results, we abstracted our network to be relatively agnostic to input and output image size, making them have a variable sized input and output shape such that they could be easily configured. This allowed us to replace the black and red grid images with the actual images from the dataset we intended to use in the long term. As we increased the possible resolutions in our efforts for variable input and output sizes, the grid-based architecture still remained central to our application. For instance, a full-scale output space where the output size is the same as the input size was still represented internally with a 1x1 grid size.

B. Initial Revision

Since we developed our own GUI, we had the freedom to add new capabilities to aid in the development of our neural network as we deemed necessary. OpenCV functions were leveraged throughout the project to manipulate data input and output from the neural network, we chose to also leverage it for its basic UI capabilities. However, as our GUI became a little more complex we decided to migrate away from OpenCV entirely. The library wasn't made for creating production-ready interfaces. Instead, OpenCV's GUI capabilities are intended to just be used as a handy tool for quickly testing or visualizing OpenCV-related functions. Thus, some simple UI components would have needed to been made from scratch, and since the GUI was becoming more expensive to maintain as we tested new capabilities, continuing to use OpenCV would have distracted us too much from the actual development of the neural network.

C. *Final Revision*

Thus, we chose to rewrite the user interface without the use of OpenCV and instead chose to utilize Qt. At a high level, the entire interface is comprised of one major window component and smaller functionalities are split into subcomponents within the Qt architecture. We re-implemented all of the previous features that aided in both the development and visualization of our neural network. The neural network outputs values representing the confidence it has that each part of an image is a path with values between 0 and 1. The GUI's confidence threshold slider allows the user to display this confidence output dynamically, visualizing at what point the confidence drops off. This allowed us to better compare the accuracy between different models, it made it easy to understand which features in an image caused the confidence levels to drop off significantly from one image to another. Finally, we added the capability to save samples of the masked output in reference to the original image in order to document the evolution of our model as we made improvements.

V. CONCLUSIONS

A. *Viability*

With just a few thousand images, the neural network learned to identify paths around the Cal Poly computer science building with an accuracy over 95%. However, it is worth mentioning that the strength of the neural is in part limited by the quality of the dataset used to train it.

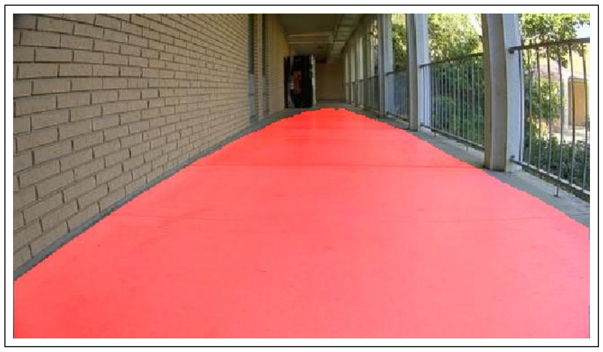

Fig. 8. An example of a straight path in which the neural network excels.

The neural network tended to excel in straight paths as shown in fig. 8. This is largely due to the fact that it is easier to obtain and hand-label images in this scenario in contrast to the difficulty of labeling images with people and accurately creating a polygon around their feet. Additionally, if a new image is significantly different from anything that the network has ever seen as shown earlier in fig. 4.

However, the network was able to effectively generalize a process for labeling paths in images that were similar to the ones it trained on. Consider the image shown in fig. 9.

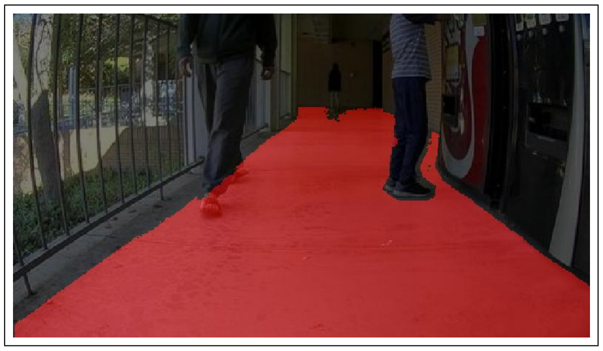

Fig. 9. An example of an output with features forming a curved path.

Although it lightly clipped the feet to the person walking on the left hand-side, it avoided the person on the right entirely. In the dataset used for training, similar images to this one were likely labeled such that the small segment to the person on the right were not included. Yet, the neural network was able to successfully correctly distinguish that portion as part of the path.

B. *Lessons Learned*

1) Tooling: In the context of our overall project timeline, a large portion of our time was spent developing tools for ourselves such that we could increase the speed at which we developed our neural network. Although we spent a lot of time up front to ensure a good development workflow, this allowed us to quickly make iterations to our model. Spending this time greatly improved both the quality of our project as well as the ease in developing it.

2) Data Collection: Our final result utilized the hand labelled dataset provided to us by our advisor. However, we did initially spend some time developing our own methodology for labeling images with the intention of aiding in our understanding of the end-to-end development of a convolutional neural network. In our first major revision for training our neural network, we created our own hand-labeled dataset using a 30x30 gridsize given images with a resolution of 480x270. Although we considered our GUI to be relatively well developed for the task of hand-labeling images in this grid-like fashion, it still took the two of us over 4 hours to hand-label just 1000 images.

3) Development Platform: We began development on our own laptops, but as the size of our model increased, the time spent training started to be long enough to slow down our iterative development cycles as we waited for training to complete. Some time was then spent experimenting with training on Cal Poly's high performance computer science servers. While this was an improvement over training on laptops, we eventually settled on developing in the Google Cloud Compute Platform. We used a virtual machine that utilized Nvidia's Tesla K80 video card for tensor computation, making it possible to train a hundred epochs in the time it would have taken to train a single epoch on our own laptops. This allowed us to determine the accuracy of new changes to our model much more quickly, and allowed for quick experimentation that would not have been possible otherwise.

C. *Extending the Project*

1) Real-time computation: After the model is generated, it is used on the fly to generate an output for each test image as it is requested from the graphical user interface. The model is fast enough to be used dynamically, and could feasibly input a video feed and display augmented video with the computed path in real time. This would increase the usability of the neural network in applications involving autonomous vehicles for dynamic pathfinding, and would also have the added benefit of reducing the risk of error in the network by spreading the decision making process in pathfinding across the result of multiple images.

2) Data Labeling Bootstrapper: The process for collecting then labeling data is extremely time consuming. Thousands of images were captured and polygons distinguishing the path were created manually for all of the images captured. However, it takes a surprisingly small amount of data (potentially less than 200 images) for the trained model to have an accuracy that beats random with few false positives. With this in mind, the network could feasibly be utilized in the creation of training data for either this application or even an entirely different application. Instead of creating entire polygons from scratch, the network could train simultaneously as more data is added to the dataset. Then, it could add suggestions for polygons that could be accepted or modified to speed up the data collection and labeling process. Especially since we have achieved an accuracy over 95%, this could be valuable as a tool whose primary purpose is to assist in labeling data.

3) Integrating Odometry: The dataset utilized in this project specified sections of each image that could be considered a path. Thus, the output of our neural network functions only as a classifier for determining drivable paths in an image. One way of improving upon this functionality is to add another data input layer into our neural network consisting of a robot's odometry data as it collects images while navigating the surrounding environment. The inclusion of this data would allow the neural network to provide estimations regarding its spatial position as well as its distance from parts of the image. This would greatly increase the usability of this neural network in a real world setting as it could then be used to not only determine directionality of a path, but provide insight into how the robot's actuators should behave in order to safely travel on the path.

D. *Related Applications*

The same convolutional neural network we developed could be used in a variety of applications involving binary classification without modification to the design of the network itself. Since our convolutional neural network is capable of learning features of images at a high level, it only requires a new labeled dataset in the same grid-based format to be used in another application.

In a completely different application, we have successfully applied our convolutional neural network without modification to the model itself to identify birds in images. Although

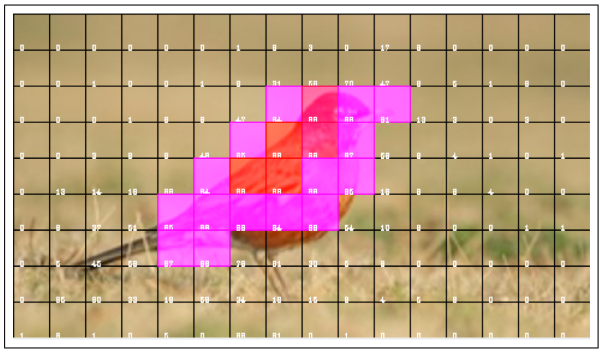
Fig. 10. The output of our convolutional neural network after being trained to recgonize birds in images.

the raw image dataset we used comprised of relatively high quality images taken by photographers, it was still able to distinguish the location of birds in images with an accuracy of over 90% for images it has never seen before. For the output shown here, only 200 images were utilized for training as well as an output space delimited by a 30x30 grid resolution. It should also be noted that not all of the images were successful in accuracy depicting the shape of the birds, but the neural network was still able to correctly detect at least some part of the bird without false positives.

VI. Appendix

A. *Prerequisites for Installation*

Our project was developed using Python 3.6 and has major dependencies on Tensorflow (1.1.12) for the development of the convolutional neural network and OpenCV(3.4.4) for image processing utilities in manipulating the raw input data as well as the raw output data from the neural network. Qt(5.11) was utilized for the lightweight graphical user interface for the creation of input data as well as the visualization of output data for testing or debugging the neural network.

B. *Training the Model*

The model was trained using a dataset collected around the Cal Poly computer science building. Images were captured and were labeled manually to describe the polygon bounding the path within each respective image. In the dataset, the polygons were described in an XML format. At a minimum, a dataset of images and data annotating which portions of the image are a path is necessary to train a new model. To increase the effectiveness of the network, images should be captured of as many different situations as possible. Although we have had some success in training the model for certain applications with just 200 images, the accuracy in our path labeling only started to become extremely effective after utilizing over 1000 images for training. Especially since we split up the data and only 10% ends up being used for testing, it makes sense to have enough data to be able determine the effectiveness of the convolutional neural network. Training can be executed by running the training script with the path of both the input and desired output directories as shown below:

```
python3 net/train.py
    --data_in my_data_input_directory
    --data_out my_data_output_directory
```

It is worth noting that the training script that is run is primarily a scaffolding script that stitches together and runs our neural network. Thus, high level modifications such as the number of epochs to train for or the desired the output shape should be configured in that script.

C. Testing the Model

The training process generates a new folder containing all of the necessary information to run the model, and it can then be used repeatedly on different input images for testing. The effectiveness of the model can be visualized by running the model against test images in our graphical user interface. To do so, simply execute the testing script with the path of the generated directory as an argument. An example command for running the test script is shown below:

```
python3 net/test.py logs/SNETZ-00033/
```

D. Code

The source code for this project is hosted on GitHub [1]. Comprehensive instructions for installing and running the model are included in the repository's README file. Included below is the code describing the model used for this project.

```
model = Sequential([
MaxPool2D(pool_size=3,
    padding='same', input_shape=shape),
Conv2D(filters=8, kernel_size=5,
    activation='relu'),

Dropout(0.10),
MaxPool2D(pool_size=3, padding='same'),
Conv2D(filters=16, kernel_size=5,
    activation='relu'),

Dropout(0.25),
MaxPool2D(pool_size=3, padding='same'),
Flatten(),

Dropout(0.25),
Dense(512, activation=tf.nn.relu,
    kernel_initializer='glorot_uniform',
    bias_initializer='zeros'),

Dropout(0.50),
Dense(256, activation=tf.nn.relu,
    kernel_initializer='glorot_uniform',
    bias_initializer='zeros'),

Dropout(0.50),
Dense(128, activation=tf.nn.relu,
    kernel_initializer='glorot_uniform',
    bias_initializer='zeros'),

Dense(output_shape[0] * output_shape[1],
    activation=tf.nn.sigmoid)
])

model.compile(
    optimizer=Adam(lr=0.0001),
    loss='mean_squared_error',
    metrics=[acc_meas])
```

ACKNOWLEDGMENT

We gratefully acknowledge the contribution of our advisor for allowing us to utilize his personal dataset of the Cal Poly computer science building. Over 3000 images were manually collected and painstakingly hand-labeled over the course of a year. The success of this project would not have been possible without this contribution.

REFERENCES

[1] Wallace. Wuerch, Labeling Paths with Convolutional Neural Networks, (2019), GitHub repository, https://github.com/heatsink/SNetZ