Teaching Computers to Teach Themselves

Synthesizing Training Data based on Human-Perceived Elements

James I. Little

Advised by Prof. Eric Chown
An Honors Paper for the Department of Computer Science
Bowdoin College, Brunswick, Maine

May 2019

Bowdoin College, 2019

Abstract

Isolation-Based Scene Generation (IBSG) is a process for creating synthetic datasets made to train machine learning detectors and classifiers. In this project, we formalize the IBSG process and describe the scenarios—object detection and object classification given audio or image input—in which it can be useful. We then look at the Stanford Street View House Number (SVHN) dataset and build several different IBSG training datasets based on existing SVHN data. We try to improve the compositing algorithm used to build the IBSG dataset so that models trained with synthetic data perform as well as models trained with the original SVHN training dataset. We find that the SVHN datasets that perform best are composited from isolations extracted from existing training data, leading us to suggest that IBSG be used in situations where a researcher wants to train a model with only a small amount of real, unlabeled training data.

Acknowledgements

As would be expected, this thesis would not exist without the help of many different people in my life, to whom I'll never be able to express enough gratitude.

Thanks first go to Professor Eric Chown for his weekly willingness to talk about academia, programming, teaching, and life throughout this year, and for his guidance over the course of my Bowdoin career. I relied many times over the past four years on his patience and his seemingly-never-ending trust that everything from RoboCup to this project would all work out. I wouldn't be the scholar or the computer scientist I am today (and will be tomorrow) without his guidance. *Roll!*

Thanks as well to Stephen Majercik, my second reader, and to the rest of the Bowdoin Computer Science faculty. I'm so glad I got to learn from such a fun, knowledgeable bunch of people. Thanks too to RoboCup captains and team members who came before me for giving me a space to learn and grow.

I owe so many thanks to my parents and brother for the love and support they have unconditionally given over the course of my life. Thank you for teaching me how to be the best person and learner I could be — this paper couldn't be what it is without you. To my girlfriend, Maddie, as well: words cannot express how grateful I am for the motivation you've provided. Thank you for everything.

Finally, to various friends and roommates: Thank you for bringing endless fun to the past four years of our lives. I treasure you all.

James Little
Coles Tower 8A, Bowdoin College, Brunswick, ME
May 2019

"Synthetic data is like synthetic fabric: it looks the same, but it's just not quite as good as the real thing."

–Madeleine Tucker

Written in Ulysses, Visual Studio Code, and Vim, and typeset in LaTeX. Diagrams made in Affinity Designer. Body text (and nearly everything else) is set in Minion Pro, the sans-serif type in diagrams is set in Avenir, and monospace text is set in `Operator Mono`.

Visit `http://honors.jameslittle.me` for scripts and other digital assets.

Contents

List of Figures

In recent years, researchers have looked to machine learning as a solution to otherwise challenging technical problems. Artificial neural networks (ANNs), the leading machine learning technique used today, are used to build complex problem-solving algorithms that would be too difficult for researchers to build on their own. An ANN algorithmically constructs an intricate, specialized linear function that transforms high-dimensional input data into a lower-dimensional output. ANNs are an example of *supervised machine learning*: a neural network will "learn" over the course of a lengthy training procedure by examining pre-classified input/output pairs, randomly adjusting the linear function (the *model*) to minimize the error between the function's output and the known ground truth values of the training data.

One major challenge in setting up an ANN is in obtaining *labeled training data*: a dataset that annotates input data with the desired output data. Traditionally, a large amount of human effort goes into creating such a dataset, since labeling collected input data often requires a human to classify (and validate the classification of) tens of thousands of data points.

In this project, we explore a new method for producing a training dataset based on *synthetic data generation*, called *Isolation-Based Scene Generation* (IBSG). IBSG involves identifying and extracting key "ingredients" in the dataset and computationally compositing these ingredients to create artificial, automatically classified scenes. We train Tensorflow-based neural networks with hand-labeled, real-world-collected data[1] and train similar networks with IBSG data that has been designed to mimic that real data. We compare the accuracy of those neural nets using a real-data test set on both the IBSG-trained models and the real-trained models. Finally, we explore if and when ISBG is an appropriate replacement for real data, and make suggestions on how ISBG datasets should be created and designed.

[1] For the rest of this paper, data of this type will be described as "real data" or "real training data," for the sake of brevity.

Chapter 1

Background

1.1 Artificial Neural Networks

ANNs are loosely based on the neuronal structure of the human brain. In the brain, neuron cells are cells connected to each other through *synapses*: physical gaps between these cells across which electrical information is propagated. The typical neuron will have a synaptic connection to 100 to 100,000 other neurons via that neuron's dendrites: long tendrils that extend from the soma, the neuron's central body.[1] Synaptic connections between neurons transmit binary information: a cell is either transmitting an electrical impulse to its connecting cells, or it is not. The receiving neuron might be receiving these electrical impulses from many different cells, and has an activation threshold (around -55 milliVolts) at which point it itself fires, propagating an electrical signal forward. The brain is "programmed" through the different cellular connections: the specific neurons that get a certain signal forwarded to it describe which pieces of brain functionality are connected. These cellular connections are fluid: they can be created or strengthened (this is when the brain "learns") and can also be weakened or destroyed (when the brain "forgets").

Artificial neural networks are computer programs written to follow similar principles as the neuronal system in the brain, hopefully allowing computers to programmatically make complex inferences similar to those learned biologically. In a neural network algorithm, neurons and their connections are represented as nodes and edges in a directed graph. Each edge has a *weight*: a scalar value that determines the proportional strength between the outgoing node's output and the incoming node's corresponding input. Each node has an *activation function*, a function that operates

[1] Mehrotra, Mohan, and Ranka 1997

on the node's input values, rendering a single value; and a *bias*: a threshold for that value that determines how easily that node will fire (send output). In this way, the weights, activation functions, and biases of the system determine how information is forwarded between nodes, how information is propagated along the input, and therefore how input data gets transformed into output data. While the activation functions of certain neurons are determined beforehand by the researchers who design the network, the biases and weights across the system are determined by an automated learning process, which is described later.

Feed-forward networks, a simpler type of ANN, are built from directed graphs whose nodes are connected in layers, where each node has an edge connecting it to every node in the preceding layer and to every node in the following later (Figure 1.1). The first layer of neurons, deemed the "input" layer, starts out with node values set directly from the input information.[2] The following nodes are termed "hidden layers:" the node values themselves have little to no semantic meaning, but instead provide intermediate values that can complicate the transformation.[3] Finally, the nodes in the "output" layer are determined through the propagation of information over the previous neuron layers. These output neurons are usually defined in the training stage as corresponding to the different classifications that are trying to be learned; it is up to the implementer to decide how the training data will be labeled,

[2]For example: If the input data is a color image, where each pixel is represented by a red, green and blue tuple, the input layer will have three nodes (holding red, green, and blue data respectively) for each pixel in an image. If a network takes a 640x480 color image as its input, the first layer will have 921,600 nodes. This means that input data has to be normalized for the type of network being described here. If a network is designed for (i.e. trained on) 640x480 color images, the only input data it will be able to analyze will be 640x480 color images. Larger images will have to be downscaled.

[3]A "human" way of understanding this might be: when a person looks at a face, they first work out what small features they see. The nodes in the intermediate layer might determine whether a left eye, right eye, nose and mouth are present, while the final layer would determine whether the image has a face in it. The network needs to traverse through these lower-level understandings before coming to a final conclusion. The hidden layers often have no discernible meaning (so, then, they wouldn't represent the presence or absence of the facial features above); instead, they represent what the computer has deemed as intermediate values. But adding more hidden layers gives the computer more variables to tweak during the training process, adding more nuance to the network and possibly giving it better results. The design of a neural network is currently a completely different field of research in itself, since the amount of intermediate layers and the activation functions for nodes in those layers can drastically change the success of a neural network.

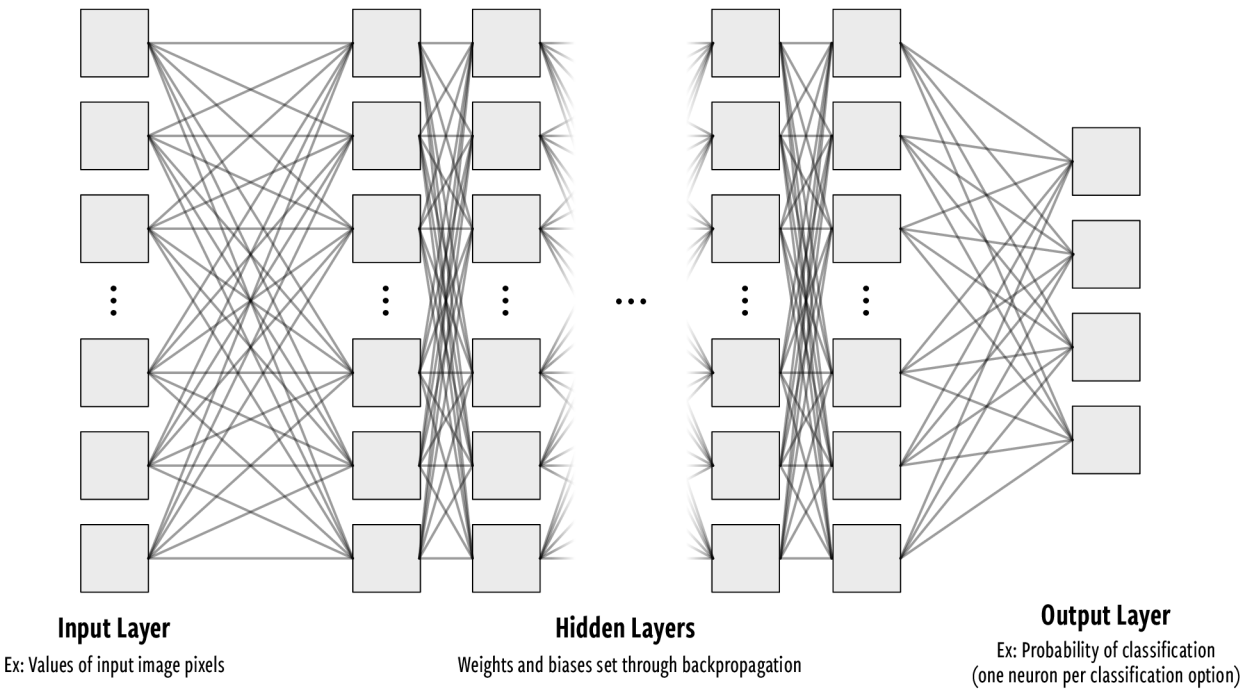

Figure 1.1 – A graph diagram of a feed-forward artifical neural network. Any edge in this diagram is a directed edge: information only flows rightwards from one layer to the next. In a typical ANN, the output layer is much smaller than the input layer, and the hidden layers generally step down to progress from the high-dimensional input space to the lower-dimensional output space.

and therefore, what output the ANN is testing for.

Researchers building a feed-forward neural network hope to create a linear transformation function that turns a point in high-dimensional space (the "space" here being the range of possible input values) into a point in a lower-dimensional space (the range of possible output values). Organizing a feed-forward network into layers serves to organize the propagation of information; these layers control the flexibility of the network in building that function, meaning that with more layers, the network (via the training process) has more opportunity to converge on the "most correct"

function.[4] Typically, each node in a given layer has the same activation function, normalizing how information gets processed through that layer. Different layers, however, will employ different activation functions. The properties of the network such as the number of layers, the number of nodes in each layer, and the activation function for each layer's nodes are set before the training process, in contrast with the weights and biases, which are set *during* the transformation process. The former are called the *hyperparameters* of the network; the latter are called *parameters*.

1.2 Training

It was discovered by researchers in the 1970s and 80s that the weights and biases of the system could be learned algorithmically, so that a computer can make its own connections between nodes instead of relying on human-created connections. As part of this learning process, the weights of each edge and biases of each node could be thought of as different variables that could be tweaked to adjust the network's output. When the network was given sample input/output pairs, the computer could automatically adjust the variables to minimize the difference between expected output and calculated output. The training algorithm to tweak these variables automatically was designed and termed *error back-propagation*,[5] and functions as described:

1. The computer randomly assigns biases for each node and weights for each edge in the network.
2. For each piece of data in the training data, the computer runs the input through the network.
3. The computer calculates the error between the network's output and the data's annotations (known as the *ground truth*), and then
4. Uses the mathematical process of *gradient descent* to modify the weights and activation functions so that the amount of error is reduced.

[4] However, because there is more flexibility, the convergence process involves searching through a larger problem space, resulting in a longer process.

[5] Error back-propagation was supposedly discovered independently by several researchers in the 1970s, and spread by Cun (1986), Parker (1985) and Rumelhart et al. (1986) later. For more information on the history of neural networks, see Müller, Reinhardt, and Strickland 1995.

5. This process is repeated until the testing results start to show signs of *overfitting*, a phenomenon in which the training error rate and the testing error rate start to diverge, and the network becomes unable to perform well when tested with data from outside the training set.

After this training process is complete, the weights and biases of the neurons have been set in such a way to minimize testing error, and the computer is reasonably sure that, given the network's hyperparameters, it has found a set of parameters that most accurately[6] correlate the training data's input with the corresponding output. If training is successful and error has been minimized enough, the researcher can be confident that the ANN, when given new, unknown input data, will yield output data with a high probability of correctness. For example, if a computer trains on pictures of street signs labeled with what type of sign they are (e.g. a stop sign, yield sign, speed limit sign, etc.), it can then be given a new picture of a street sign, run it through the trained neural network (really just a set of linear equations), and can output the relative probabilities of which street sign it is.

This training step is a fundamental part of today's machine learning. The computer can glean meaningful information from sample input data, examining the input/output pairs to make connections, with the ultimate goal of programmatically discovering what features of the input lead to changes in output. Artificial neural networks are at the heart of nearly all of today's machine learning implementations. They can take in a tremendous amount of seemingly unrelated input data, pick out relevant features (perhaps even features invisible to humans), and draw classification conclusions that are (ideally) similar to or better than those made by real people. They fuse together the reliability and speed of computers with the biological classification ability of the human brain.

[6] The hyperparameters and parameters of a neural network both determine that network's success. In other words, if the hyperparameters of the network are ill-suited to the problem space, there might not be a set of parameters (weights and biases) that the computer can optimize toward to accurately describe the relationship between input and output. The training step, therefore, is not guaranteed to find an accurate linear equation that transforms input into output; it can only find a local minimum for testing error given the parameters available.

1.3 ANN Training Data

As described previously, one challenge in using ANNs is obtaining *labeled training data*: a dataset that both models the intended input data for the network and annotates each piece of data with the neural network's desired output, were the network to receive that piece of data as input. ANN researchers find that a high-quality training data set will exhibit four overarching characteristics:[7]

1. The annotations must have a certain level of **accuracy**.
2. There must be a large **quantity** of training data. MNIST (Figure 1.2), a dataset of handwritten digits collected from Census Bureau employees and high school students, has over 60,000 images as part of its testing set.[8]
3. The training data must be widely **varied**: it must represent each class with a diverse set of samples. Figure 1.3 gives an example of a training dataset that exhibits poor variety.[9]
4. The dataset should exhibit **realism**: there should be very little difference between a representative sample of the training data set and a similar sample of the testing data. Figure 1.4 gives an example of a training dataset that exhibits poor realism.

The quality of the training data set directly impacts the success of the machine learning algorithm, so researchers will often go to great lengths to ensure that the variety and labeling of the training data set is accurate. Therefore, a traditional well-implemented ANN has a large, varied, well-labeled training dataset backing it, a dataset that has had many man-hours put into maintaining its quality.

In recent history, researchers have been exploring the use of large, computer-generated datasets to train ML models. Instead of collecting and labeling data from measuring equipment data, a computer algorithm generates a data set that mimics the data that might have been collected naturally. Since the computer algorithm

[7] Three of these four elements are discussed in further detail in §2.8 of Weiss and Kulikowski 1991, entitled "What Can Go Wrong?".

[8] LeCun, Cortes, and Burges 2019

[9] The consequences of poor variety are described further in Batista, Prati, and Monard 2004.

Figure 1.2 – Sample digits from the MNIST database. Each digit is its own data point and is labeled with the number that is drawn. Researchers training models with the MNIST dataset often have the goal of training the model to detect digit values from handwritten numbers. Source: Josef Steppan, WikiMedia (`https://en.wikipedia.org/wiki/MNIST_database#/media/File:MnistExamples.png`)

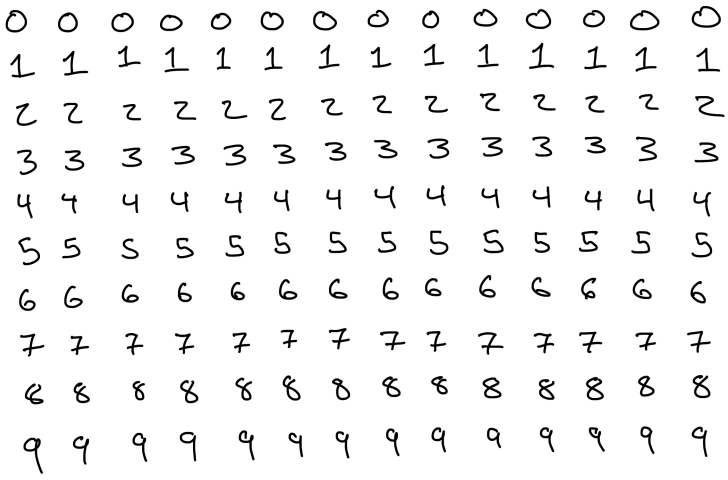

Figure 1.3 – A training dataset that exhibits bad variety when compared to the MNIST dataset. Instead of collecting handwritten samples from many different people, this dataset only contains handwriting samples from one person. If a model trained with this data is tested with a handwriting sample from another person (or even the same person, writing differently) it will perform poorly.

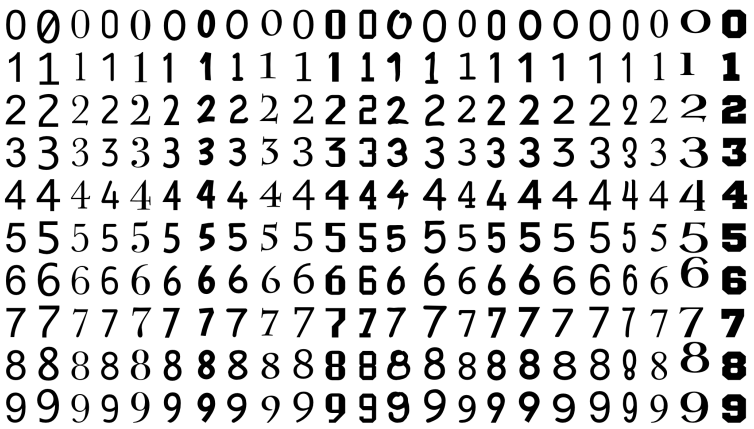

Figure 1.4 – A training dataset that exhibits bad realism when compared to the MNIST dataset. This dataset consists of digit samples from computer fonts instead of handwriting. When tested with handwriting data, a model trained with this dataset will perform poorly.

generates each piece of data, it also has insight into specific features of the data, and can thereby label each piece of the data with the learned property as it is generated. When creating synthetic training data, the job of the human supervisor is no longer to collect and label data; the computer can take care of both. Instead, the supervisor must ensure the training data generation algorithm outputs data with a wide enough variety to match the variety seen in testing, while also ensuring that the data remains realistic.[10] For a synthetic training data set to be successful, it must have the same properties as a high-quality human-curated data set: namely, quantity, variety, and realism. The responsibility in creating such a model synthetically thereby falls to the programmer: they must encode the variety and unpredictability of the real world into a data-generating algorithm.

Synthetic training data has been used for years already to generate working machine lelarning models. Several published applications of ML use synthetic data

[10] Compare this to real data collection, in which variety and realism are guaranteed, since a representative sample of a dataset will inherently be as realistic and as varied as the overall dataset. Researchers building a training dataset from real data have to work to ensure annotation accuracy and data quantity; researchers building a training dataset from synthetic data have to work to ensure variety and realism while accuracy and quantity come for "free."

to either augment or replace the human-supervised data, but the method through which the synthetic data is generated can vary. Some studies use a statistics-based approach: finding the mean and standard deviation of numerical data and generating other data based on these values.[11] Other studies that experiment on image data use 3D modeling to generate a scene and use special rendering techniques to make the render as realistic as possible. In this project, we explore a new data generation methodology using a system based on manipulating and compositing *isolations*: elements from samples of real data that a human observes and extracts. We test the viability of using isolation composition to augment or replace hand-labeled data in different feed-forward ANN applications, and explore the different ways isolation-based synthetic data can be generated.

The following section gives an overview of existing techniques and applications for synthetic data to train ANNs. Section 3 presents a formal definition of an isolation, indicates which form of data can be generated using isolations, and describes types of ANNs that work best with these data types. Section 4 sets up the experimental methodology for testing viability and presents its results. Section 5 proposes future work.

[11] An example can be found when training a neural network to model a linear ($y = mx+b$) function. The function approximated by the neural network will converge on that line, and the training data can be calculated by sampling any number of points along that line.

Chapter 2

Prior Work

Prior research in the field of synthetic ANN training data has focused almost entirely on image analysis, in which a neural network is trained to detect certain features of 2D images. I present two common methods for synthetic data generation within the context of creating a vast quantity of labeled images to feed into a neural network: generation using 3D modeling and generation using 2D image composition.

2.1 3D Modeling

Shrivastava et al. 2016 turned to synthetic data as a way of automatically gathering annotations for data: they used advanced 3D rendering techniques to generate realistic data while maintaining the annotations the model was able to attach. This advanced rendering used Generative Adversarial Networks (GANs) as part of the rendering process to transform "perfect" 3D renders into degraded renders that more closely match the degradation seen in real-world data, shown in Figure 2.1. They tested five scenarios, each combining different amounts of real data, traditionally-rendered synthetic data, and GAN-assisted rendered synthetic data. This study found that a model trained on traditionally-rendered synthetic data performed slightly worse than one trained on real data, but a model trained on GAN-assisted renders, especially when the amount of synthetic data was tripled, surpassed the model trained only on real data. In this case, with advanced rendering of 3D models, synthetic data can be a powerful addition to a training dataset.

Others still use 3D-modeled data from other sources with a similar result. In a 2016 paper, Richter et al. 2016 used gameplay footage from *Grand Theft Auto*, *Watch Dogs*, and *Hitman* as a source of synthetic data (Figure 2.2) and found that

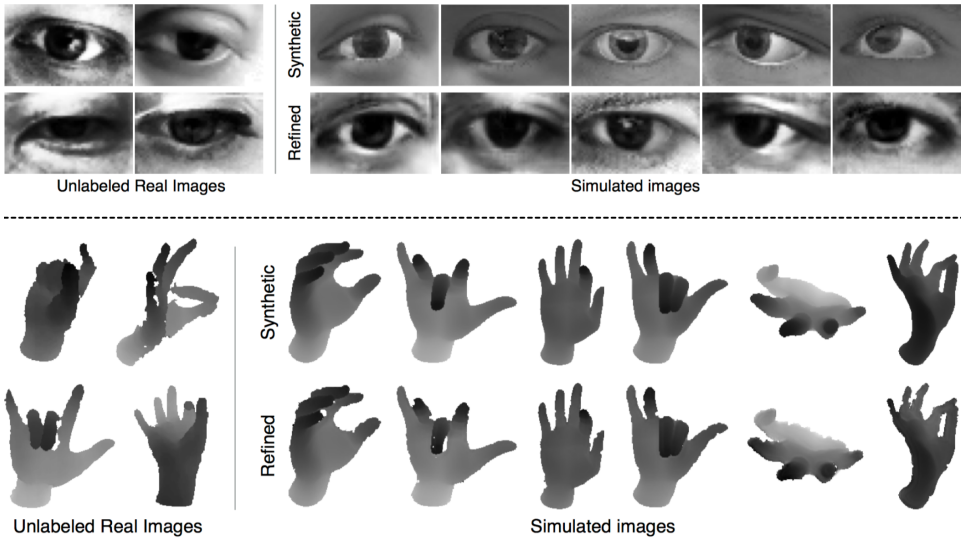

Figure 2.1 – Real, synthetic, and refined synthetic data from Shrivastava et al. 2016's research. The refined synthetic data uses an advanced, machine-learning-assisted rendering process to make the synthetic data match the real data in realism.

a model trained on one third of the real data mixed with lots of gameplay footage performed better than a model trained on the complete real data training set. This shows a large opportunity for synthetic data generation to replace much of the real-world data collection and labeling that takes place: in this way, large datasets can be generated faster and with greater accuracy. Clearly, there is some benefit to using synthetic training data: it can, in certain scenarios, outperform real data both in dataset collection time and in the resulting performance of the classifier.

3D renders are a valuable source of synthetic training data in cases where real-world data cannot be collected. Hattori et al. 2015 presented a scenario in which a new surveillance system was installed and a model needed to be trained to detect pedestrians with no prior observations available; using synthetic data to train the model was deemed to be a possible solution to training the domain-specific model. The team rendered footage of a 3D environment that matched the viewpoint of the new camera, modeling the objects and buildings in the scene and introducing simulated pedestrians (Figure 2.3). They found that synthetic data models outperformed naïve pedestrian detection algorithms, models trained on hybrid data, and, in cases

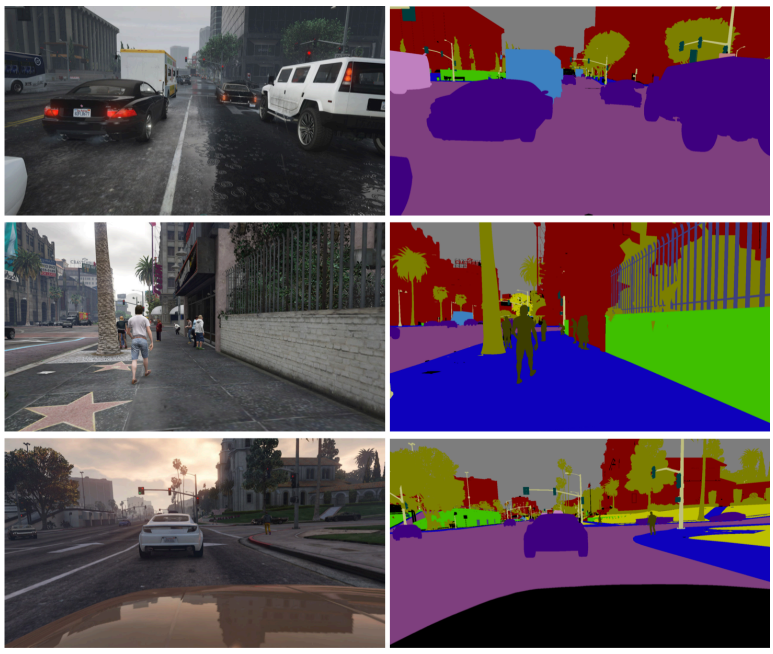

Figure 2.2 – Examples of video game footage and the associated classification labeling from Richter et al. 2016 Different colored pixels in the right images indicate different human-labeled classifications.

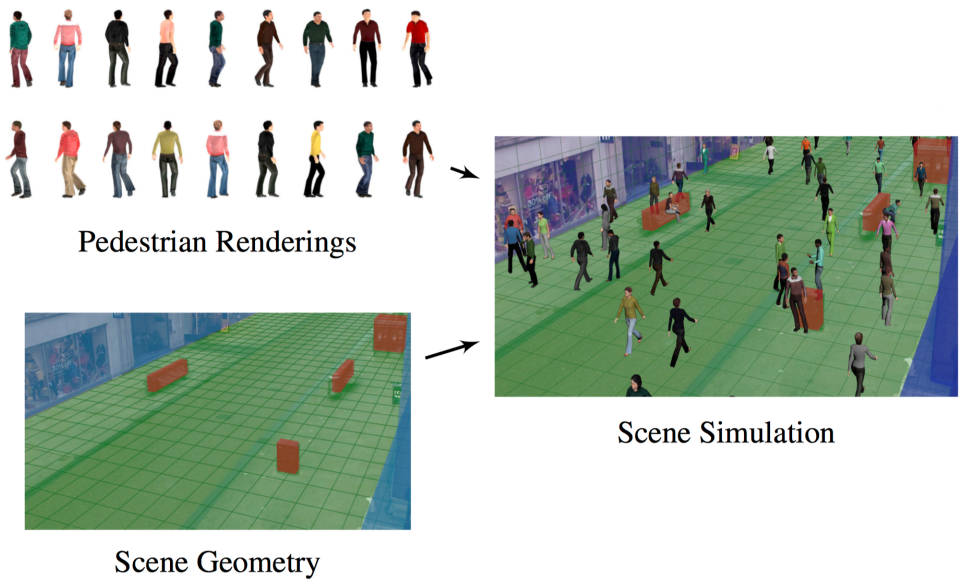

Figure 2.3 – Synthetic data used to train a person detector for a specific security camera vantage point, created by Hattori et al. 2015. The camera's viewpoint was rendered in 3D software and the renders were used to train the detector.

where real data is limited, even models trained on real data.

3D modeling can be a powerful tool for generating synthetic data; however, building a 3D model that mimics a scene can be labor intensive and requires specialized 3D modeling abilities. Furthermore, depending on the modeling and rendering capabilities of the 3D artist, the scene might not resemble testing data closely enough: note that the synthetic data described in the three studies above were either generated using advanced 3D modeling and rendering techniques (as seen in Shrivastava et al. 2016 and Richter et al. 2016) or the fidelity of the 3D render was not important for training (as seen with Hattori et al. 2015, who modeled a scene from a camera view far away). Where 3D rendering methods become less feasible, other data generation techniques have come into play.

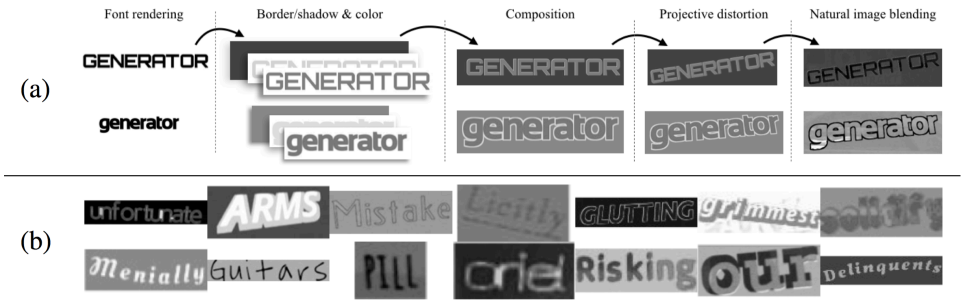

Figure 2.4 – Figure (a) shows the image composition process by Jaderberg et al. 2014 for two-dimensional word classification. Figure (b) shows an example of images created by this process.

2.2 Image Composition

Two-dimensional image composition and manipulation, much like what one can do in Adobe Photoshop, is another method frequently used to synthetically generate image data. With this method, multiple 2D images are combined and mutated to create another 2D image; that output image then becomes part of the data training set. Image composition is utilized by Jaderberg et al. 2014, who designed a synthetic data generator to create training for ANN models, ultimately trying to teach a model to recognize text in real-world images. The images they created (Figure 2.4) were built algorithmically, compositing and modifying different layers on top of each other to randomly generate realism and introduce variety. The model trained with this data is meant for classification instead of detection; given an image of text, the model should be able to determine which word is shown. To generate their training data, they used a six step process: 1) a word was rendered to an image using a given font; 2) that rendering was given a random border and shadow; 3) the background, border, shadow and text were colored using color clusters taken from real-world data; 4) the text, border, and shadow are randomly transformed using a perspective distortion; 5) the image is randomly blended with a crop of a real image; 6) noise, blur, and compression artifacts are added. This data generation process trained a model that could outperform three other traditional word detection algorithms.

2D image composition can be especially powerful when combined with 3D

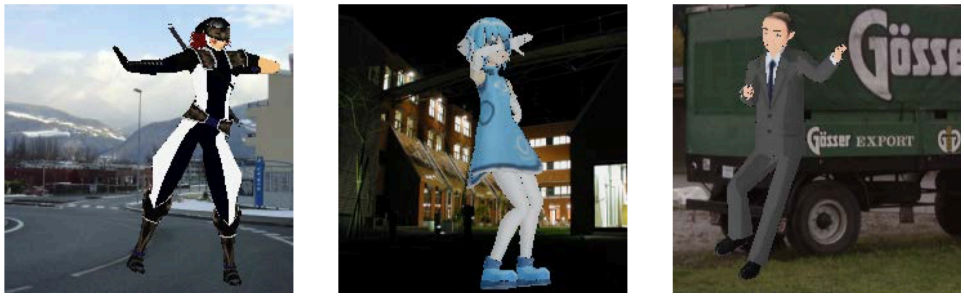

Figure 2.5 – Images of rendered characters in specific poses composited over 2D backgrounds, by Khungurn and Chou 2016 The resulting two-dimensional images are used to train a model to recognize the three-dimensional pose the character is in.

modeling. Khungurn and Chou 2016 explored image composition while studying pose estimation—estimating how the joints and limbs of a human body are placed in 3D space—of two-dimensional anime characters. The team used pose information from 3D characters and rendered those characters to a 2D image. They then combined those character renders with artistic 2D backgrounds, using minimal transformation and composition techniques, the output of which is shown in Figure 2.5. They were able to build an estimator that could effectively estimate human poses in images that contained anime characters.

The use of synthetic datasets to solve individual machine learning problems has been well-explored. Each research group above has struggled within their specific problem domain and found a synthetic data generation process that works to better the accuracy of their detection or classification model. We hope to generalize some of their work. The techniques presented so far have proven that synthetic datasets can be successful when the generation technique is tailor-made for the ANN's learning objective. In borrowing from these techniques, we hope to describe a general-purpose process for data synthesis that can aid researchers in creating detectors and classifiers, regardless of what is being detected or classified.

Chapter 3

Isolations and Synthetic Data Generation

I introduce a new method of synthetic data generation based on image composition and 3D modeling. This method, which I call *Isolation-Based Scene Generation* (IBSG), can be used to generate both visual and auditory detection models, and is optimized for minimal supervision while still providing quantity, variety, and realism. IBSG, I argue, works to bridge the gap between computational detection and human perception, working with the best parts of both to create object detection algorithms that are both effective and easy to build. To understand how IBSG connects to human perception, however, it first must be explained how humans gather and process information about the world around them.

3.1 The Psychology of Detection

Humans are very good at using their senses—particularly sight and hearing—to infer things about the world and their current environment. Visual and auditory stimuli combined with intense, specific processing in the brain are the foundation for object recognition: the primary way we orient ourselves to a new environment is by trying to determine what we see and hear. It is these two types of data, sight and sound, that the brain has evolved to process incredibly well, since it is these types of data that give us the most information about our surroundings.

But what does it mean to "process" visual and auditory data? Psychologists recognize the importance of object recognition: the ability to extract features from stimuli and recognize an area of input as representing a certain object. In studying the brain, scientists have determined that humans search for specific features of stimuli: defining or distinct characteristics of a whole object. The brain then analyzes

those features and tries to understand what known category that object belongs to. There are two leading theories behind categorization that elucidate how the brain thinks about a scene—exemplar theory and prototype theory—both of which have implications for how data is modeled in a neural network training dataset.[1]

Exemplar theory states that as we go about our lives, we build up a library of objects in our brain that are already categorized: when we see a new type of fish, we store it as a fish. When we see a new object, we compare it in parallel to every stored object before and make a comparison: if the object's features match up with the features in almost every fish we've ever seen, we can be confident it's a fish. Conversely, if a new object doesn't resemble any instance of a certain category, we have trouble categorizing it as such.

Prototype theory is similar, but argues for a center-out comparison model characterized by pre-computation. As we see different types of fish, the brain build up an idealized prototype fish that is the "average" of every fish it has seen. New observations are compared against these prototypes: the farther away the observation is from the prototype, the more difficulty we have detecting it.

Prototype theory and exemplar theory are similar in practice, but exemplar theory argues for a system in which an object is categorized based on a category's boundaries: if we've seen another fish that resembles this fish, the object is within the *fish* category's bounds, and can be classified as such. Prototype theory instead builds a model around a category's center, and every new object is judged by how close or far it is from that center: in this way, an object is either more fish-like or less fish-like.

It happens to be the two types of data that humans are best at classifying—auditory and visual data—that computers have the most difficulty classifying. Ever since the earliest computers were designed, data has been modeled as numbers in a one-dimensional list. This format is particularly bad for defining visual and auditory data, both of which rely on spatial dimensions: vision is a two-dimensional projection of a three-dimensional world, and sound only has meaning because it can be plotted as different combined frequency values over time. It proves difficult for

[1] Exemplar and Prototype theory knowledge from Reisberg 2016

computers, which store values in only one dimension, to computationally recognize features in a two or more dimensional space.

Much of today's ANN research is aimed at giving computers the ability to classify auditory or visual data to a similar degree of accuracy with which humans can perform that same classification. I argue, in fact, that audio and image classification is an area of research that further bridges the gap between humans and computers. It is for these types of input that neural network based recognition can have the most impact; as seen in prior research, neural networks can effectively process auditory and visual data to recognize objects in the scene with a similar degree of accuracy to human recognition.

3.2 Isolation-Based Scene Generation

IBSG is a method of building artificial datasets of images or audio made for training ANNs. An IBSG dataset is created through a series of steps.

First, the researcher either obtains a small amount of training data or imagines what such data would look like. This small dataset will contain *scenes*: individual data points that take the form of a single image or a single audio clip. From these scenes, the researcher identifies *isolations*: individual components of these scenes that correspond to individual entities the brain can recognize as distinct. An example of isolations identified within a scene can be found in Figure 3.1. For most visual inputs, identifying isolations is a relatively simple process: for the most part, the researcher can observe which physical objects are present in a scene, and all those objects would count as a category of isolation. Elements like lens flares, however, would also count as isolations. For audio scenes, isolations can be sound samples from individual sources that can be reproduced on their own, like a ticking clock, a piece of music, a car driving by, or a sample of one's voice. Importantly, though, the isolations chosen by the researcher should be present in different forms across the different scenes in the sample data. Ultimately, these isolations will be recombined computationally to create an artificial scene which should look as much like a real scene as possible.

The researcher then extracts examples of those isolations from existing data or,

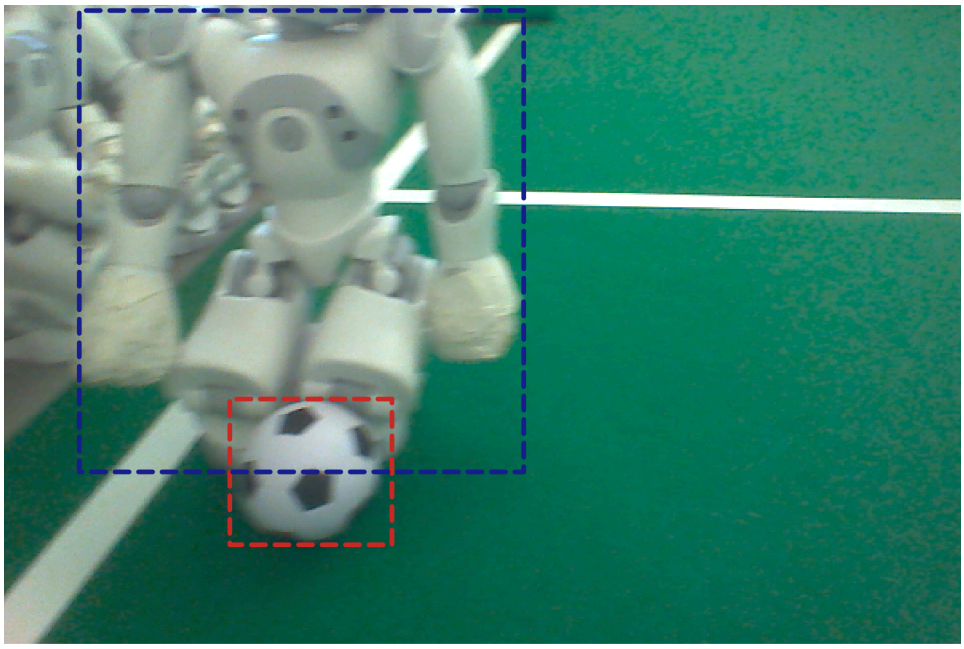

Figure 3.1 – An image collected by a robot in a robotic soccer field, showing the field in the background, an adversarial robot, and the soccer ball. A human looking at this image might describe three isolations: the background, the robot, and the ball. These isolations have been highlighted here with dashed boxes indicating how the human object detection process would segment the image.

if possible, records new samples. These extracted isolations might take the form of transparent cutouts of objects, texture layers, or background images. A sample of assorted extracted isolations is shown in Figure 3.2. These extractions might be extracted manually using an image manipulation program like Photoshop, or they might be generated computationally; for example, image grain, text, or blocks of color might be candidates for computational isolation generation since there might not be a simple way of extracting these elements from existing images. For models training on audio data, researchers might have to record individual bits of audio like dialog, foley sounds, or background noise — extracting these elements from a recording in which they are combined is a notoriously difficult problem. For each isolation, researchers should actually generate several examples so that the compositing program can pick and choose which examples interact with each other in the final scene, creating more variety in the final generated dataset.

These extracted isolations can be thought of as exemplars: specific instances of a category of object that the computer, like a human, uses to create a conceptual model for how to recognize other objects within that category. The ANN training process, when using IBSG, can be compared to a brain that works exclusively using exemplar-based detection. Such a brain would not be able to recognize an object if that object were dissimilar from any exemplar seen before: even if that brain saw a rare fruit, it would be unable to categorize it as a fruit if the brain hadn't seen a similar example before. IBSG presents a computer with a series of examples in the context of a scene. These examples—and the variety and realism that is present in the collection of these examples—determines the body of knowledge the computer has surrounding each category of objects.

Importantly, one of the isolations must be the object that the machine learning model is trying to detect, called the *learned isolation*. Since the computer will have knowledge of where those isolations exist in the final, generated scene, and since the computer will control the random placement (or selection) of that isolation, the output data must be some information the computer can record when the scene is generated.

With these isolations identified and prepared, the researcher must create a computer program (called the *compositing program*) to join these isolations together into

Figure 3.2 – Images of balls, robots, and field backgrounds extracted in photo manipulation programs or taken independently. These isolation examples fall into the categories found by the human in Figure 3.1.

an output scene, with some element of randomness as an inherent part of the program. Examples of the ouput of a compositing program is shown in Figure 3.3. The program must randomly choose compositional elements such as:

- Where each isolation is placed in the final scene
- How the isolation is transformed (rotated, scaled, sheared, etc.)
- Which example of each isolation is chosen

This amount of randomness alone is often not enough to create the variety and realism needed for a high quality training dataset. Therefore, the program should also introduce randomized *modifications* into the scene, acting on both the individual isolation examples and on the final image (shown in Figure 3.3(b)). For example, an image isolation might be blurred, darkened, lightened, sheared, projected, or otherwise transformed. An audio isolation might have its volume changed arbitrarily over time, might be pitch shifted, might be compressed or muffled or distorted. The final scene may be distorted in similar ways. The overall randomness of the composition is up to the researcher; one might strive for ultimate realism by introducing limitations on the modifications (like ensuring that humans are never bigger than trees); however, this is not always necessary. Ultimately, the nature of the distortion, much like the nature of the composition, must be determined by the human making the artificial dataset; this decision will be much more artistic than it will be procedural. In generating the dataset, the compositing program must also record an associated annotation for each scene; this annotation is almost always the location or selection of the learned isolation, and will be the resulting information that the ANN learns after going through the training step.

The final step in the IBSG process is to run the compositing program to generate a large amount of artificial scenes: these scenes will all make up the final training dataset. While there will be many dimensions of randomness in the compositing program, the final dataset will, in theory, be large enough to cover nearly every combination of these random elements.

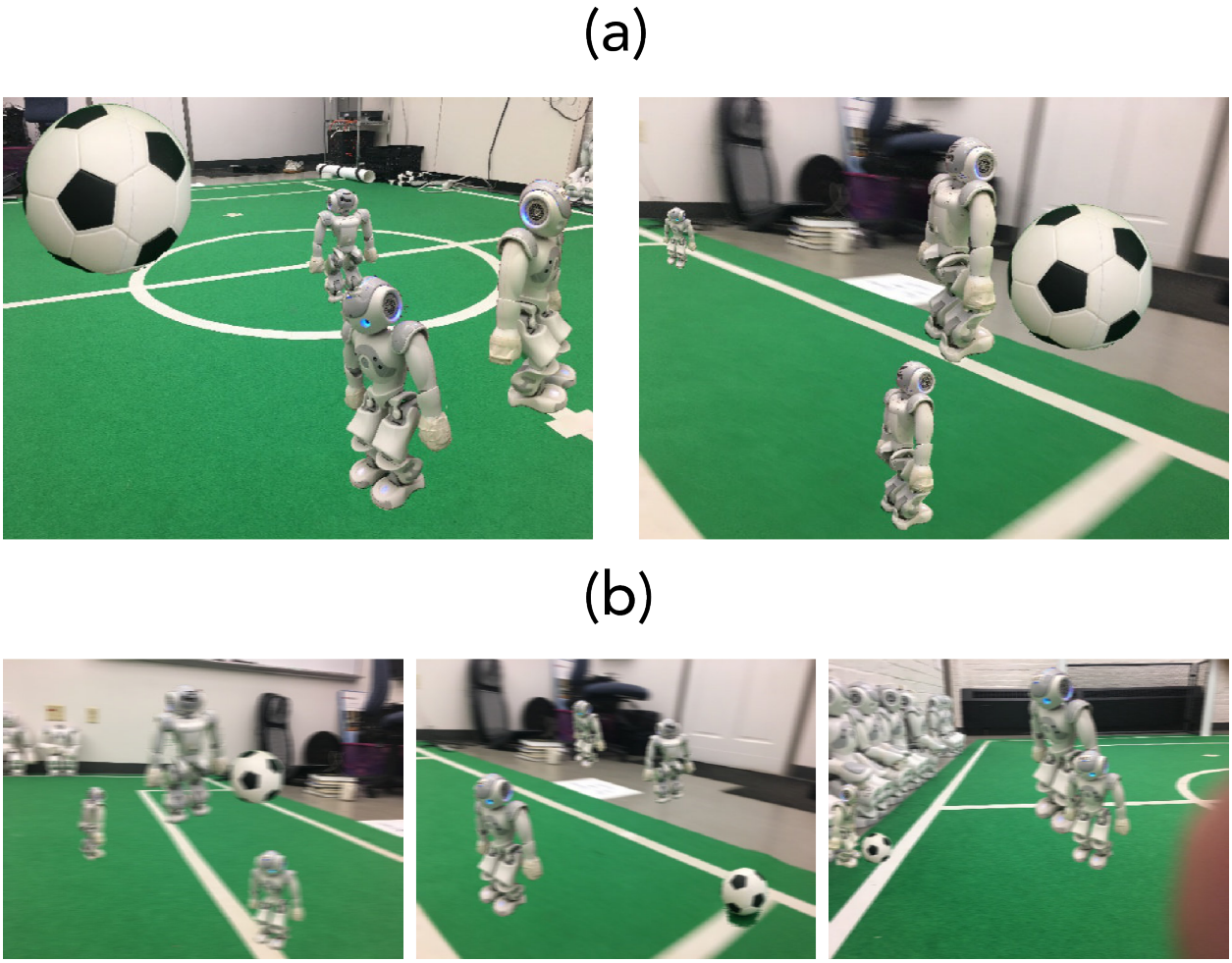

Figure 3.3 – Images programmatically generated by randomly combining the extracted isolation examples from Figure 3.2. Subfigure (a) shows images without any modification: the ball and robots are randomly scaled and placed. Subfigure (b) shows images that have motion blur added to the output image, as is typical of images taken from a roving robot. These modifications serve to enhance the realism of the final image, making the model perform better on real testing data.

3.3 When IBSG is Useful

When a scene in an IBSG dataset is created, the compositing program knows the location or selection of each individual isolation within the scene. In this way, the compositing algorithm can output the input to the neural network (the scene) and the corresponding output (the annotation for the given scene, which is usually the placement or selection of a chosen isolation); the collection of scenes and annotations is then used as the training dataset. Importantly, this means that the neural network's output—the information the network learns—must be information that the compositing algorithm can output. Since IBSG builds up scenes based on the objects humans can detect and isolate, it does not make a lot of sense for IBSG to be used for spreadsheet data; humans have trouble detecting patterns in numerical data. These scenes, I argue, should only be images or sound clips, the two types of sensory input that humans can easily separate into component parts (easily knowing which objects are in an image or which individual sounds are in an audio clip), but which are recorded by digital input devices as already blended together as a single image or a single sound file.

As described previously, IBSG serves a use case where humans can easily identify *where* or *what* an object is, especially in scenarios where computers might not be able to make such an identification: the neural network is therefore either learning how to *locate* or *identify* objects in a scene, as shown in Figure 3.4. In scenarios involving location, the compositing program has placed the learned isolation somewhere in the image, possibly obscured by other isolations or modifications; the trained model should ideally be able to output a bounding box for the location of that object. In identification scenarios, the compositing program has chosen a certain sub-type for the learned isolation (if the computer needs to recognize types of street signs, the computer will record whether it placed a stop sign, yield sign, or speed limit sign in the final output).

IBSG or any other synthetic data generation method only makes sense if the researcher doesn't have a real labeled dataset available. As described previously, real datasets will provide much more realism and variety than a synthetic dataset. If a researcher only has an unlabeled real dataset or a very small labeled real dataset,

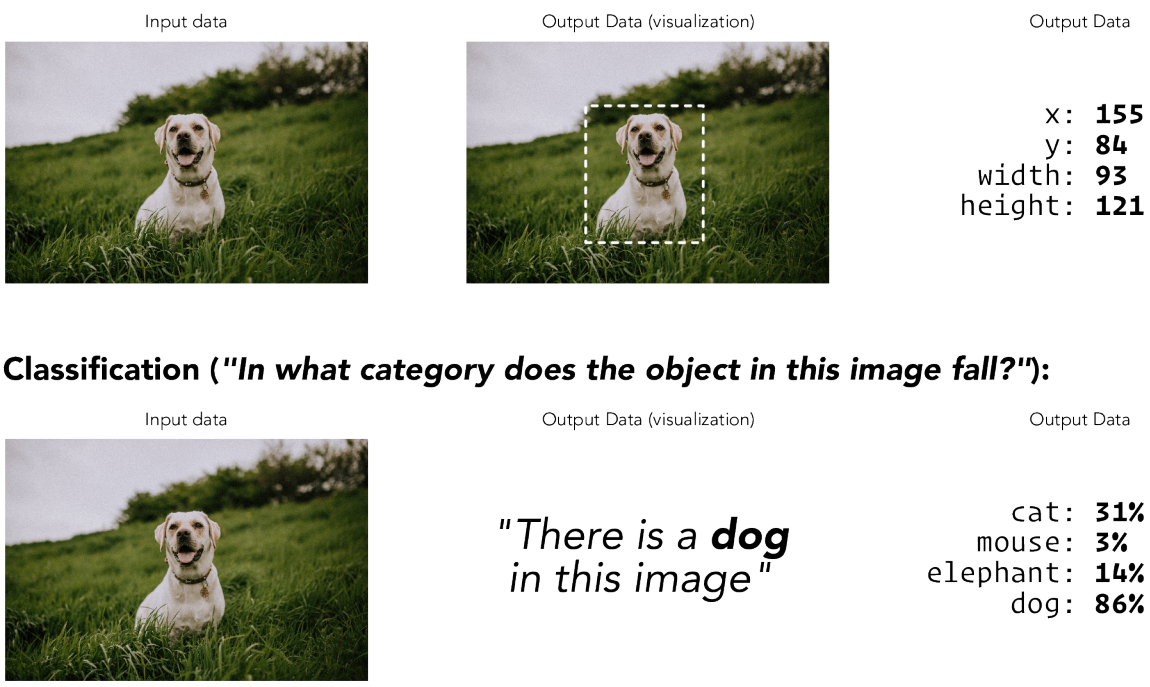

Figure 3.4 – A graphical representation of object classification and object detection. Detection hopes to locate a known object: in this figure, the model doesn't have to know what type of object it's looking for, it only knows that the object is bound by a certain box. The classification process lets the computer sort the image into categories; the output probabilities shown here let the computer conclude that there is likely a dog in the image. Often, detection and classification take place simultaneously, so a computer can look at an image and be able to say not just what is in the image, but where those different objects are as well. Note that any object the computer needs to be able to recognize needs to be known to the model beforehand: in the classification example, the computer will have been trained with many examples of cats, mice, elephants and dogs—those are the "classes" of objects it will be programmed to recognize. Dog image from Mitchell Orr on Unsplash.

synthetic data generation can be used to fill in the gaps.

Chapter 4

Experimental Methodology

The experiment carried out in this project is meant to test the viability of IBSG synthetic data for training ANNs. As explained previously, there are many different design decisions that the researcher must make while generating an IBSG dataset. I plan to create and test IBSG datasets based on Stanford's Street View House Number (SVHN) dataset (Netzer et al. n.d.), a real dataset in which images of house numbers have been cropped from Google Street View, and the individual digits in the image have been annotated with their bounding box. Since this is a dataset built for locating objects in an image, it is a good candidate for IBSG.

In this experiment, we construct a multitude of training datasets using different compositing algorithms, different isolation and modification sets, and different combinations of real and synthetic data. We train many different neural networks with these training datasets, then compare their performance when they are tested on a real test dataset.[1] We then use this test data to draw conclusions about the usefulness of the IBSG method, and make suggestions about how an IBSG dataset should be designed. More specifically, we first take existing datasets built for training detectors and explore how the data could be modeled as an IBSG dataset. We then train neural networks having the same hyper-parameters (layer configuration and size) with different ratios of real and IBSG training data and examine whether or not the IBSG data increases or decreases the accuracy of the ANN. We use this data to draw conclusions about the usefulness of the IBSG method in training detectors.

[1] Testing on an artificial dataset would be less than helpful. We want to see if IBSG can be a drop-in replacement for training with real data, and there would not be a production scenario in which one would test against artificial data. We used the exact same training dataset for every model we trained.

4.1 Controlled Variables

Beyond the scope of this project is designing a neural network specifically for object detection, especially a neural network made specifically for IBSG training datasets. It could be reasonably hypothesized that one could tune network hyper-parameters for IBSG-specific training data; however, since this project aims to test IBSG as a drop-in replacement for real training data, we instead use the Tensorflow Object Detection API, (Huang et al. 2017), a project from Google Research that aims to create a machine learning model "capable of localizing and identifying multiple objects in a single image," according to the project's mission statement. For each model trained, we employ the same computational setup for training and testing. We train the models usingon Bowdoin's HPC Grid for 50,000 training iterations, and use the same neural network framework—Tensorflow 1.12 (Martín Abadi et al. 2015)—throughout. Any further configurations remain fixed across different models; the only differences between the trained models will be the training dataset used.

4.2 Dependent Variables

Tensorflow includes a built-in data visualization program, known as TensorBoard, that many of the final results are drawn from. Machine learning models can register different metrics with TensorBoard, and these metrics get tracked and displayed to users. The Tensorflow Object Detection API has registered many metrics to TensorBoard; these metrics fall into two general categories. The first, *precision*, is a measure of how close the detected bounding box dimensions get to ground truth. The second, *recall*, measures how many bounding boxes were successfully found.[2] For example, a model could have low recall and high precision if it only found a few of the objects it should have, but the objects it did find had bounding boxes very close to the ground truth data.

[2] More explanation of precision and recall can be found in Chapter 5.

4.3 Independent Variables

To test the viability of IBSG is to mimic different use cases and different ways of creating an IBSG dataset and see how well they work. We designed three different compositing algorithms based on different naïve observations of SVHN data, in rough order from simplest to most complicated in terms of implementation time. The first algorithm assumes no modifications, and uses a short list of isolations. It also computationally generates the isolation image, drawing nothing from real data. The second algorithm assumes the same list of isolations, but introduces more (and varied) modifications. The third algorithm, which comes in multiple varieties, looks fairly different. It draws its isolation examples from photoshopped versions of real data. The differences between the varieties—along with the compositing algorithms for each type of SVHN data—is explained below.

4.4 Synthetic SVHN-mimicking Data Generation

The SVHN dataset comes in two formats: one with full numbers (meant to train locators) and one with cropped digits (meant to train classifiers). Since we had already performed preliminary experimentation with the Tensorflow Object Detection API, we used the full number dataset (Figure 4.1) for our experiment so that we could use the same API, which we had spent time setting up and testing. The SVHN dataset contains a test set (with 13,068 images and 26,032 digits), a train set (with 33,402 images and 73,257 digits), and an extra set with 531,131 that remained unused.[3] Furthermore, with each set of images, there is a Matlab struct file included that provides data for the bounding boxes of each digit in the image. To prepare the data, I extracted the bounding box data from the Matlab struct and prepared a CSV. From that, we generated two `.tfrecord` files, which can be input as test or train data for the Tensorflow Object Detection API.

[3]The data collection process for the SVHN dataset was partially automated, according to the researchers who built it. The image collection algorithm used a threshold that was lowered significantly for collecting the extra set — this is how the researchers were able to get so many images. Because of that, they are deemed easier for detection, and I avoided using them based on this description so as to not make the models any less accurate than they had to be.

Figure 4.1 – A sample of real data from the SVHN dataset. Blue boxes around each digit are not part of the dataset, they are visualizations of the annotations that come with the dataset.

With the real data prepared, we set about making synthetic data that looked similar to the real images, but could be generated programatically. Each "type" of synthetic data has an associated Python script that generates a number of images, as well as a CSV that describes the location of each digit within each image. Using the same process as the real data, we can create `.tfrecord` files from the synthetic data.

Different types of data have different balances of realism, possible variety, and compositing program creation time. With these differences, we hope to glean more information about what kind of synthetic data needs to be created for successful models, where the best data falls in that balance, and what factors surrounding the data creation process determine where along that balance a dataset lies.

4.4.1 Type A

The first synthetic dataset created was the most naïve. We identified three isolations within the SVHN dataset and used them to generate type A scenes: brick patterns, entryway façades, and house numbers. In addition to those isolations were mild modifications, such as matrix transformations and color manipulation. The compositing algorithm took a random picture from a bank of backgrounds and performed random 3D transformation and color manipulations on it. It then randomly chose a 1-4 digit house number, a random font from a bank, and a random color (either white or black) and drew that number in that font and color. The number was resized, had a 3D transformation performed, and was composited on top of the background.

Images from type A were highly unrealistic. Since the text was computer-drawn, there was aliasing and other composition artifacts plaguing the image.[4] The compositing algorithm was not advanced enough to create contrast between the text and numbers, leading to scenarios in which white text was placed on top of pri-

[4]This aliasing almost certainly came from the downsampling applied to the images. Images had to be placed along pixel boundaries in the image manipulation framework used, which would reduce the number of options our random number generator could choose, therefore reducing the entropy. All type A and type B images, therefore, were composited on a canvas ten times larger than the resulting image, and was then downsampled at the end of the composition.

Figure 4.2 – A sample of our Type A data. Digits are composited on top of brick textures or house façades. Image dimensions are randomly sampled from SVHN images.

Figure 4.3 – A closer view of selected Type A images. Note the white text on top of the white background in the bottom image: this type of naïve compositing meant that realism was drastically lower than the real data.

marily white background plates. In examining these images, we determined that they would be the lowest end of the realism spectrum — we didn't want any other datasets to be any less realistic than these.

4.4.2 Type B

The second compositing algorithm, based on that for type A, was meant to solve some of the contrast issues from the first set. In reexamining the original dataset, we noticed that there were often colored boxes that created contrast between the digit itself and the wall surface. We posited that this contrast was an integral part of putting up a house number; since the number should be visible by postal employees, delivery drivers, and emergency services, we felt it could be argued that the contrast was an important feature of the number itself, and added it as a modification that should exist among the two isolation layers.

On top of the background layer, the compositing algorithm for type B introduced a shape layer. This shape was filled with a dark color with a random hue, faded to 50% transparency, then transformed using a random matrix transforma-

Figure 4.4 – A sample of our Type B data. More 3D transformation took place, and a layer was added between the numbers and the background images to enforce contrast.

tion. The number was generated similarly as in type A; however, it was limited to a light color with a random hue. The number and shape were sized such that the number always fit within the shape.

While the type B images were similarly unrealistic, we hoped that the added contrast would aid the computer in understanding what a digit would look like; in other words, we were hoping that limiting the contrast would add enough realism to improve the testing results. Otherwise, however, the type B images suffered from the same aliasing, the same lack of realism from using a specific set of fonts, and the same feeling of being computer generated.

4.4.3 Type C1

Type C images were a significant departure from types A and B, and required several different iterations.[5] Type C images are generated from extractions of real training data, simulating a scenario in which researchers have access to a small amount of

[5]Furthermore, because we got significantly better results (discussed later) with type C images, we decided to pivot our data generation methods to focus only on type C for the rest of the project.

Figure 4.5 – A sample of our Type C1 data. Digit images have been extracted from real SVHN data and have been composited into the center of backgrounds extracted from real SVHN data. Type C data on the whole performed much better than types A and B data.

training data but in which it would be infeasible for the researchers to label a large amount on their own. For all three iterations of type C data, we used Photoshop to extract cropped digits and backgrounds from real data. These backgrounds and digits, therefore, would contain the same artifacts, noise, contrast, and other properties that the real data had. The image, whose sizes had been generated with a random number generator in the previous types, were now all the exact sizes of the backgrounds used. We were concerned about how well the digits would blend into the background (rectangles of high contrast might have the potential to distract the neural network), but waited to see the results before deciding to fix that.

For type C1, digits and backgrounds were extracted from 20 images of real training data, yielding roughly 50 digit examples and 17 background examples.[6] Between 1 and 4 digits were randomly selected, resized, stacked next to each other horizontally, and placed in the center of a randomly chosen background. These images, especially when viewed in actual size, looked much more similar to real data than

[6]Not all images were suitable for background extraction, since the background extraction process required some visual material that could be clone-stamped over the numbers.

Figure 4.6 – A sample of our Type C2 data. C2 introduced random scaling and placement, in addition to more isolation examples.

the fully synthetic images. The compositing algorithm took much less time to write (since there was not a large amount of image generation and manipulation taking place); however, the process of extracting digits took roughly an hour, somewhat offsetting the shortened programming time.

4.4.4 Type C2

Hypothesizing that the computer was learning to guess that there were some digits in the middle of images, we decided to split up and resize the individual digits for type C2. While the digits were still in the middle of the image, now they were vertically staggered and had randomly chosen heights. The bounding boxes of the images, therefore, were less predictable. Furthermore, we extracted digits and backgrounds 20 more images of real training data, introducing more entropy into the random data.

4.4.5 Type C3

In designing the type C3 compositing algorithm, we hoped to include some of the manipulations from types A and B into type C. Digits and backgrounds, in the C3 algorithm, were transformed using a matrix transformation. They had new color op-

Figure 4.7 – A sample of our Type C3 data. Here, digit images were spaced apart and rotated randomly.

erations applied to them. Finally, we took digits and backgrounds from yet another 20 images of real training data.

4.5 Training Models

Ultimately we designed seventeen training data configurations using these five synthetic data types along with the real training dataset. We included different quantities of training data in different sets to quantify how much quantity affected the results. For each training dataset prepared, we trained five different models; since there is an element of randomness in the training process, we wanted multiple trials to get more accurate results. The training datasets designed are:

- Exclusively real data: **10,000 images** and **60,000 images**
- Synthetic data type A: **10,000 images** and **60,000 images**
- Synthetic data type B: **10,000 images** and **60,000 images**
- Synthetic data type C1: **4,000 images**[7]

[7] C1 was only trained with 4,000 images because it acted as a preliminary test: we hoped to see how

- Synthetic data type C2: **10,000 images** and **60,000 images**
- Synthetic data type C3: **10,000 images** and **60,000 images**
- Blends:
  - 10% type B, 90% real: **30,000 images**
  - 25% type B, 75% real: **30,000 images**
  - 50% type B, 50% real: **30,000 images**
  - 75% type B, 25% real: **30,000 images**
  - 90% type B, 10% real: **30,000 images**
- Finally a special real dataset prepared using only the 60 images used in the process of making the type C data.

The configurations for TensorFlow, the training script, and the Object Detection API can be found on Github. We ran each training step for 50,000 iterations, which was the number suggested by the developers of the API as the default value, to be changed if it was not enough.[8] Furthermore, in preliminary testing we learned that training an object detection model from scratch would take weeks or months given the hardware we used. At the suggestion of the Object Detection API developers, we trained our custom models from an existing baseline trained model: the "SSD with Inception v2 configuration for MSCOCO Dataset" (Lin et al. 2014) model.[9] This baseline was not set up to detect digits in any way, but was already able to turn areas in an image into bounding box output. Using this baseline reduced model training time to roughly 24 hours on Bowdoin's HPC grid.

---

significantly different the results would be when switching dataset design methodologies so drastically. When C1's performance showed to be drastically improved (results discussed later in Chapter 5), we moved onto C2 and C3, which were more robust, realistic versions of the same dataset.

[8] Found at `https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/running_locally.md`

[9] Found at `https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md`

Chapter 5

Results and Analysis

Precision and recall are the two foremost metrics of ML-based object detectors; each measures a different element of what one might intuit as "correctness." A model with high precision but low recall will be able to predict bounding boxes that are very close to ground truth values, but might not be able to detect all the bounding boxes that should be found in an image. A model with high recall but low precision will be able to draw the right number of bounding boxes, but those boxes will not be in the right place. Generally, the two measures balance each other out, and models can be plotted along a *Precision-Recall (PR) Curve*, in which as one metric rises, the other falls. Researchers working to train a specific model often determine which metric is more important for the specific application and adjust thresholds to find a balance along the PR curve that can get the intended results (Oksuz et al. 2018).

5.1 Model Accuracy Metrics

The TensorBoard dashboard for the Tensorflow Object Detection API exposes the precision and recall metrics for each trained model to users. In addition to presenting average recall and average precision, the dashboard shows ten other metrics: five relating to precision and five relating to recall. Since the baseline model we trained our models from was trained using the COCO dataset (Lin et al. 2014), the COCO project lists the different metrics shown in TensorBoard:

Average Precision (AP):

- `AP`: AP at IoU=.50:.05:.95 (primary challenge metric)
- $\texttt{AP}^{\texttt{IoU=.50}}$: AP at IoU=.50 (PASCAL VOC metric)
- $\texttt{AP}^{\texttt{IoU=.75}}$: AP at IoU=.75 (strict metric)

AP Across Scales:

- $\texttt{AP}^{\texttt{small}}$: AP for small objects: area < 322
- $\texttt{AP}^{\texttt{medium}}$: AP for medium objects: 322 < area < 962
- $\texttt{AP}^{\texttt{large}}$: AP for large objects: area > 962

Average Recall (AR):

- $\texttt{AR}^{\texttt{max=1}}$: AR given 1 detection per image
- $\texttt{AR}^{\texttt{max=10}}$: AR given 10 detections per image
- $\texttt{AR}^{\texttt{max=100}}$: AR given 100 detections per image

AR Across Scales:

- $\texttt{AR}^{\texttt{small}}$: AR for small objects: area < 322
- $\texttt{AR}^{\texttt{medium}}$: AR for medium objects: 322 < area < 962
- $\texttt{AR}^{\texttt{large}}$: AR for large objects: area > 962

While the COCO project defines AP as its primary metric, we found in preliminary tests that the main detection mechanism IBSG-trained models would suffer from was recall: if the training data was unrealistic, the model would fail to detect anything (instead of overdetecting, which would be the primary reason to measure results using precision). Therefore, we measured our results using the $\texttt{AR}^{\texttt{max=100}}$ (Average recall given 100 detections per image) metric. Ultimately, however, the metric used to measure model success (among the several metrics given as options in the TensorBoard dashboard) is relatively unimportant, since nearly all of them gave the same relative results: the relative success of each model did not seem to differ between different metrics.[1]

TensorBoard presented data on a per-model basis, displaying each metric plotted as a function of training time; an example of this plot is shown in Figure 5.1[2] As described in the figure's caption, the data given here is not terribly useful in determining a model's success; however, it does provide some insight into how the data should be processed. For these models and every other model trained, the recall stops increasing at around 35,000 training steps, marked in the figure by a red

[1]The exceptions to this rule are the metrics measuring large objects. These metrics ultimately ended up just measuring noise, since there were not enough large objects in the test dataset to garner meaningful results.

[2]Measurements were collected on a scale of the number of training iterations, rather than strictly temporal. Because two different types of GPUs were used to train the models, the duration between start and end would vary across different training sessions. In this way, the number of training steps became the barometer for training completeness, rather than duration.

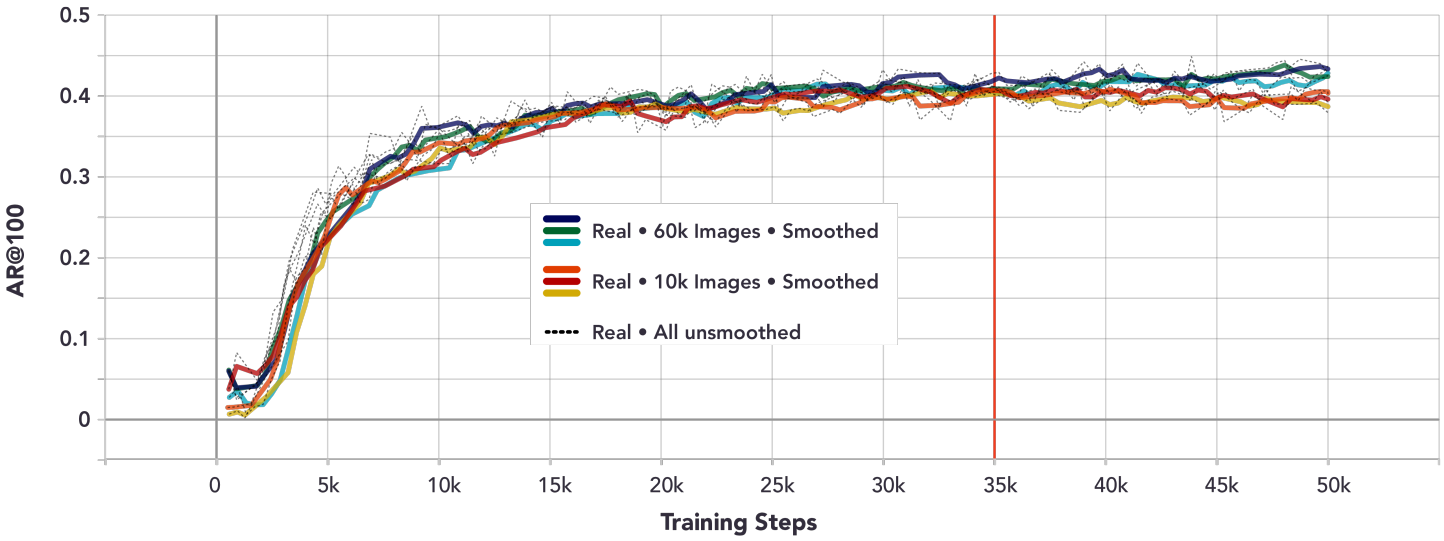

Figure 5.1 – Average Recall (`AR@100`) per real-data-trained model as training progresses through 50,000 steps. The models represented in this figure represent three trials each of a 60,000-image training set and a 10,000-image training set. The solid colored lines have had smoothing applied to them (default TensorBoard smoothing with a coefficient of 0.6), and the dotted black lines represent the corresponding unsmoothed data. The unsmoothed data is close enough to the smoothed data (and it makes it easier to see long-term trends, like the subtle gradual split between the 10,000-image data points and the 60,000-image data points) that any subsequent visualization of recall over training step will be shown with this smoothed data. The metric that best defines the success of each individual model involves the `AR@100` value towards the end of training—the recall values before the model has been fully trained are irrelevant since they will be low by definition. Instead, in many subsequent visualizations, a model will be displayed as a box-and-whisker plot of the `AR@100` values of training steps greater than 35,000 for all trials trained with the same training dataset.

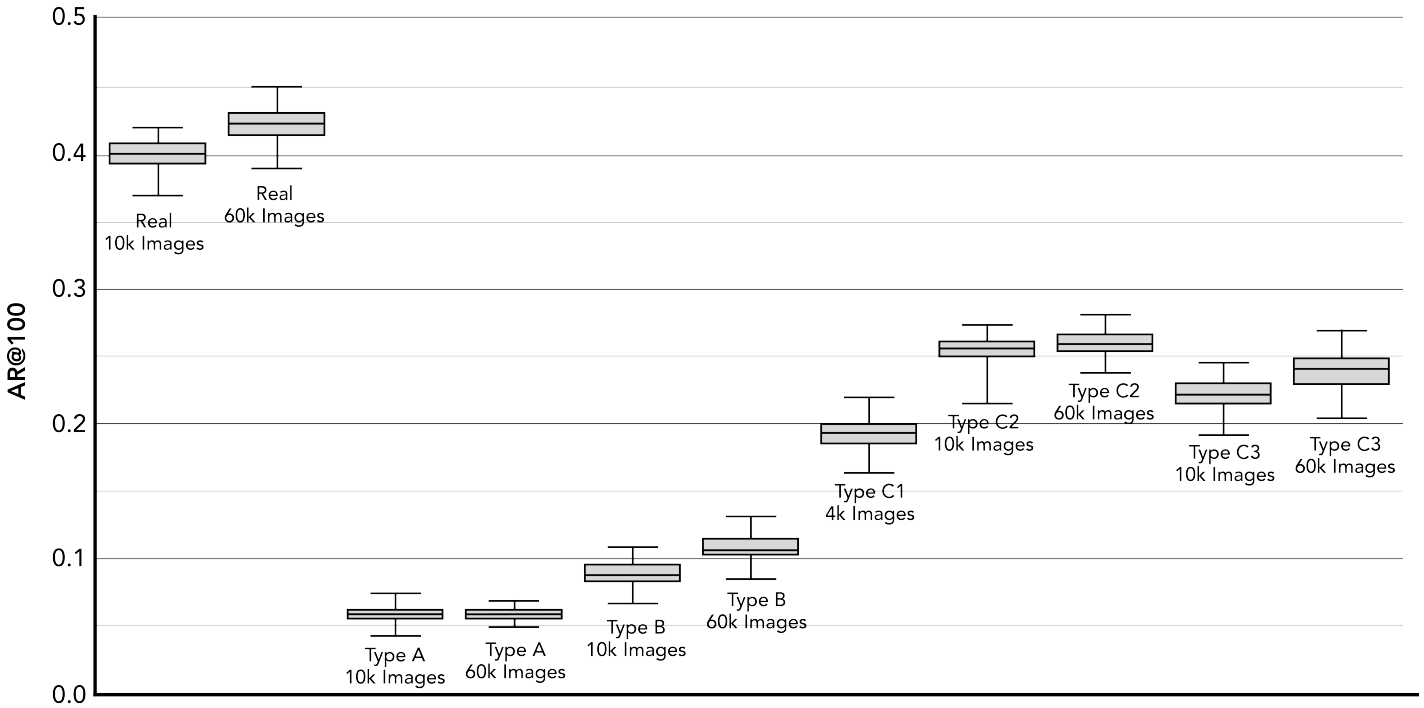

Figure 5.2 – Percentile values for Average Recall (`AR@100`) for models trained with different training datasets. All values that form each box-and-whisker plot is from training steps over 35,000. The data used to create this (and other) box and whisker plots in this section can be found in Appendix A.

line. It can be assumed that by this point in the training process, we have reached our final accuracy level. We can average the post-35,000, TensorBoard-smoothed `AR@100` values for each trial of a single training dataset to get a single output accuracy value (along with statistical error) for all models trained with that specific training dataset. These values and errors for all non-blended training datasets are shown in Figure 5.2.

5.2 Results

The first two models trained used completely real training data. This served as our baseline: if any synthetic-data-trained models performed at the level of the real data,

that synthetic dataset would be considered successful. For our setup, this success barometer was set at roughly 0.41, since the real-data-trained models had a recall value slightly lower and slightly higher than that, for the 10,000-image set and the 60,000-image set respectively.[3]

None of the synthetic-trained models were able to match the real-trained models in recall. The most performant models, trained with the Type C2 dataset, were still significantly worse than the real-trained models, with a median value of roughly 0.26 compared to the real data's 0.41. With this, we can see that regardless of how realistic or varied the training data is, randomly generated synthetic data made using the IBSG method cannot match real data in performance, even if the isolations themselves are created from samples of real data, as was the case for all Type C datasets.

As was expected, putting more effort into simulating realism in the datasets increased performance. The contrast layer that Type B introduced almost doubled the model performance; the switch to real-data-extracted isolation samples doubled performance again. These increases in realism were much more important to performance than variety. While the 60,000-image models constantly outperformed the 10,000-image models, the median recall would only increase by a few percentage points, at maximum. The Type A models saw almost no effect, indicating that if a model is fed with faulty training data, no increase in variety can help: the increased quality of the training data brings about more improvements than the increased quantity (and, by proxy therefore, the increased variety).

In addition to the completely synthetic datasets used to train models, we also trained models using blends of real and Type B data (Figure 5.3). While the blended models performed between the completely real and completely synthetic datasets, the linear relationship between performance and blend ratio that might be expected was not shown in the final outcome. Instead, the blended datasets congregated around a single value between the real and synthetic, and changing the ratio seemed

[3] 41% recall is not considered an objective success; however, remember that this experiment was not designed to see how performant a model we could build. Rather, we wanted to see if synthetic data could be *as successful* as real data: in this scenario, with this object detection API and this dataset, success was defined as a relatively low value.

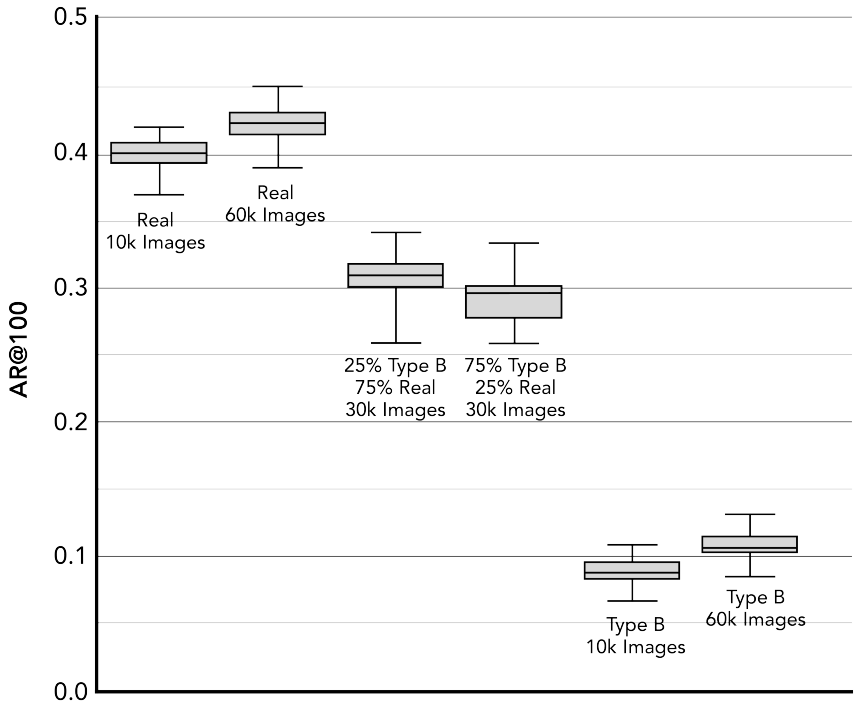

Figure 5.3 – Percentile values for Average Recall (`AR@100`) for models trained with real images, type B images, and different blend ratios of real and type B images. Even with a drastic reduction in the amount of real data, as long as there is still some real data in the training set the results seem to have an accuracy close to that of the real data.

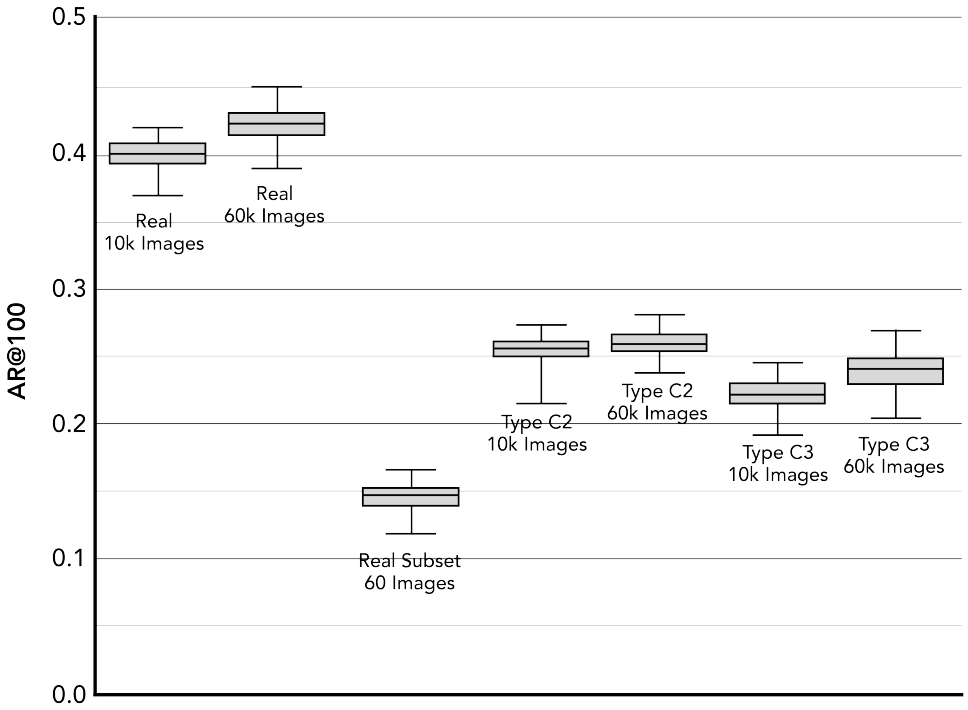

Figure 5.4 – Percentile values for Average Recall (`AR@100`) for models trained with real data, type C data, and the subset of real data (60 images) used to create type C data. The IBSG process is able to turn a small amount of original data into a large, varied, but still realistic training dataset, increasing the recall even with the same original data source.

to have little effect on the model's performance. It is possibly the case that the real data in both datasets brought about most of the model's success, and that the addition of different amounts of synthetic data contributed little to the model's knowledge.

The last type of model trained used a small subset of the real dataset: only the sixty images used in the process of making the Type C data. This training dataset was prepared with the hope of understanding how the IBSG process itself affected performance: comparing the Type C models to the model trained on the subset would help extract the added "value" gained from the IBSG process, providing insight into how useful IBSG is as a whole.

Figure 5.4 presents the compared results of real, real-subset, and Type C trained

models. The real subset models performed significantly worse than the Type C models, indicating that the IBSG process introduces enough variety into the training data that performance is improved: in other words, even with the same sources of pixels as input, the variety gained by the IBSG process outweighs the loss in realism that same process introduces. This result, along with the blended result, shows that IBSG can be a realistic solution if researchers have only a little bit of training data and are willing to sacrifice some performance.

5.3 Analysis

While the IBSG process was not completely unsuccessful in our testing, it is not as robust of a training-data generation method as we had originally hoped. As seen by the prior research, fully synthetic training data can be used to create performant ML models. The results from this project show, however, that IBSG is not the best way of doing so, since the realism afforded by purely synthetic datasets built using IBSG strays so far from the actual real data that performance is greatly impacted. Using IBSG to create datasets from nothing, as shown by the failure of the Types A and B trained models, is a poor idea; more advanced rendering techniques like 3D modeling are necessary to match the realism of the real world.

IBSG has potential as a way of generating large datasets based on small, real-world datasets. The blending results and Type C results show that when datasets have enough elements of realism, the IBSG process can add variety in such a way that augments the original real data. If given a small dataset, a researcher could viably use IBSG to make it larger and more robust, improving the success rate of her model with only the work needed to extract isolations and blend them together.

IBSG does a poor job of creating something from nothing, since a small Python program cannot accurately model the realism of the natural world. Even in a limited scenario like generating extracted house numbers, the lighting, placement, and quality of isolations needed to build up a realistic scene is more than a single researcher can model on his own, making the IBSG process too involved. To suggest that IBSG datasets created from scratch, like Types A and B were, can reduce the time and effort a machine learning researcher must put in to train a successful model would

be false. With isolations extracted from real data, however, the realism can already exist, and IBSG can be used as a producer of variety, rather than a producer of realism. With IBSG taking care of this variety, a researcher can still create a successful model without having to label training data for hours on end. In these scenarios, when a researcher wants to create a detector or classifier and does not have a robust, large training dataset (but *does* have access to small samples of training data), the IBSG process can be a valuable tool for turning a small, unlabeled dataset into a large, labeled one.

5.4 Future Work

Because IBSG is a generalized process, it can be tested in many different situations. In preparing for this project, we considered comparing IBSG's performance using many different datasets as references, instead of just SVHN. One might build an IBSG dataset based on the COCO dataset and see how well it performs. IBSG, as described, only works when researchers are trying to create training data for detection or classification purposes. In this project, we were only able to test detection: we were finding digits' bounding boxes within an entire image. One might also test IBSG's ability to train classification models, changing the goal from detecting digits within an image to classifying images of digits (in other words, the model would figure out which digit is being displayed instead of figuring out where any arbitrary digit is). This approach is similar to the work of Jaderberg et al., who seemed to have more success with completely synthetic data.

IBSG also has the potential for use in audio. Because IBSG can be used in scenarios in which a model is trying to detect or classify a piece of data obscured within a scene, one could try using audio isolations to build up an audio scene. Detection models would try to find a certain noise, like a whistle, within a longer-duration sound file; classification models might try to detect which word a person says.

Even within the context of the SVHN dataset, researchers could continue exploring the viability of IBSG training datasets. One might consider building a neural network specifically made for IBSG datasets. Exploring different ways of generating scenes, using more dimensions of randomness and more realistic compositing tech-

niques, might improve the quality of fully synthetic datasets. Finally, researchers might dive deeper into one of the metrics used to create synthetic datasets, like the number of isolations used, the blend ratio, or others to create a stronger recommendation for the exact metrics one should use while creating an IBSG dataset.

Chapter 6

Conclusion

This project lowers the barrier of entry into machine learning, making it possible for single researchers to spend less time labeling data and more time understanding their dataset and training a model. As machine learning becomes more popular, especially within the context of detection and classification of images and sound, it becomes important to make the technology accessible to as many people as possible so that everyone, not just the most mathematically inclined, can use technology to better the world. In a small way, IBSG gives smaller-scale researchers, like undergraduates in colleges, the ability to use a machine learning solution for a problem that would be too hard to solve without ML. Personal experience has shown that problems exist with a fairly straightforward ML solution, but that couldn't be explored because of the lack of a high quality training dataset. These are the scenarios in which IBSG can make object detection machine learning more available in the research community.

Bibliography

Batista, Gustavo E. A. P. A., Ronaldo C. Prati, and Maria Carolina Monard (June 1, 2004). "A study of the behavior of several methods for balancing machine learning training data". In: *ACM SIGKDD Explorations Newsletter* 6.1, p. 20. ISSN: 19310145. DOI: 10.1145/1007730.1007735. URL: http://portal.acm.org/citation.cfm?doid=1007730.1007735 (visited on 04/19/2019).

Hattori, Hironori et al. (June 2015). "Learning scene-specific pedestrian detectors without real data". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Boston, MA, USA: IEEE, pp. 3819–3827. ISBN: 978-1-4673-6964-0. DOI: 10.1109/CVPR.2015.7299006. URL: http://ieeexplore.ieee.org/document/7299006/ (visited on 04/19/2019).

Huang, Jonathan et al. (July 2017). "Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Honolulu, HI: IEEE, pp. 3296–3297. ISBN: 978-1-5386-0457-1. DOI: 10.1109/CVPR.2017.351. URL: http://ieeexplore.ieee.org/document/8099834/ (visited on 04/19/2019).

Jaderberg, Max et al. (June 9, 2014). "Synthetic Data and Artificial Neural Networks for Natural Scene Text Recognition". In: *arXiv:1406.2227 [cs]*. arXiv: 1406.2227. URL: http://arxiv.org/abs/1406.2227 (visited on 04/19/2019).

Khungurn, Pramook and Derek Chou (2016). "Pose estimation of anime/manga characters: a case for synthetic data". In: *Proceedings of the 1st International Workshop on coMics ANalysis, Processing and Understanding - MANPU '16*. the 1st International Workshop. Cancun, Mexico: ACM Press, pp. 1–6. ISBN: 978-1-4503-4784-6. DOI: 10.1145/3011549.3011552. URL: http://dl.acm.org/citation.cfm?doid=3011549.3011552 (visited on 04/19/2019).

LeCun, Yann, Corinna Cortes, and Chris Burges (2019). *MNIST handwritten digit database*. URL: http://yann.lecun.com/exdb/mnist/ (visited on 05/16/2019).

Lin, Tsung-Yi et al. (2014). "Microsoft COCO: Common Objects in Context". In: *Computer Vision – ECCV 2014*. Ed. by David Fleet et al. Vol. 8693. Cham: Springer International Publishing, pp. 740–755. ISBN: 978-3-319-10601-4 978-3-319-10602-1. DOI: 10.1007/978-3-319-10602-1_48. URL: http://link.springer.com/10.1007/978-3-319-10602-1_48 (visited on 04/19/2019).

Martín Abadi et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

Mehrotra, Kishan, Chilukuri K. Mohan, and Sanjay Ranka (1997). *Elements of artificial neural networks*. Complex adaptive systems. Cambridge, Mass: MIT Press. 344 pp. ISBN: 978-0-262-13328-9.

Müller, Berndt, J. Reinhardt, and M. T. Strickland (1995). *Neural networks: an introduction*. 2nd updated and corr. ed. Physics of neural networks. Berlin ; New York: Springer. 329 pp. ISBN: 978-3-540-60207-1.

Netzer, Yuval et al. (n.d.). "Reading Digits in Natural Images with Unsupervised Feature Learning". In: (), p. 9.

Oksuz, Kemal et al. (July 4, 2018). "Localization Recall Precision (LRP): A New Performance Metric for Object Detection". In: *arXiv:1807.01696 [cs]*. arXiv: 1807.01696. URL: http://arxiv.org/abs/1807.01696 (visited on 05/16/2019).

Reisberg, Daniel (2016). *Cognition: exploring the science of the mind*. Sixth edition. New York: W.W. Norton & Company. 692 pp. ISBN: 978-0-393-93867-8.

Richter, Stephan R. et al. (Aug. 7, 2016). "Playing for Data: Ground Truth from Computer Games". In: *arXiv:1608.02192 [cs]*. arXiv: 1608.02192. URL: http://arxiv.org/abs/1608.02192 (visited on 04/19/2019).

Shrivastava, Ashish et al. (Dec. 22, 2016). "Learning from Simulated and Unsupervised Images through Adversarial Training". In: *arXiv:1612.07828 [cs]*. arXiv: 1612.07828. URL: http://arxiv.org/abs/1612.07828 (visited on 04/19/2019).

Weiss, Sholom M. and Casimir A. Kulikowski (1991). *Computer systems that learn: classification and prediction methods from statistics, neural nets, machine learning, and expert systems*. San Mateo, Calif: M. Kaufmann Publishers. 223 pp. ISBN: 978-1-55860-065-2.

Appendices

Appendix A

Table for Summary Data

Data Type	Min	Q25	Median	Q75	Max
Real (10,000 images)	0.367	0.390	0.398	0.405	0.417
Real (60,000 images)	0.387	0.411	0.421	0.428	0.448
Synthetic type A (10,000 images)	0.043	0.054	0.059	0.062	0.074
Synthetic type A (60,000 images)	0.049	0.055	0.059	0.062	0.068
Synthetic type B (10,000 images)	0.067	0.082	0.089	0.096	0.108
Synthetic type B (60,000 images)	0.085	0.103	0.107	0.115	0.132
Real (60 images)	0.119	0.138	0.147	0.152	0.165
Synthetic type C1 (4,000 images)	0.163	0.185	0.193	0.200	0.219
Synthetic type C2 (10,000 images)	0.215	0.230	0.255	0.261	0.272
Synthetic type C2 (60,000 images)	0.236	0.253	0.259	0.266	0.280
Synthetic type C3 (10,000 images)	0.192	0.214	0.220	0.229	0.251
Synthetic type C3 (60,000 images)	0.203	0.228	0.240	0.248	0.267
25%/75% Type B/Real (30,000 images)	0.259	0.299	0.308	0.316	0.340
25%/75% Real/Type B (30,000 images)	0.257	0.276	0.295	0.300	0.332

Table A.1 – Raw data for figures 5.2, 5.3, and 5.4.

Appendix B

Color Figures in Grayscale

Figure B.1 – (Figure 2.2): Examples of video game footage and the associated classification labeling from Richter et al. Different colored pixels in the right images indicate different human-labeled classifications.

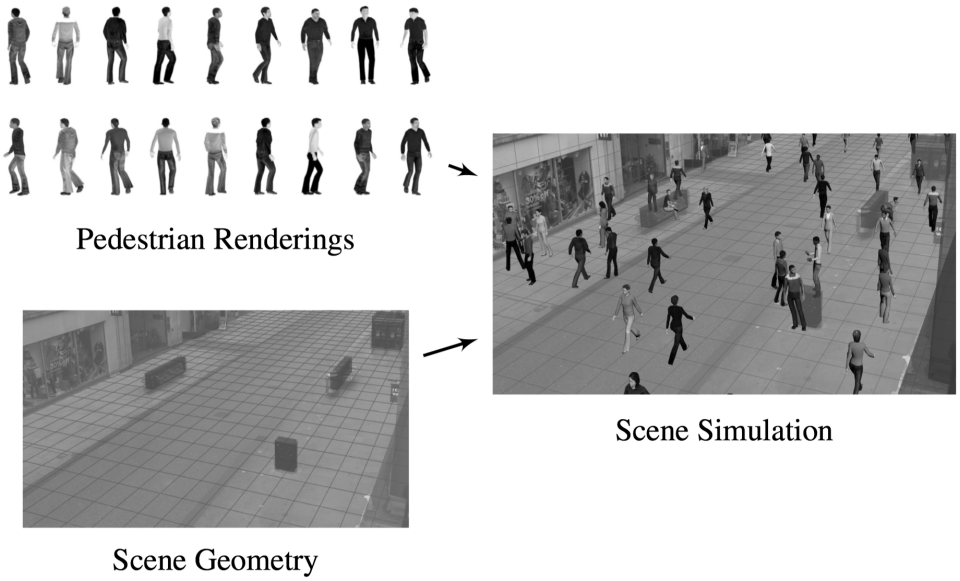

Figure B.2 – (Figure 2.3): Synthetic data used to train a person detector for a specific security camera vantage point, created by Hattori et al. The camera's viewpoint was rendered in 3D software and the renders were used to train the detector.

Figure B.3 – (Figure 2.5): Images of rendered characters in specific poses composited over 2D backgrounds, by Khungurn et al. The resulting two-dimensional images are used to train a model to recognize the three-dimensional pose the character is in.

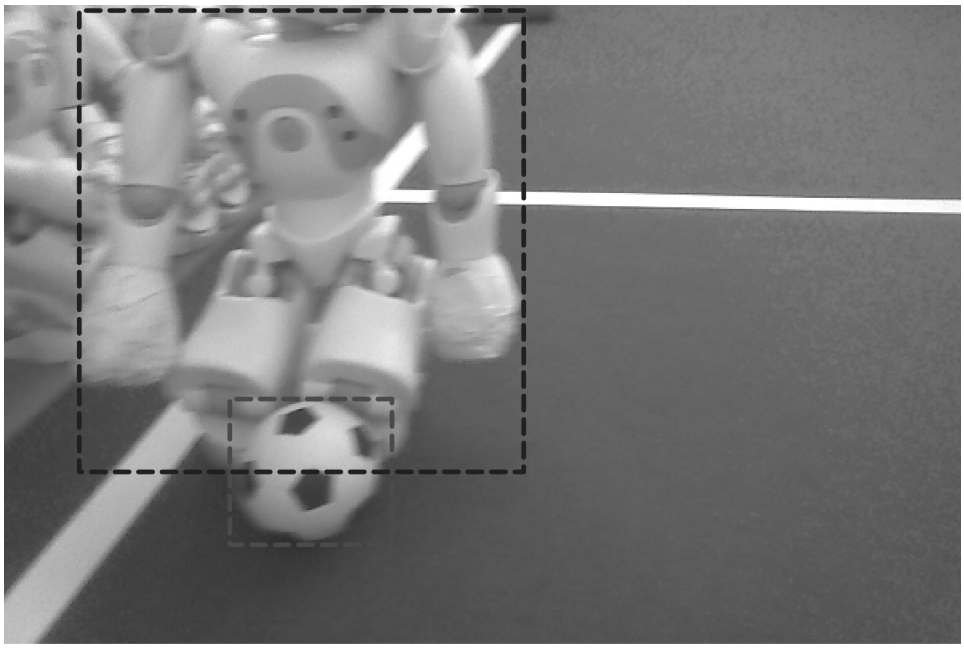

Figure B.4 – (Figure 3.1): An image collected by a robot in a robotic soccer field, showing the field in the background, an adversarial robot, and the soccer ball. A human looking at this image might describe three isolations: the background, the robot, and the ball. These isolations have been highlighted here with dashed boxes indicating how the human object detection process would segment the image.

Figure B.5 – (Figure 3.2): Images of balls, robots, and field backgrounds extracted in photo manipulation programs or taken independently. These isolation examples fall into the categories found by the human in Figure 3.1.

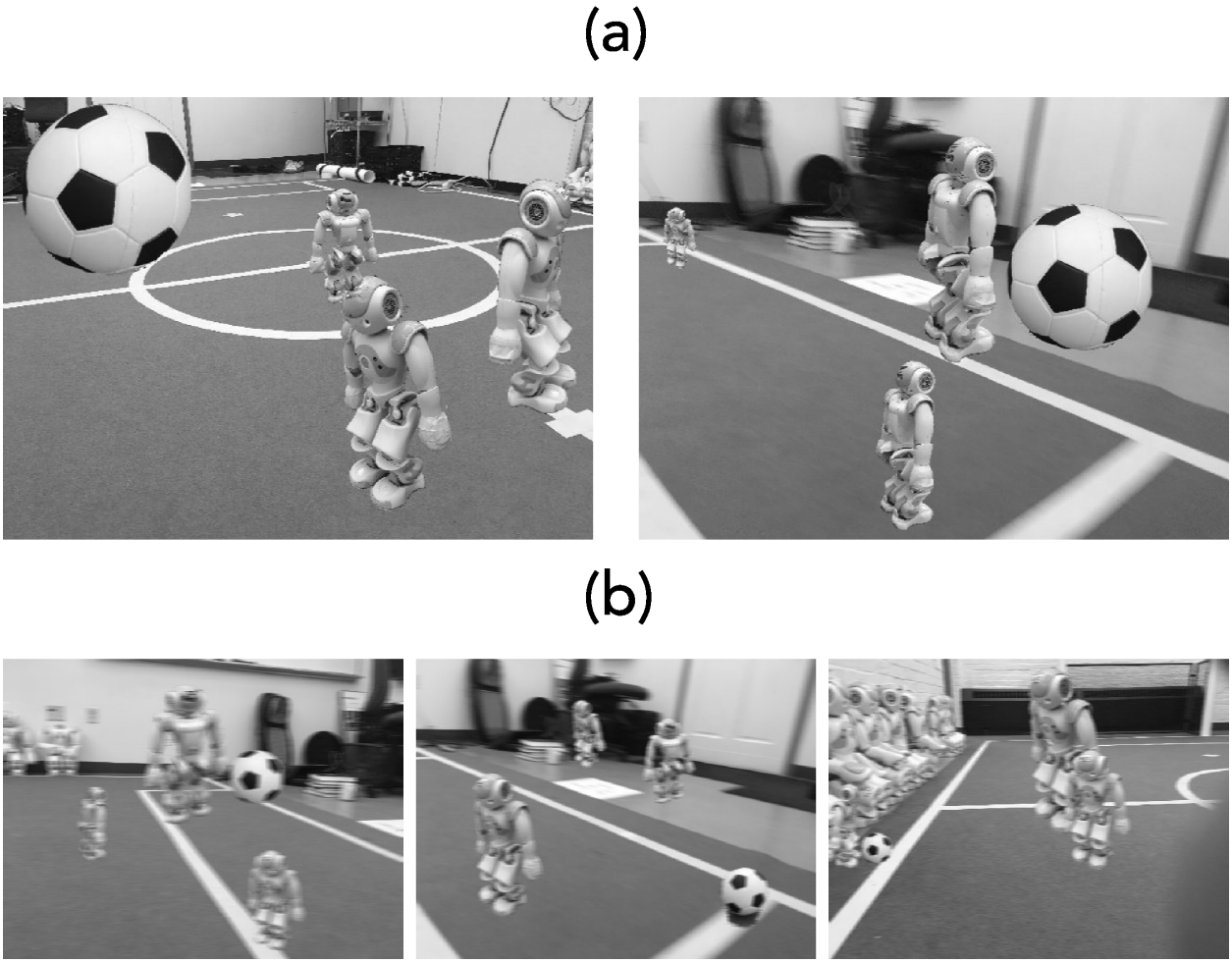

Figure B.6 – (Figure 3.3): Images programmatically generated by randomly combining the extracted isolation examples from Figure 3.2. Subfigure (a) shows images without any modification: the ball and robots are randomly scaled and placed. Subfigure (b) shows images that have motion blur added to the output image, as is typical of images taken from a roving robot. These modifications serve to enhance the realism of the final image, making the model perform better on real testing data.

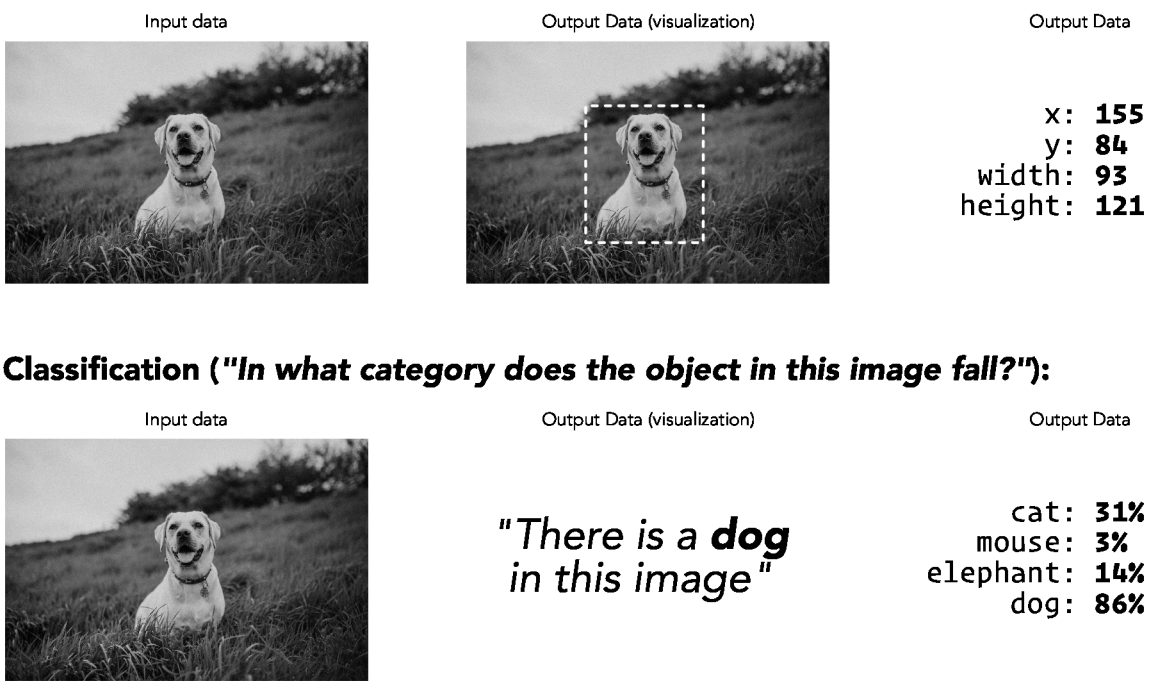

Figure B.7 – (Figure 3.4): A graphical representation of object classification and object detection. Detection hopes to locate a known object: in this figure, the model doesn't have to know what type of object it's looking for, it only knows that the object is bound by a certain box. The classification process lets the computer sort the image into categories; the output probabilities shown here let the computer conclude that there is likely a dog in the image. Often, detection and classification take place simultaneously, so a computer can look at an image and be able to say not just what is in the image, but where those different objects are as well. Note that any object the computer needs to be able to recognize needs to be known to the model beforehand: in the classification example, the computer will have been trained with many examples of cats, mice, elephants and dogs — those are the "classes" of objects it will be programmed to recognize.

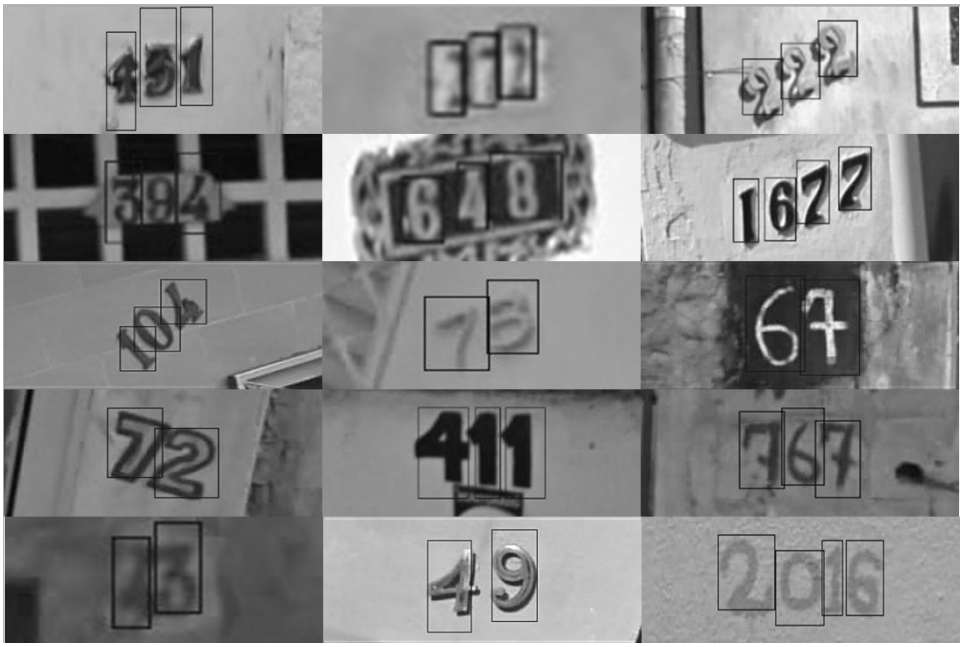

Figure B.8 – (Figure 4.1): A sample of real data from the SVHN dataset. Blue boxes around each digit are not part of the dataset, they are visualizations of the annotations that come with the dataset.

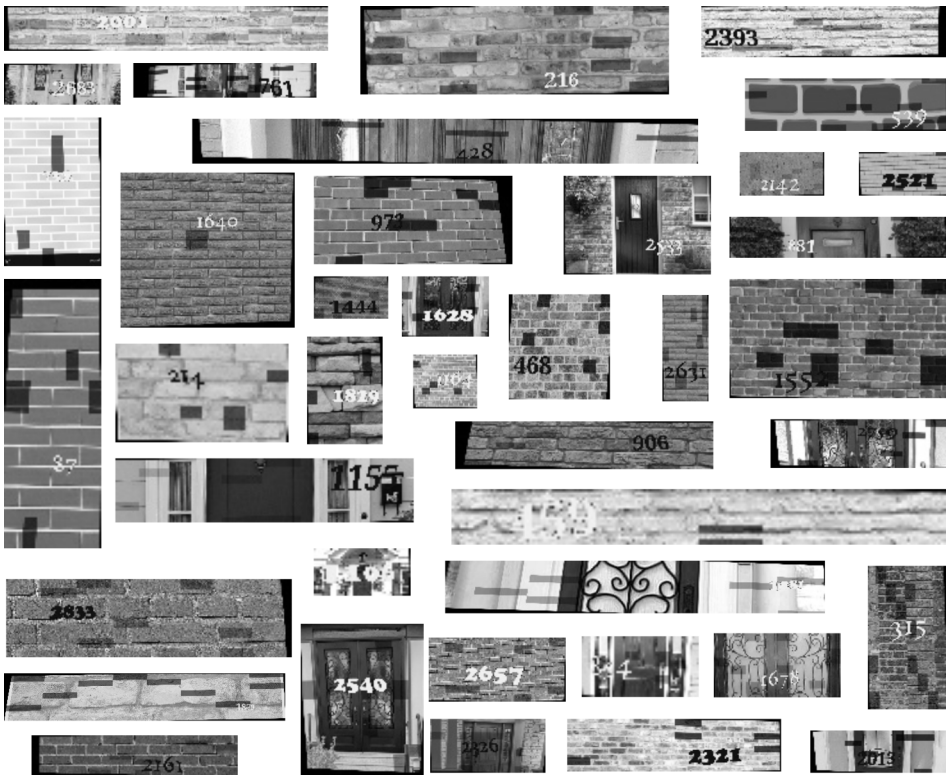

Figure B.9 – (Figure 4.2): A sample of our Type A data. Digits are composited on top of brick textures or house façades. Image dimensions are randomly sampled from SVHN images.

Figure B.10 – (Figure 4.3): A closer view of selected Type A images. Note the white text on top of the white background in the bottom image: this type of naïve compositing meant that realism was drastically lower than the real data.

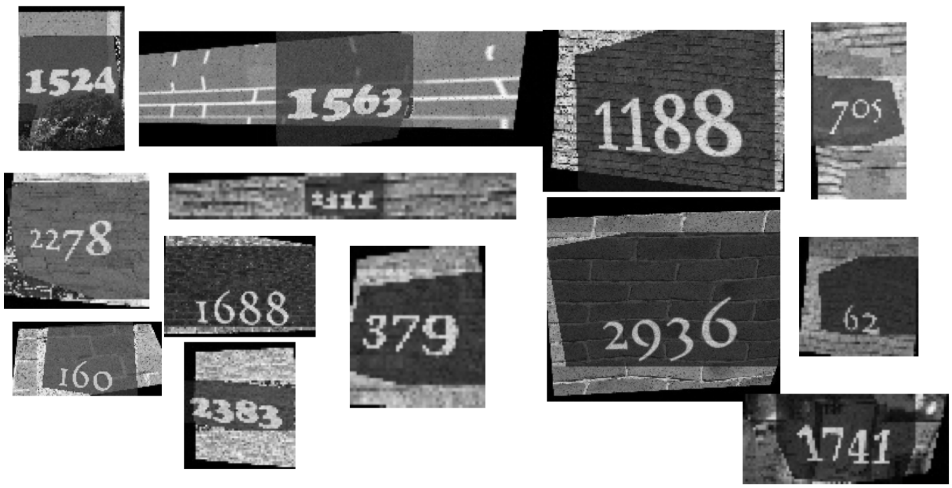

Figure B.11 – (Figure 4.4): A sample of our Type B data. More 3D transformation took place, and a layer was added between the numbers and the background images to enforce contrast.

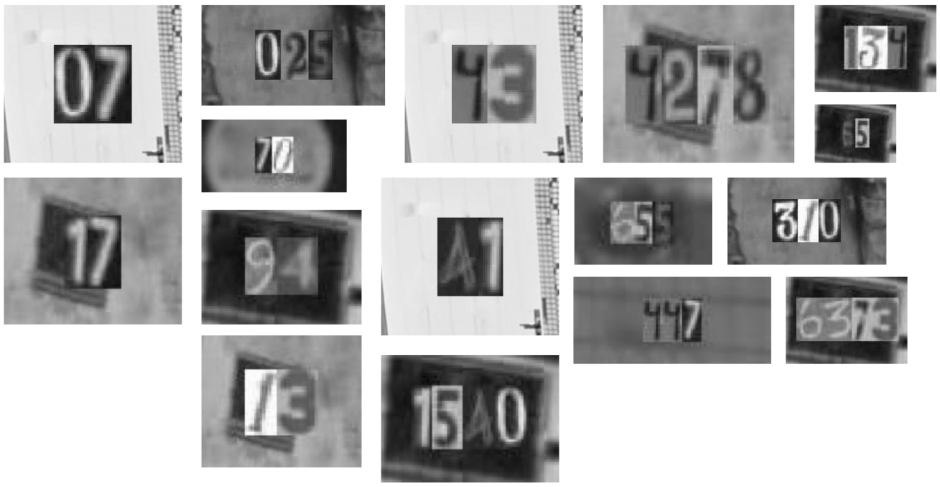

Figure B.12 – (Figure 4.5): A sample of our Type C1 data. Digit images have been extracted from real SVHN data and have been composited into the center of backgrounds extracted from real SVHN data. Type C data on the whole performed much better than types A and B data.

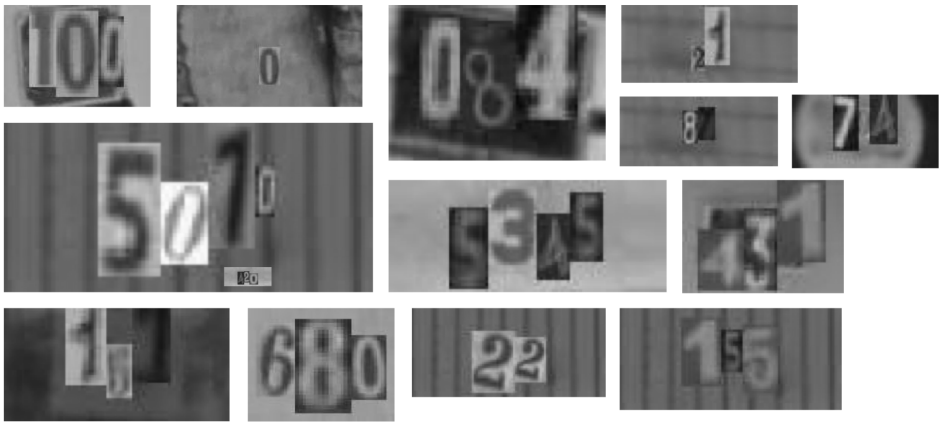

Figure B.13 – (Figure 4.6): A sample of our Type C2 data. C2 introduced random scaling and placement, in addition to more isolation examples.

Figure B.14 – (Figure 4.7): A sample of our Type C3 data. Here, digit images were spaced apart and rotated randomly.

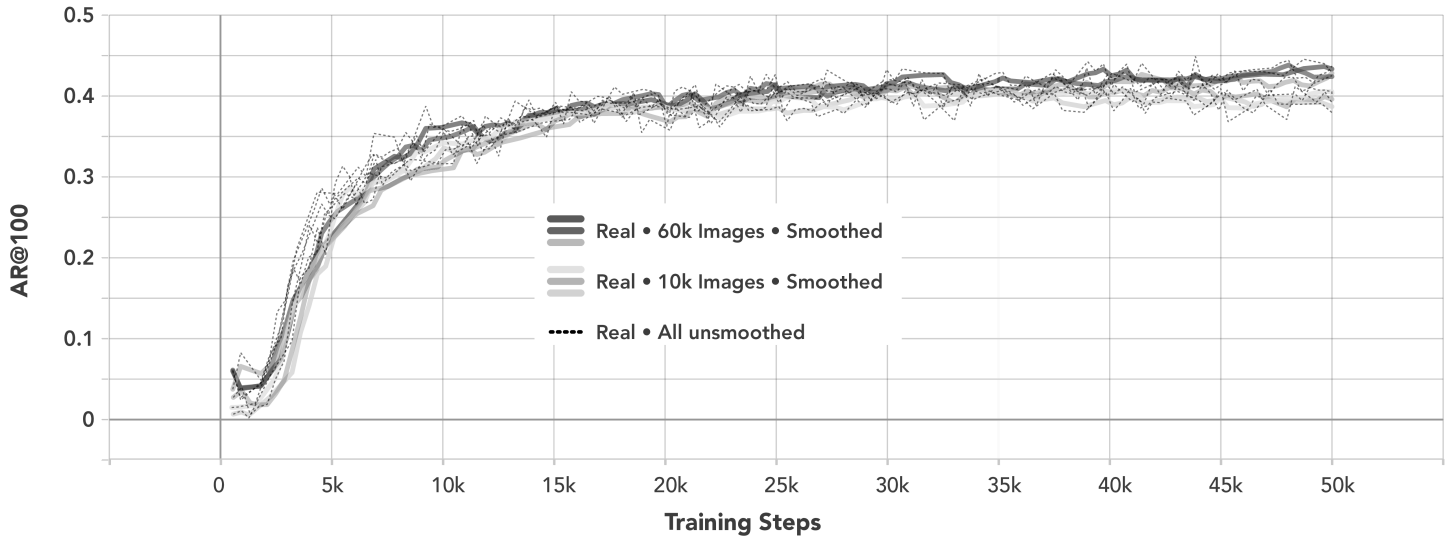

Figure B.15 – (Figure 5.1): Average Recall (`AR@100`) per real-data-trained model as training progresses through 50,000 steps. The models represented in this figure represent three trials each of a 60,000-image training set and a 10,000-image training set. The solid lines have had smoothing applied to them (default TensorBoard smoothing with a coefficient of 0.6), and the dotted lines represent the corresponding unsmoothed data. The unsmoothed data is close enough to the smoothed data (and it makes it easier to see long-term trends, like the subtle gradual split between the 10,000-image data points and the 60,000-image data points) that any subsequent visualization of recall over training step will be shown with this smoothed data. The metric that best defines the success of each individual model involves the `AR@100` value towards the end of training—the recall values before the model has been fully trained are irrelevant since they will be low by definition. Instead, in many subsequent visualizations, a model will be displayed as a box-and-whisker plot of the `AR@100` values of training steps greater than 35,000 for all trials trained with the same training dataset.