OBJECT TRACKING IN GAMES USING CONVOLUTIONAL NEURAL NETWORKS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Anirudh Venkatesh

June 2018

COMMITTEE MEMBERSHIP

TITLE:	Object Tracking in Games using Convolutional Neural Networks
AUTHOR:	Anirudh Venkatesh
DATE SUBMITTED:	June 2018
COMMITTEE CHAIR:	Alexander Dekhtyar, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	John Seng, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Franz Kurfess, Ph.D. Professor of Computer Science

ABSTRACT

Object Tracking in Games using Convolutional Neural Networks

Anirudh Venkatesh

Computer vision research has been growing rapidly over the last decade. Recent advancements in the field have been widely used in staple products across various industries. The automotive and medical industries have even pushed cars and equipment into production that use computer vision. However, there seems to be a lack of computer vision research in the game industry. With the advent of e-sports, competitive and casual gaming have reached new heights with regard to players, viewers, and content creators. This has allowed for avenues of research that did not exist prior.

In this thesis, we explore the practicality of object detection as applied in games. We designed a custom convolutional neural network detection model, SmashNet. The model was improved through classification weights generated from pre-training on the Caltech101 dataset with an accuracy of 62.29%. It was then trained on 2296 annotated frames from the competitive 2.5-dimensional fighting game Super Smash Brothers Melee to track coordinate locations of 4 specific characters in real-time. The detection model performs at a 68.25% accuracy across all 4 characters. In addition, as a demonstration of a practical application, we designed KirbyBot, a black-box adaptive bot which performs basic commands reactively based only on the tracked locations of two characters. It also collects very simple data on player habits. KirbyBot runs at a rate of 6-10 fps.

Object detection has several practical applications with regard to games, ranging from better AI design, to collecting data on player habits or game characters for competitive purposes or improvement updates.

ACKNOWLEDGMENTS

Thanks to:

- My family who've supported all my decisions and encouraged me every step of the way.
- Dr. Dekhtyar for giving me the opportunity to work on something that I was passionate about while constantly meeting with me every week to discuss concepts and improvements with regard to both my project and paper.
- Dr. Seng and Dr. Kurfess who've given me great ideas and kept me on track over the course of my research.
- My friends John and Michael who discussed ideas with me among others who kept me motivated.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Computers are an integral part of our daily lives. Whether it be productivity, accessing the web for information, or even for playing games, computers have seen prominent usage. The evolution of computers over the past few decades has contributed to great strides and development in fields such as Biology, Mathematics, and Finance. Since the advent of Machine Learning, computers have even been able to process more human-like qualities such as pattern, speech, image, and even visual object recognition [19].

Machine Learning succeeds with the availability of large quantities of data. Due to the current abundance of data, research has been prosperous in the field. This research contributes algorithmic and computing performance improvements that have allowed machines to analyze and learn from this data.

One of the most interesting areas of Machine Learning is Computer Vision. Computer Vision involves the derivation of a high-level understanding from low-level raw pixel information [40]. It is a complicated field of study due to the difficulty of understanding what these pixels represent. For example: given an image of a sunset and asked to give a description of the image, a human, under normal circumstances will be capable of quickly identifying the sunset. A machine, however, is not quite so capable. The image is processed as raw pixel data but the machine does not understand what this data represents. Advancements in Computer Vision help bridge this disconnect between the machine's and the human brain's abilities to interpret visual cues.

Computer Vision has recently been gaining traction. Image classification and object detection are prominent subsets of Computer Vision which are being adapted to fields such as medicine and engineering. In the automotive industry, research has been proposed utilizing both for applications like autonomous cars with regard to pedestrian detection [42] and road detection [25].

Another interesting area of utilization for Computer Vision is video games. A study published in 2015 suggests that with the recent advent of the e-sports industry and rise of competitive gaming, live streaming of games will increase by 140.6% between 2014 - 2019 [29]. The study also predicts an increase with regard to the viewing audience for games by approximately 305.5% within the same time period [29]. This increased growth of competitive gaming opens new avenues for Computer Vision research. Examples of potential benefits of this type of research include: collecting data on player habits to find faults or improvements, development of more complex game AI to help players learn faster, or implementing ideal balance updates. Computer Vision research in games is currently lacking. There are a few studies which seem to focus specifically on image classification combined with reinforcement learning or Q-learning and creating agents that can play games [18], [26]. Other subsets of Computer Vision such as object detection don't seem to have been recognized by the gaming community as a useful area of study. With this in mind, we decided to push the limited research done in this area by applying object detection to video games.

For our research, we chose to focus on object detection as a whole with several core classification concepts. We implemented a real-time detection model to classify and track video game characters from the popular fighting game *Super Smash Brothers Melee*. With this, we constructed a simplistic bot, capable of movement based on tracked locations of a secondary character on screen. This bot was trained using labeled frame data from the game and successfully performs basic actions in real time.

Object detection research has several practical applications:

- It can be used towards making an adaptive AI. Regardless of casual or competitive gaming, it can help the player practice their fundamentals and improve.
- Detection combined with classification can be used to track poses of the character in games allowing for new players to learn button combinations.
- With gaming competitions on the rise, tracking[1] objects in real-time allows for information extraction about characteristics of the player, the game, or even the character to find valuable improvements or flaws in gameplay.

Our major contributions directly coincide with the first and third practical applications detailed above. The proof-of-concept bot qualifies as an adaptive AI and can also be used to collect player and character information in real-time. These practical applications are handled as use cases. A secondary contribution is our custom CNN detection model which, with slight modifications, can be adapted to handle detection with various games. A final contribution is the annotated SSBM dataset used for detection.

The rest of this document introduces the core concepts of classification and detection in Chapter 2, then follows it up with related research in Chapter 3. Chapter 4 discusses the design and implementation details of our system pipeline. Chapter 5 details the results and validation metrics used to determine our detection model's success rate. Chapter 6 details the proof-of-concept bot. Chapter 7 focuses on possible future work and improvements to our current research and Chapter 8 provides a summary along with our closing thoughts.

[1]tracking may also be referred to as detecting for the purposes of this research.

Chapter 2

BACKGROUND

This chapter details the core concepts used in our research starting with image classification. Object detection is discussed in Section 2.3 as it builds on the concepts defined by classification. Since these concepts are part of the overarching field of Machine Learning, the understanding is that the computer needs to "learn" how to classify and detect. Let's start with the classification problem.

2.1 Classification

Classification aims to solve the problem of correctly categorizing an image or the object in an image as one of k possible classes [40]. This problem was popularized by the ImageNet Large Scale Visual Recognition Challenge (ISLVRC) since the year 2010 using the popular ImageNet dataset [36]. The classification problem qualifies under the concept of *supervised learning* where learning consists of using labeled image data. The result of this learning is the capability of accurately predicting the labels for unlabeled data. There are many solutions to this problem, but we opted to use a Neural Network based approach using Convolutional Neural Networks.

2.1.1 Generic Architecture of a Neural Network

Neurons are the fundamental units of computation in a neural network. Each neuron consists of weights and a bias associated with it which are used to extract information from from its incoming input. An example of a single neuron with respective inputs $\mathrm{x_i}$, weights $\mathrm{w_i}$, and output y is depicted in Figure 2.1.

The output is obtained as a result of the computation performed by the neuron

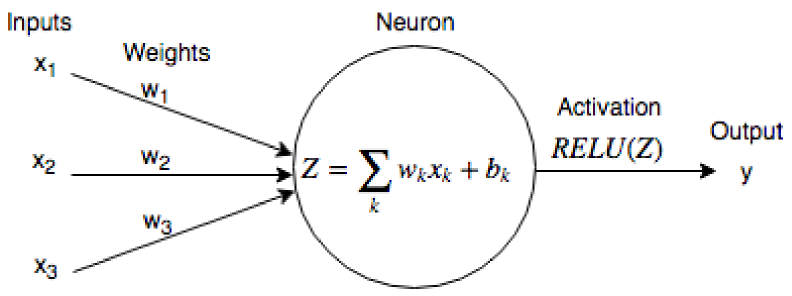


Figure 2.1: Visualization of a Neuron and its fundamental variables

which is subsequently subjected to a non-linear activation function ($RELU(Z)$ in Figure 2.1). The computation and activations are detailed in Sections 2.1.2 and 2.1.3 respectively. Neural networks consist of an input layer which is the data that is being sent in, a variable number of hidden layers containing stacked neurons to extract information, and an output layer for prediction. Each neuron in a neural network's hidden layer is typically connected to all neurons of the previous layer [1]. Figure 2.2 depicts a visualization of a simple 3-layer neural network with 2 hidden layers and 1 output layer.

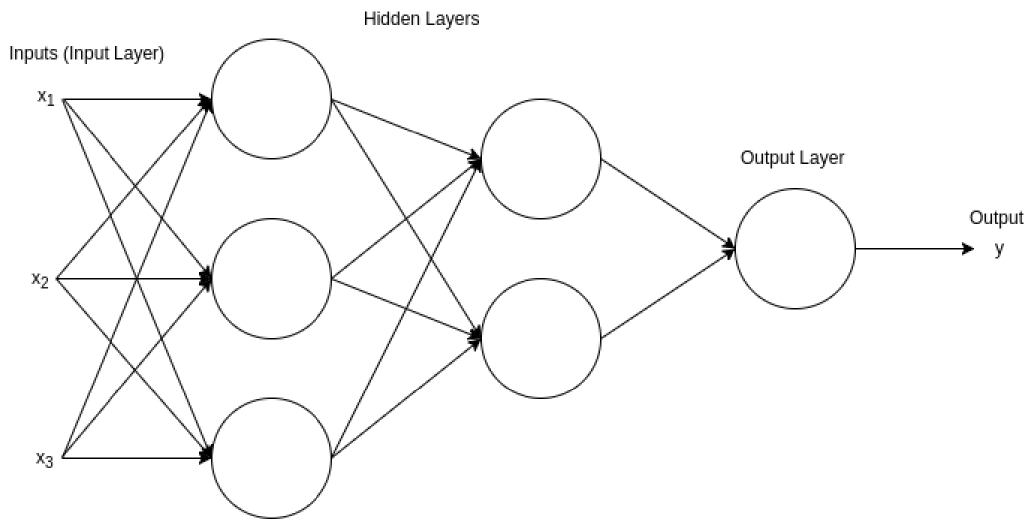


Figure 2.2: 3 Layer Neural Network with 2 hidden layers and 1 output layer

2.1.2 Neural Network Propagation

Neurons in a network do not start with the ideal information extracting weights. They need to undergo learning using an optimization algorithm in order to improve these weights to extract better information and make better predictions. This learning is done in two steps: forward and back propagation. During a forward propagation across the network, each neuron performs a dot product between its weights and the incoming input vector and adds the bias [1] as seen in Figure 2.1. The forward propagation equation for an individual computation for a neuron and an incoming connection can be understood as:

$$z(output) = w_{\mathrm{k}}x + b_{\mathrm{k}} \quad [27] \tag{2.1}$$

For a full vectorized version for all input connections to a neuron, the equation becomes:

$$Z(output) = WX + B \quad [27] \tag{2.2}$$

In the above equation, the weights W, input data X, and bias B for all input connections to a neuron are stored as matrices. The summation of a neuron's computations, Z, is passed through a non-linear activation function (Figure 2.1) for reasons detailed in Section 2.1.3. The output vector y (Figure 2.1) is the result of this activation. This computation happens across all layers of the network resulting in a final output vector y from the output layer (Figure 2.2).

The output or predicted vector y, is then compared with the true vector values and the loss[1] is calculated. The loss represents the difference between the true output and predicted output of the neural network after an instance of forward propagation [35].

[1]loss is also defined as "error" or "cost". Functions used to calculate loss are called "loss functions".

The loss across a single input can be understood by the example loss function:

$$L(y,t) = -((t)log(y) + (1-t)log(1-y)) \quad [27] \tag{2.3}$$

This loss function is commonly used with logistic regression where the possible outcomes are binary. In this loss function, t represents the true value of the input and y represents the predicted value. The total loss across all inputs is represented by the following equation:

$$TL(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(y_{\mathrm{i}}, t_{\mathrm{i}}) \quad [27] \tag{2.4}$$

Here, the total loss TL with respect to the weights w and bias b is calculated as the loss across all inputs m using their respective weights and biases.

Since the goal of the neural network is to decrease the loss, the best way to do so is to reach output values similar to the true values. In order to do so, the network goes through a process called back propagation through gradient descent. During back propagation, partial derivatives of the outputs, weights, and biases for each layer are calculated with the goal of adjusting the weights and bias values of the connections [27]. Let's take an example of three inputs x_1, x_2, and x_3 connecting to a single neuron with a loss of L(y, t). We'd like to update the weight and bias parameters w_1, w_2, w_3, and b_1, b_2, b_3 such that the loss decreases on the next iteration of forward propagation. We can assume that the output vector is y and the neuron's computation is Z (Figure 2.1). Back propagation is done in the following way:

First we compute the partial derivative of the loss with respect to the predicted output y:

$$\frac{\partial L(y,t)}{\partial y} = \frac{-t}{y} + \frac{1-t}{1-y} \quad [27] \tag{2.5}$$

We then compute the partial derivative of the loss with respect to the neuron's computation Z:

$$\frac{\partial L(y,t)}{\partial y} * \frac{\partial y}{\partial Z}, \quad \text{where: } \frac{\partial y}{\partial Z} = y(1-y) \quad [27] \tag{2.6}$$

$$\frac{\partial L(y,t)}{\partial Z} = y - t \quad [27] \tag{2.7}$$

We can now compute the partial derivative of the loss with respect to the weights and the bias values for input i:

$$\frac{\partial L(y,t)}{\partial w_{\mathrm{i}}} = x_{\mathrm{i}} * \frac{\partial L(y,t)}{\partial Z} \quad [27] \tag{2.8}$$

$$\frac{\partial L(y,t)}{\partial b_{\mathrm{i}}} = \frac{\partial L(y,t)}{\partial Z} \quad [27] \tag{2.9}$$

Using these partial derivatives, the updated weight and bias values for input i can be calculated as follows:

$$w_{\mathrm{i}} = w_{\mathrm{i}} - \alpha * \frac{\partial L(y,t)}{\partial w_{\mathrm{i}}} \quad [27] \tag{2.10}$$

$$b_{\mathrm{i}} = b_{\mathrm{i}} - \alpha * \frac{\partial L(y,t)}{\partial b_{\mathrm{i}}} \quad [27] \tag{2.11}$$

In the above equation, α is defined as the "learning rate" which is an adjustable hyperparameter when doing neural network training [27]. An intuitive understanding of the learning rate is that it allows the neural network the opportunity to learn at a certain pace by adjusting the weights using a modifier. It also allows the opportunity to increase or decrease this value during training given the need for it.

These equations fall under the concept of gradient descent - an optimization algorithm where the goal is to minimize the loss function. The reasoning behind this idea is that updating the weights based on the error can potentially help the predicted vector of the network become more similar, or "converge", with the true vector. This convergence will minimize the loss during each forward propagation after a gradient descent parameter update. When training a neural network several forward and back propagations are done utilizing variable input batch sizes. With the weights being updated after every instance of back propagation, the future forward propagations yield predictions closer to the true value of the inputs.

2.1.3 Activation Functions

Non-linear activation functions are used to introduce non-linearity in the model allowing the neural network to output information that cannot be produced by a linear combination of its inputs. This is done because most of the time, the data that is used is non-linear and the goal is to find a non-linear relationship to help with predictions [27]. There are several activation functions used for non-linearity, however this section details only the activation functions used in our research.

RELU

The Rectified Linear Unit or $RELU$ activation function computes the equation $f(x) = max(0, x)$ [1]. If the value x is less than 0, the resulting activation $f(x)$ is 0 as it is thresholded at a minimum of 0 [1]. Figure 2.3 depicts a visualization of the RELU activation function.

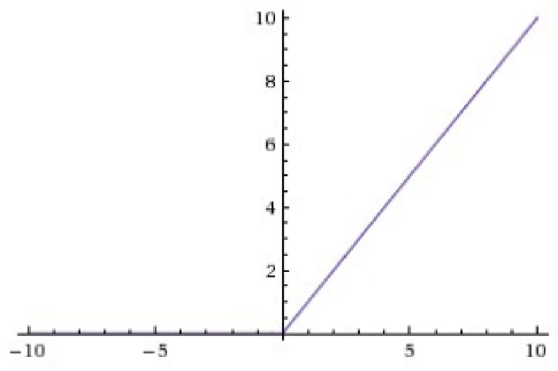


Figure 2.3: Relu Activation Function [1]

Softmax

The Softmax activation function is typically used in the final fully connected layer of a neural network. It is a commonly used activation function in classification research as it predicts a vector of probabilities for the various input classes [27]. The highest probability is the predicted class of the input [27]. Assuming a single input i from vector x for C classes to the softmax layer, an output y_{i} is defined as follows:

$$y_{\mathrm{i}} = \frac{e^{x_{\mathrm{i}}}}{\sum_{i=1}^{C} e^{x_{\mathrm{i}}}} \quad [27] \tag{2.12}$$

For the entire input vector x, the resulting output vector y is also a vector of the softmax activation applied to each input i of x.

2.1.4 Optimization Algorithms

Optimization algorithms are utilized in helping the neural network "learn" [27]. They do this by helping minimize the loss function (Equation 2.3) output which measures the loss between the predicted vector and true vector. Gradient descent during back propagation is an example of optimization as it helps alter the parameters of the network. This section focuses on the two optimization algorithms we used in our research: SGD Nesterov and Adam.

Stochastic Gradient Descent Nesterov

Stochastic Gradient Descent or *SGD* is a type of optimization which performs a gradient descent update using a particular batch size. It updates the parameters after computing the average across the gradients for the batch of inputs. Gradient descent in general has two types of momentum based learning functions which are commonly used - Momentum and Nesterov momentum. Nesterov builds on concepts introduced by Momentum. To understand it, let's start by defining a parameter update similar as follows:

$$P = P - \alpha dP \quad [27] \tag{2.13}$$

In this equation: dP represents the derivative of the parameter P with respect to the loss. This parameter update is identical to the parameter updates introduced in back propagation with Equations 2.10 and 2.11. Momentum introduces a velocity term and utilizes it to calculate smoother gradients [27].

$$v_{\mathrm{dP}} = \beta v_{\mathrm{dP}} + (1-\beta)dP \quad [27] \tag{2.14}$$

β is the momentum coefficient usually set to 0.9. The calculated velocity term for parameter P is used in calculating the updated parameter weights.

$$P = P - \alpha v_{\mathrm{dP}} \quad [27] \tag{2.15}$$

Let's take an example of a moving ball to understand the difference between the two optimization methods. This momentum function first calculates the gradient and then the ball moves towards the total gradient [15]. The Nesterov approach is different in that the ball moves in the direction given by the previous gradient, and then calculates the gradient update and corrects its position if needed [15]. The equation defined by this approach is as follows:

$$v_{\mathrm{dP}} = \beta v_{\mathrm{dP}} + d(P - \alpha\beta v_{\mathrm{dp}}) \tag{2.16}$$

The derivative with respect to the previous gradient is calculated and the parameter value is updated in the same way as Equation 2.15. The Nesterov method is a more feasible approach as Hinton himself puts it: "It turns out it's better to gamble and then make a correction, instead of making a correction, and then gamble" [15]. It is worth mentioning that there exist other implementations of SGD Momentum and Nesterov which factor α as part of the v_{dP} calculation and update the parameter using just v_{dP} as demonstrated in [34].

Adam

Adam is another optimization algorithm that has performed as well as momentum for a large range of tasks in machine learning [27]. It is a combination of gradient descent with momentum and Root Mean Squared propagation or *RMSProp* [27]. RMSProp is an optimization algorithm which takes a moving average of squares across the previous gradients [16]. It divides the current gradient by the square root of this moving average of squares as a normalization method to allow for better parameter updates [16]. The equation for RMSProp and momentum for parameter P are defined as follows:

$$RMSProp \quad s_{\mathrm{dP}} = \beta_2 s_{\mathrm{dP}} + (1-\beta_2)dP^2 \quad [27] \tag{2.17}$$

$$Momentum \quad v_{\mathrm{dP}} = \beta_1 v_{\mathrm{dP}} + (1-\beta_1)dP \quad [27] \tag{2.18}$$

In the RMSProp equation, s is introduced as a term utilized to normalize the current gradient when performing a parameter update. These equations are usually corrected for bias (not shown here) [27]. Using Equations 2.17 and 2.18, Adam does a parameter update as follows:

$$P = P - \alpha \frac{v_{\mathrm{dP}}}{\sqrt{s_{\mathrm{dP}}} + \epsilon} \quad [27] \tag{2.19}$$

ϵ, β_1, β_2, and α are tunable hyperparameters of the neural network.

2.2 Convolutional Neural Networks

Convolutional Neural Networks or *CNNs* are used specifically for learning from image data as standard neural networks scale poorly to images. An input image of width 32 pixels, height 32 pixels, and 3 color channels[2] (red, blue green) would require 3072 weights for a single fully connected neuron at the first hidden layer [1]. However, an image of size 800x800x3 would require 1,920,000 weights for a fully connected neuron at the first hidden layer which is an exponential increase. Therefore, CNNs were created as an alternative. In CNNs, the layers arrange the neurons in a 3 dimensional structure to specifically account for the width, height, and depth dimensions of an image [1].

[2]Channels are also referred to as "feature maps" or "depth"

2.2.1 Architecture of a CNN

The fundamental unit of computation for all CNNs are Filters[3]. Figure 2.4 depicts a simple CNN with a single channel input layer connecting to a single filter convolutional layer.

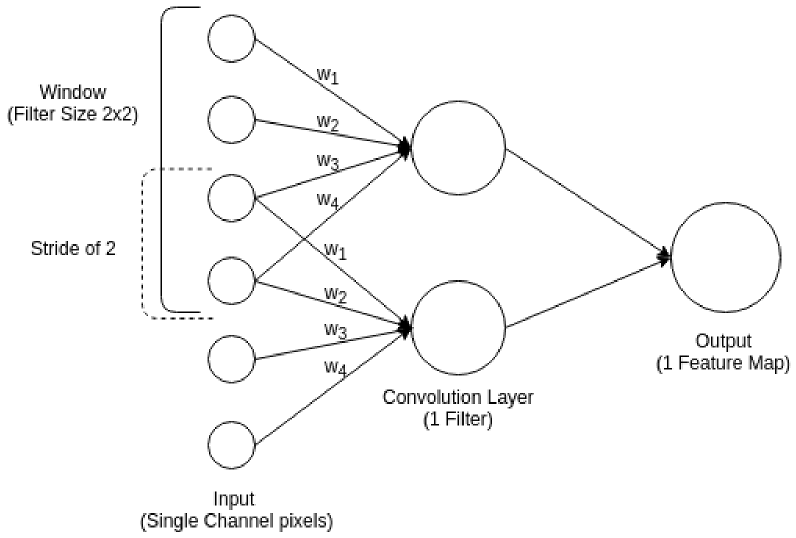


Figure 2.4: Visual of a simple CNN depicting a single-channel Input Layer and single filter Convolutional Layer.

The inputs are the pixels in an image which are fed into the convolutional layers as a means of extracting information during learning. They are typically, subsequently subjected to a non-linear activation function and a pooling operation (not shown in Figure 2.4). The following sections further detail the important layers of a CNN and their respective roles in learning.

Input Layer

The input layer is the first layer of any neural network which feeds in the data that the network learns from. In the case of a CNN, this input is restricted to raw pixels across 3 dimensions: width x height x depth [1]. While Figure 2.4 assumes input as a

[3]Filters are also referred to as Neurons.

single channel image, the depth is typically comprised of 3 color channels - red, blue, green [1]. These channels are considered as feature maps as they detail all the features of the image such as horizontal and vertical edges, color, blobs, among others. For the purposes of this research it is ideal to understand that each feature map or channel of an image can be represented as a matrix of pixel values.

Convolutional Layer

Convolutional layers are considered "the core building block of a CNN that do most of the computational heavy lifting" [1]. A convolutional layer consists of n filters that can be learned via training [40]. These learned filters are used to extract simple features such as angled edges, blobs, colors in early convolutional layers of a CNN. However, later layers extract more complex features representative of the input data. Figure 2.5 depicts a visual of learned filters obtained from training a CNN.

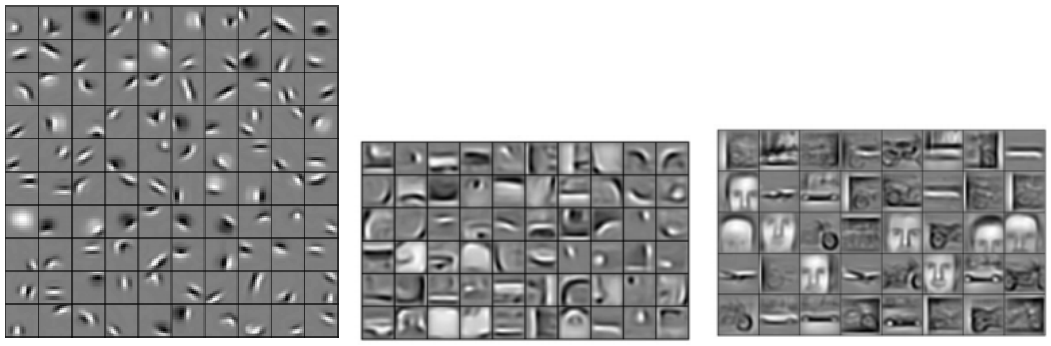


Figure 2.5: Visual of Convolution layer filters used for feature extraction - early layer filters (left), later layers (middle and right) [21]

These filters extract information by performing the convolution operation with the input using a specific window size and stride value (Figure 2.4). An example convolution operation is depicted in Figure 2.6.

The computations done in the convolutional layer involve superimposing the filter on the input as a - "receptive field"[4] of interest. The filter is then moved across the

[4]Receptive field is also referred to as "window size".

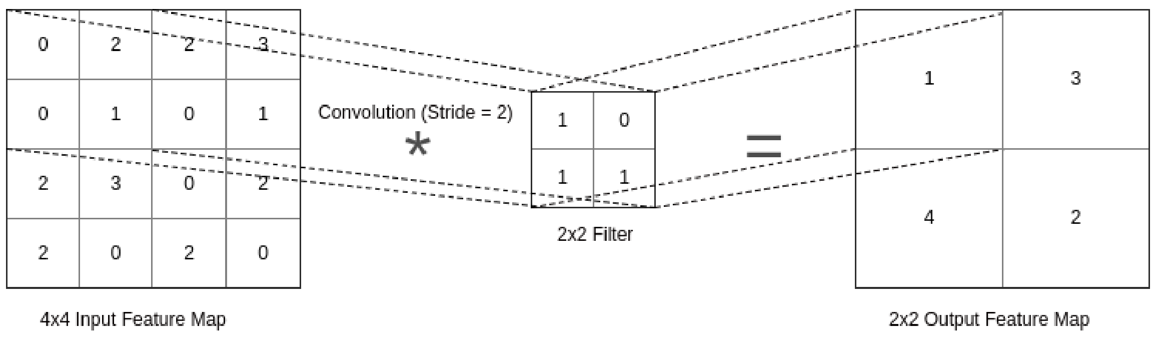


Figure 2.6: Convolution Operation example using a window size of 2x2 and stride of 2.

width and height of the input volume, by column first, and by row second until the end is reached. With every movement, the dot product (convolution) between the two matrices is calculated and a 2-dimensional feature map of the extracted spatial features is produced [1]. The size of this 2-dimensional feature map can be calculated as follows:

$$Size = \frac{(W - F + 2P)}{(S + 1)} \quad [1] \tag{2.20}$$

In this equation, W is the size of the input feature map, F is the size of the filter, P is any padding, and S is the stride. The padding refers to any outer layers of 0's added to the input feature map, while the stride refers to the amount of pixels to shift the filter's field during convolution. For example, if the stride is 2, the filter will move over by two columns when moving right and two rows when moving down. Padding is normally added to produce a desired size for the output map.

As a whole, the convolutional layer takes in as input the feature maps from the previous layer and outputs n (number of filters) stacked feature maps [40]. This is a result of each filter performing the convolution operation across the entire input volume. This idea is depicted in Figure 2.7 for n=2 filters.

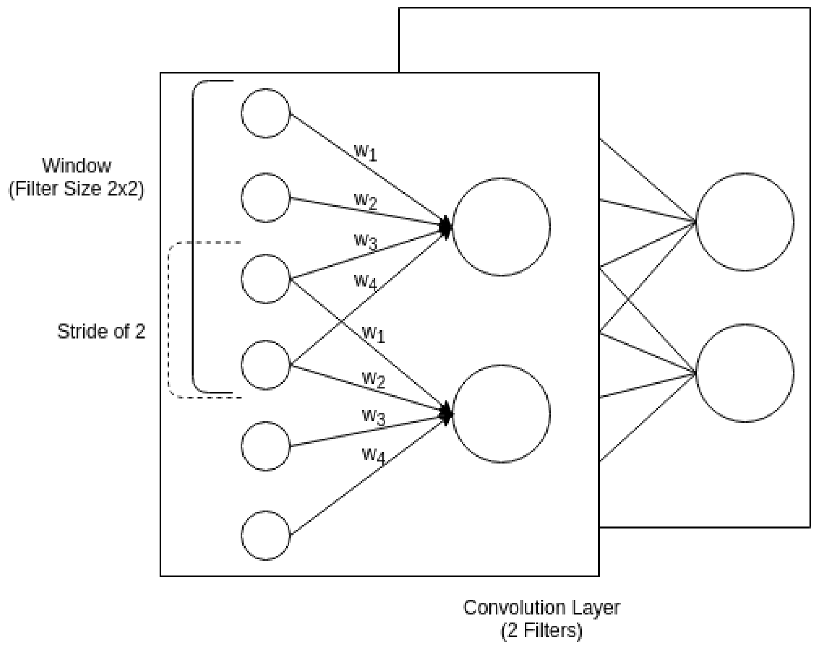


Figure 2.7: Visual of a simple CNN depicting a single-channel Input Layer and two filter Convolutional Layer. The outer rectangles can be considered as the resulting feature maps for each filter's convolution across the input volume.

Activation Layer

An activation layer is typically used after performing the computations from a convolutional layer. Most CNNs commonly stack sets of convolutional-activation layers and then follow it up with pooling layers [1]. RELU is the activation function used in most CNNs. RELU activation of max(0,x) is applied to every element across the input space resulting in the same output volume.

Pooling Layer

Pooling layers are used in CNNs for downsampling an image over the width and height dimensions, but the number of feature maps remains unchanged. The reduction of these dimensions allows for a lowered amount of both computations and overfitting due to a smaller number of parameters [1]. Pooling also allows for translational invariance thereby learning features regardless of the location or small changes in the image [40]. The operation is performed independently on every feature map across the total input volume. The pooling layers are similar to convolutional layers in that they perform an operation over a receptive field which shifts across the input pixels using a stride value.

The pooling layer takes in an input of W_1 (width), H_1 (height), D_1 (depth or feature maps), and hyperparameters: F (field size) and S (stride) [1]. The resulting output's shape of W_2, H_2, and D_2 produced by pooling is defined by the following equations:

$$W_2 = \frac{(W_1 - F)}{(S + 1)} \quad [1] \tag{2.21}$$

$$H_2 = \frac{(H_1 - F)}{(S + 1)} \quad [1] \tag{2.22}$$

$$D_2 = D_1 \quad [1] \tag{2.23}$$

The most common hyperparameter values used in CNNs are F = 2 (2x2), and S = 2. Filter sizes greater than 3 often result in worse performance due to sampling across a larger cross-section [1]. Two types of pooling are commonly used in CNNs:

- ***Max Pooling***

  The max pooling operation selects the maximum pixel value across the field of interest.

- ***Average Pooling***

  The average pooling operation takes an average of the pixel values across the field of interest.

Figure 2.8 depicts a visual of Max Pooling and Average Pooling across a feature map with a field size of 2x2 and a stride of 2.

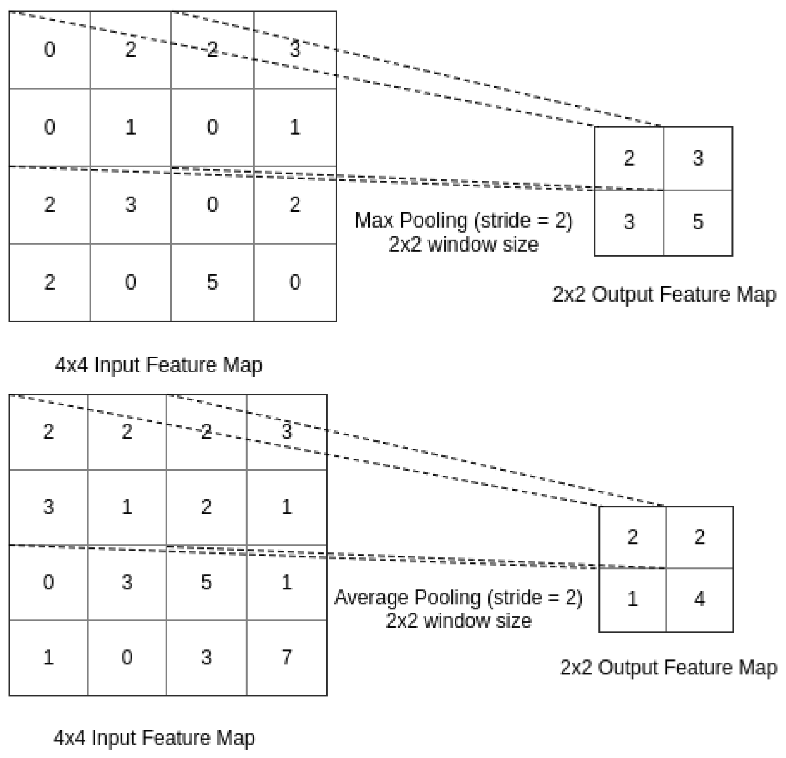


Figure 2.8: Max Pooling (Top) and Average Pooling (Bottom) across a 4x4 feature map using a field size of 2x2 and stride of 2.

Fully Connected Layer

Fully connected or FC[5] layers typically appear at the end of the CNN architecture. The neurons in these layers are similar to standard neural networks in two ways:

- They are fully connected to every input [1]. In the case of CNNs, the input is usually several feature maps as a result of a prior activation layer.
- The computation they perform is the same as a standard neural network's hidden layer computation. The activation is computed by multiplying two matrices and the addition of a bias [1]. It follows the concept introduced by formula 2.2.

FC layers in CNNs are used to change the input dimensions to 1x1xY where Y represents the number of neurons in the layer. The final layer of a CNN is an FC layer

[5]FC layers are also known as Dense Layers.

or “Softmax layer” which outputs a vector of predictions for classification. In this final layer, the activation function used is the softmax activation function (explained in Section 2.1.4) which helps compute the output probabilities [27].

2.2.2 CNN Loss Functions

Loss functions measure the difference between the true value and the predicted value. For any input, where the goal is to choose a single class from a list of classes, the softmax layer is used as the last layer of the CNN. This softmax layer outputs a vector of predictions for each class. The loss is computed by comparing this vector to the vector of true class values. The goal of choosing a class from a list of classes C, where C is greater than 2, is known as Multi-Class classification [27].

The probabilities for each class are represented in the output vector and the largest probability is selected as the decided class. The loss function is commonly known as *Categorical Cross Entropy* and to a lesser extent as just *Cross Entropy.* It is represented by the following equation:

$$L = -\sum_{j=1}^{C} y_{\mathrm{j}} \log(\hat{y}_{\mathrm{j}}) \quad [27] \tag{2.24}$$

In this equation, C represents the number of classes, y represents the true value at class j (usually a 0, when the image is not of that class, or a 1, indicating that the image is of that class), and $\hat{y}$ represents the predicted probability for class j. The overall loss ends up being the summation of the individual loss for all the classes.

2.2.3 Summary of CNN Architectures - LeNet-5 Example

CNN architectures vary significantly across one another and are complex in nature. This is due to the increased success of deeper networks in recent years, henceforth most CNN architectures use many layers. For the purposes of understanding CNN architectures, let's take a popular example - LeNet-5. Figure 2.9 depicts the LeNet-5 Architecture. It is a small and simple CNN proposed in LeCun et al.'s paper which was trained to classify digits [20].

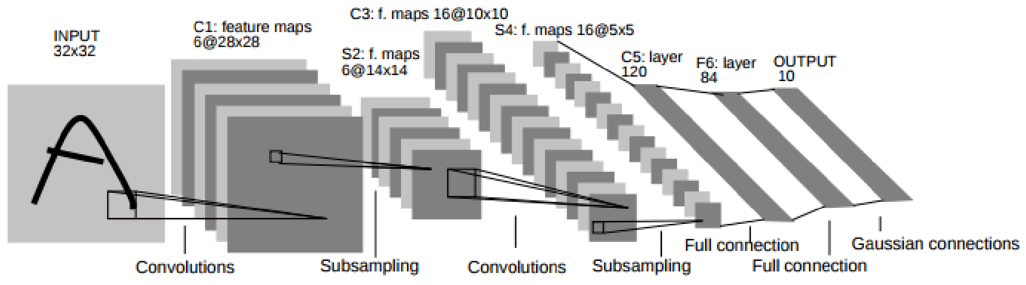


Figure 2.9: LeNet-5 Architecture for digit classification [20]

Combining the concepts detailed in earlier CNN sections, LeNet-5's structure and functionality can be understood as follows:

- **Layer 0:** Input layer which feeds images of the digit into LeNet-5.
- **Layer 1:** Perform convolution with 6 filters on input 32x32xD where D refers to the input's depth dimension. This is done to extract basic features such as edges.
- **Layer 2:** Perform pooling on the outputs to reduce dimensionality from 28x28 to 14x14.
- **Layer 3:** Perform convolution with 16 filters to extract more complex features.
- **Layer 4:** Perform pooling to reduce dimensionality.
- **Layers 5-7:** FC layers change dimensions, derive information across all inputs, and predict digit through Softmax activation.

The general idea of a CNN is to perform convolutions followed by non-linear RELU activations to extract features based on the dimension size which is controlled by pooling. This is then subject to FC layers which predict an output class using softmax activation based on all extracted features. The loss is computed and the weights[6] are updated via back propagation.

2.3 Object Detection

In the past few sections, CNN architectures and their use in classification have been detailed. The following sections explore the fundamental concepts behind object detection. Object detection has gained popularity over the recent decade due to the *PASCAL Visual Object Classes Challenge* competition. Several viable uses of object detection have been proposed during this time. One such example is by utilization of the HOG classifier combined with an SVM to detect moving cars in video footage [8]. The issue behind these classifiers however and why CNNs have gained traction is performance. CNNs are capable of achieving higher accuracy predictions when using large datasets to learn from. Our research utilizes CNNs for object detection. The object detection algorithm will be explained further in Section 2.3.2. But first, in order to understand object detection, let's first look at the concept that leads up to it - object localization.

2.3.1 Localization

Object localization is a concept also popularized by the ISLVRC similar to that of classification. Localization builds on the classification task and generates a bounding box based on the location of the classified object in the image [27].

Localization can be performed through two possible ways - with classification or classification and regression [22]. With the pure classification approach, a sliding window would need to be used to traverse across the image and classify each region

[6]"Weights" are the values of each filter. They may also be referred to as filters or filter values.

[27]. These regions would need to then be filtered by some criteria to determine the best possible bounding box. The method is simply too slow as it requires variable window sizes to handle variable sized objects [27]. With this, the solution left is classification and regression. This concept utilizes a CNN with a *classification head* and a *regression head* to predict the object's class and the bounding box of the object respectively [22]. When training this CNN, two types of input data are sent: images and ground-truth bounding box coordinates [22]. The output result ends up being a classification vector and coordinates for the bounding box in the form of (x, y, w, h) [22]. The coordinate (x,y) represents the center of the bounding box, w represents the width, and h represents the height of the box [27]. Figure 2.10 shows an example of classification-localization of an image with the class "Dog".

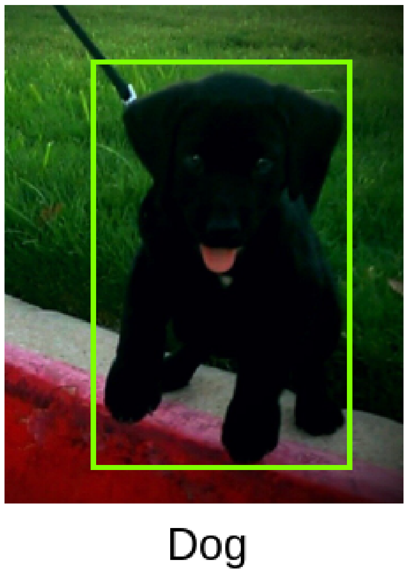


Figure 2.10: Classification-Localization of a dog in the image

This problem deemed the "classification-localization" problem handles the classification of a particular object and identifies its location in the image [27]. The institutions participating in the ISLVRC typically try to improve on this problem accuracy wise with their submissions. Let's move on to detection as many of these core concepts are utilized in that problem space.

2.3.2 Detection

The fundamental issue with localization is that it can only classify and predict boxes for one object at a time. In contrast, the goal with detection is to classify all object instances of every class in an image [27]. Figure 2.11 depicts a visual of multiple object detection of “Dog” and “Car” classes.

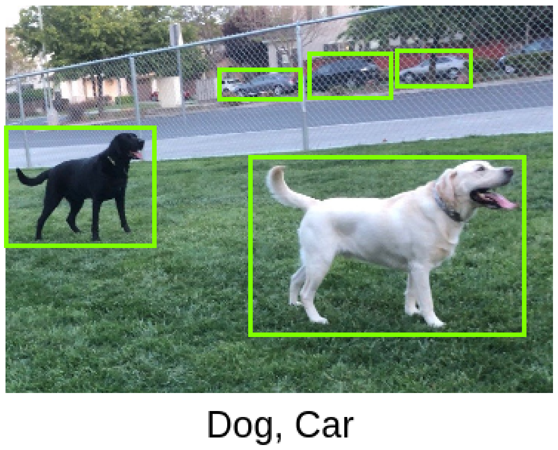


Figure 2.11: Detecting multiple classes in an image

There have been several algorithms proposed over the last decade to tackle this goal. Yet again, the methods are to utilize a classification or classification-regression approach [22]. As explained above, the pure classification approach using sliding windows is too slow, so the most common approach is to use a classification-regression approach. Two important algorithms which utilize this approach will be explained briefly below:

- **R-CNN**

  R-CNN utilizes a region proposal algorithm called selective search to propose numerous boxes across the image [14]. These boxes are fed into a CNN with an SVM and classified if an object is present [14]. The box sizes are also reduced to fit the object better via a linear regression model for each class [14]. This method produces a good prediction accuracy but is slow as a result of needing to pass the region proposals into the CNN individually and storing the predicted features on disk [13].

- **Faster R-CNN**

  R-CNN was improved into Fast R-CNN through the inclusion of an ROI-Pooling layer which allowed for computations to be shared across the different region proposals [13]. This improved performance but selective search was still slow. Faster R-CNN is an improved version of Fast R-CNN in the area of region proposal optimization [32]. It utilizes a fully convolutional network deemed the *region proposal network* (RPN) and anchor boxes - variable sized boxes [32]. After passing through a standard CNN, the resulting feature map is passed through the RPN [32]. Object containing regions are proposed, classified, and box sizes are reduced for better predictions [32].

These algorithms are some of the developments seen in object detection research. Due to its performance on the PASCAL VOC 2007 challenge [32], Faster R-CNN has been widely regarded as a state of the art object detection algorithm. However, the algorithm that was used in our research - *YOLO*, is explained in the next section.

2.3.3 YOLO Object Detection

The YOLO or *You Only Look Once* algorithm is another recent development in object detection research. The goal with this algorithm is that the bounding box predictions need to be made in just one pass across the entire image. The YOLO algorithm outperforms Faster R-CNN in speed and accuracy due to recent improvements [31]. YOLO considers detection to be a regression problem [30]. The fundamental concept behind the YOLO algorithm is that the input images are resized to a fixed size of 416x416 and then fed into a CNN [31]. More information about the CNN YOLO uses and its recent updates can be found in Redmon et al.'s papers [30], [31] respectively. This CNN outputs a volume with dimensions SxSxB*(5+C) [27]. Any CNN architecture can be utilized as long as it outputs a volume of that context, so we use a custom CNN in our research. The image is divided into a grid of width S by height S. Each object in the image is assigned to the grid cell which contains its midpoint [27]. B is the number of anchor boxes (predetermined box sizes) and C is the total number of classes in our dataset [30]. Intuitively, we perform classification-localization on every grid cell in the image [27] for each anchor box to allow for multiple object detections. This output volume contains the detected object predictions and their respective bounding box coordinates for each grid cell. The next few sections detail the important concepts of YOLO detection.

IOU

IOU or *intersection over union* is a way to evaluate the bounding box prediction with respect to the ground-truth box. It is used in both object localization and detection. A visualization of the IOU equation is depicted in Figure 2.12.

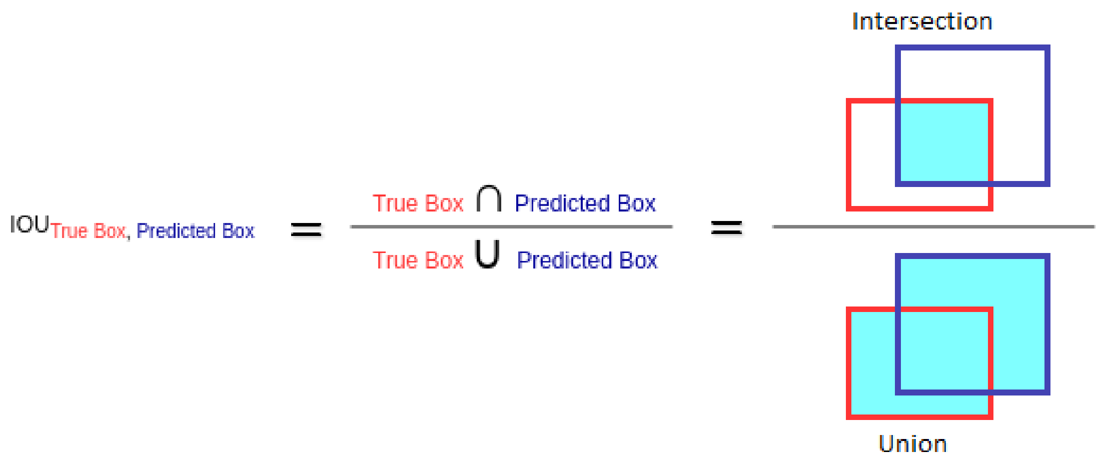


Figure 2.12: IOU algorithm representation (image was inspired by [33])

Using the bounding box coordinates defined by the anchor box and the ground-truth box, the IOU is calculated. An IOU greater than or equal to 0.5 is typically considered a good or "True" detection [27]. Figure 2.13 illustrates some good and bad IOU detections.

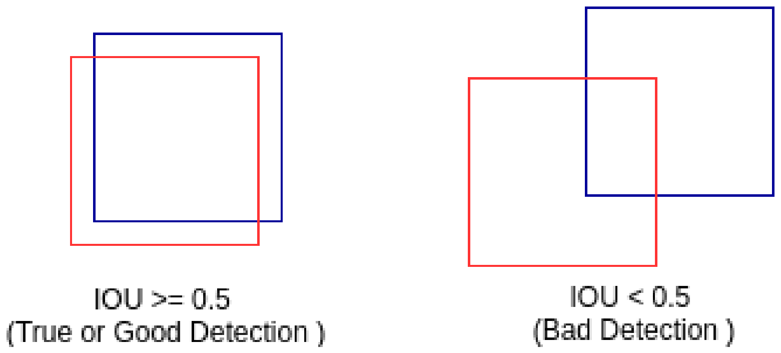


Figure 2.13: Various IOU detections (image was inspired by [33])

Anchor Boxes

Anchor boxes are predefined box shapes which are used to help grid cells detect multiple different objects. Each defined anchor box can detect one object per cell. Figure 2.14 depicts an image of two predefined anchor boxes. There are 5+C variables

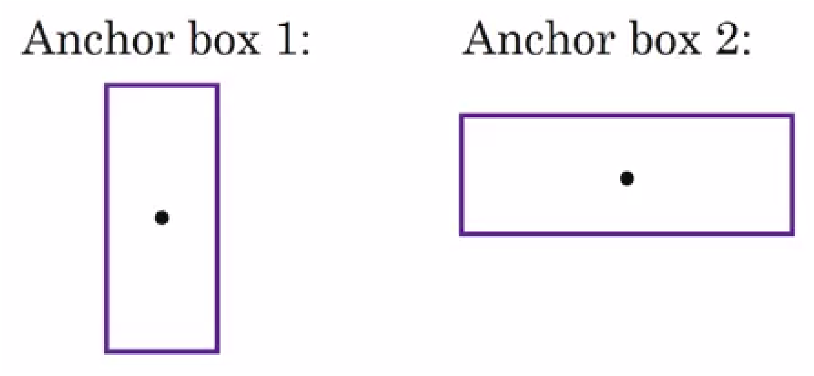


Figure 2.14: Image of two predefined anchor boxes [27]

typically associated with each anchor box - as indicated in the earlier equation in Section 2.3.3.

These variables contain information about the grid cell and are defined as follows:

- $\mathbf{p_c}$: which represents the confidence of there being an object in the image [27]. A value close to 1 indicates the presence of an object, while a value close to 0 indicates the opposite [27].
- $\mathbf{(b_x,\ b_y)}$: coordinates (x,y) which represent the center of the bounding box[7] [30].
- $\mathbf{b_w}$: which is the width of the bounding box [30].
- $\mathbf{b_h}$: which is the height of the bounding box [30].

These are the 5 variables in 5+C. The C refers to the number of classes in the dataset. If an object is detected to be of a certain class, that value is set to 1 in the anchor box output for that grid cell, otherwise it is a 0. An example of the way anchor boxes are used is depicted in Figure 2.15 below.

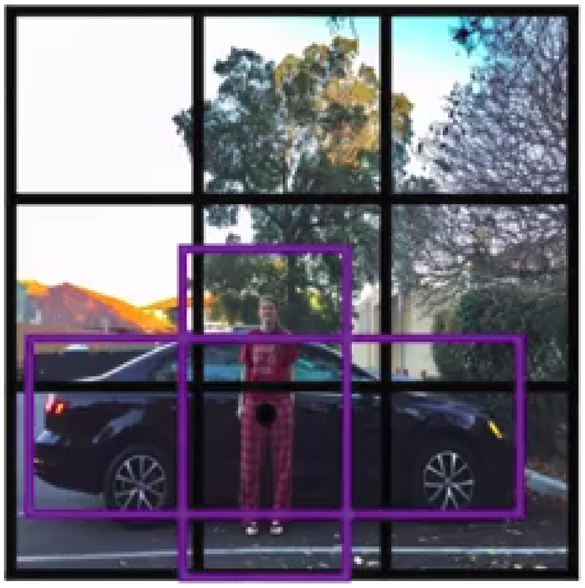


Figure 2.15: Image with anchor boxes assigned to respective objects [27]

[7]"bounding boxes" are actual values predicted by the CNN that are representative of the shapes defined by the anchor boxes. They typically encompass the detected object in its entirety.

Here, the pedestrian is detected by box 1 and the car is detected by box 2. The detected object is assigned to the box with the higher IOU when compared to the ground-truth box [27]. Assuming that there are 3 classes being detected for the image in Figure 2.15, the output structure for both anchor boxes and a single detection cell is shown in Figure 2.16.

$$\begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \\ p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

Figure 2.16: Two Anchor Box output volume per grid cell [27]

In addition, we can determine the output size of the YOLO algorithm for the image in Figure 2.15. Let's assume that the image has been divided into a 3x3 grid after detection and there are 2 anchor boxes. Based on Section 2.3.3, the output dimensions from YOLO are 3x3x2*(5+3) = 3x3x2x8. This is apparent in Figure 2.16 as there are 16 outputs per grid cell which is representative of the 3x3x2x8 (3x3x16) output.

Non-Max Suppression

The YOLO algorithm often detects the same object multiple times. *Non-Max Suppression* or NMS is a way to yield only a single detection per object. Figure 2.17 shows an image with detected objects that has not undergone NMS. If this detected

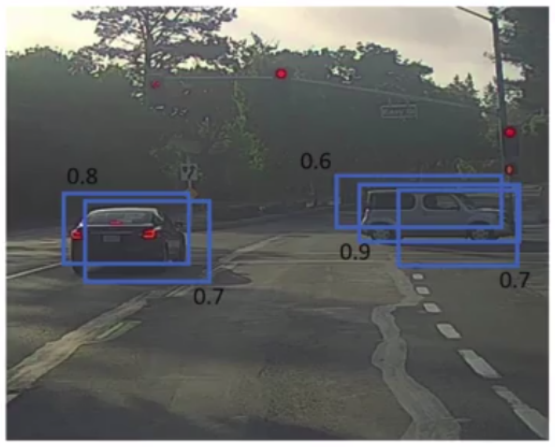


Figure 2.17: Detection without Non-Max Suppression [27]

image were subjected to NMS, the boxes with the highest confidence p_c: 0.8 for the car on the left and 0.9 for the car on the right will be selected. The rest of the boxes which have a high IOU with these selected boxes will be omitted or "suppressed" [27]. The intuitive understanding here is that the best box has been selected while the others which are very similar have been removed.

The NMS algorithm is applied after an image goes through the YOLO detection CNN process. Let's assume as an example that we are only trying to detect one object and have just one anchor box. This means that when the image went through the CNN, it was divided up into SxS grid cells and each cell detects up to one bounding box.

The algorithm follows these steps to select the best boxes:

- A threshold value for the confidence p_c is set [27]. All box predictions with p_c values less than or equal to this threshold are automatically rejected.
- As long as there are boxes remaining, the box with the best p_c is selected. Any box which has a high IOU (greater than or equal to 0.5) with this selected box is discarded [27].

This example was detecting objects of just one class. Most common research including ours detects objects of multiple classes. NMS should be applied for each of these individual classes separately [27].

Loss Function

The importance of the loss function was detailed in Section 2.2.2 on CNNs with multi-class classification loss. However, YOLO utilizes its own custom loss function. Figure 2.18 shows the YOLO loss function.

$$\begin{aligned}
&\lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
&+ \lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2
\end{aligned}$$

Figure 2.18: Loss Function for YOLO Object Detection [30]

According to Redmon et al.: "$\mathbb{1}_i^{obj}$ denotes if object appears in cell i and $\mathbb{1}_{ij}^{obj}$ denotes that the j^{th} bounding box predictor in cell i is "responsible" for that prediction" [30]. So the understanding is that $\mathbb{1}_{ij}^{obj}$ is 1 if the j^{th} anchor box detects the object in cell i (by having the highest confidence), and 0 otherwise. Similarly, $\mathbb{1}_{ij}^{noobj}$ is 1 if the j^{th} anchor box does not detect an object in cell i, and 0 otherwise. With this, we can understand the general idea of how the YOLO loss function operates by looking at the variables:

- $\mathbf{[x_i,\ y_i,\ w_i,\ h_i]}$ represent the coordinates of the bounding box at grid cell i. The coordinate (x,y) represents the center of the bounding box whereas w and h represent the width and height of the bounding box. Lines 1 and 2 of the loss function penalize inaccuracies of these coordinates across the entire SxS grid for any object predicted in cell i using anchor box j.

- $\mathbf{C_i}$ represents the confidence of an object being present in a particular grid cell. Lines 3 and 4 penalize lower confidence values of detecting an existing object and high confidence values for incorrectly detecting an object when one doesn't exist. The actual confidence is determined by calculating the IOU of the predicted box with respect to the ground-truth box. The mean squared error is then taken between the actual and predicted values.

- $\mathbf{p_i(c)}$ represents the predicted class of the object. The final line of the loss function penalizes an incorrect prediction of the object's class.

λ_{coord} represents an added penalty factor which is used to increase or decrease the severity of the penalty. We explain the values used for this in our implementation details.

2.3.4 Evaluation Metrics

The standard evaluation metric for object detection is *Mean Average Precision* or MAP. MAP is the mean across the average precision of all classes. *Average precision* or AP gives an idea of how well the detection system is performing by analyzing the precision with respect to the recall of the detections of each class. Precision and recall for object detection purposes are defined by the following equations:

$$Precision = \frac{TruePositives(TP)}{TP + FalsePositives(FP)} \tag{2.25}$$

$$Recall = \frac{TP}{TP + FalseNegatives(FN)} \tag{2.26}$$

True positives are correct object detections. False positives are detections of objects which don't exist or incorrect detections of objects. False negatives are missing detections of existing objects. These values are determined for each box detection based on its overlap with the ground-truth box. An IOU minimum threshold of 0.5 is used as a true positive whereas a smaller value indicates a false positive. The precision gives us the percentage of accurate predictions with respect to the total predictions. The recall gives us the percentage of accurate predictions with respect to the total number of objects.

Using these equations, we can calculate Average Precision as follows:

$$AP = \frac{1}{11} \sum_{r \in \{0,0.1,\ldots,1\}} p_{\text{interp}}(r) \quad [10] \tag{2.27}$$

This method of calculating AP was introduced for the PASCAL VOC object detection challenge [10]. As the equation indicates: the interpolated precision p is computed over equal recall value increments of 0.1 across the range of [0,1] [10]. The interpolation indicates taking the max precision value across recall values greater than or equal to r. This is done for each individual class by calculating the number of accurate detections of that class in the top n detections thereby establishing a base precision and recall value. The value n is increased by 1 until the recall reaches 1.0 and the interpolated precision is calculated for each increment. The mean taken across the 11 precisions at the specific 0.1 recall increments is defined to be the average precision [10].

2.4 Super Smash Brothers Melee

Super Smash Brothers Melee (SSBM) is a popular 2.5D tournament fighting game where two to four characters attempt to knock one another off of the fighting stage. The game was released by Nintendo in 2001 for the Nintendo Gamecube [28] and still continues to be played in a competitive setting. We chose SSBM as our research focus for three reasons:

- Since our goal is to track video game characters in real time, we wanted to use a game where characters could move across the screen very quickly to see if the detection model could keep up.
- SSBM is a 2.5D game where characters operate on a 2D plane. This makes it easier to calculate coordinates when determining bot movements. Figure 2.19 shows a frame from an SSBM match.

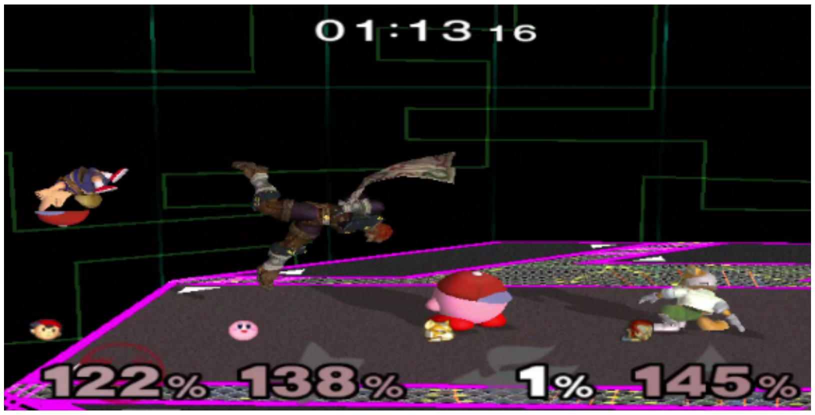


Figure 2.19: Frame from SSBM match

- 3-Dimensional characters are more complicated feature wise which will require a longer learning process and more compute power.

The variability of each character and the multitude of possible animation frames makes SSBM a prime candidate for object detection research. Factoring all possible frames for a single character across the full roster, stage and background variability, item variance, and move collision, there exist millions of possible object combinations.

2.5 Frameworks and Programs

- **Keras**: Keras is a Python deep-learning framework used to model CNNs [2]. We used the Tensorflow-GPU backend allowing for Keras to utilize the GPU (GTX 980) for training both the classifier and detector.
- **LabelImg**: An open source program which allows for the labeling of objects in images. It produces an XML file with the bounding box coordinates in PASCAL VOC format [3].
- **Basic Yolo Keras**: An open source implementation of YOLO for the Keras framework which we modified to suit our needs [6].
- **Mean Average Precision**: An open source implementation of Mean Average Precision which we used for our metrics [12].
- **Libmelee**: An open source Python API to help make an SSBM bot [4].

Chapter 3

RELATED WORKS

As mentioned before, Computer Vision research in video games has been lacking. However, this is not to say that it hasn't been done. There are a few research papers which propose the utilization of CNNs to facilitate automatically playing the game. This section will discuss three specific papers which utilize Computer Vision concepts in video games and explain how our research differs from these concepts.

3.1 Atari - Deep Learning

Mnih et al. propose the use of a CNN to learn from video data in conjunction with Reinforcement Q-Learning to allow automatic playing of Atari games [26]. Their goal was to utilize one neural network architecture to handle playing multiple different Atari games. They utilize a concept called *experience replay* where "an agent's experiences at each time-step" is stored into the replay memory [26]. These experience samples are updated by the proposed Q-learning algorithm and an action is selected for execution by the agent. They propose that utilizing this concept randomizes the training samples making the agent select better actions based on behaviors across multiple time steps [26].

They go into detail describing the type of CNN architecture that was used. Interestingly, their decision was to convert the input image pixels from RGB color channels to a single gray-scale channel prior to CNN forward propagation [26]. Seven games were trained on the CNN architecture: "Beam Rider, Breakout, Enduro, Pong, Q*bert, Seaquest, Space Invaders" [26]. Utilizing similar hyperparameters across the training of multiple games, they needed to only tweak the reward system as the scoring varies across games as part of reinforcement learning. Values were restricted to +1, 0, -1 for positive, unchanged, and negative rewards. From extensive testing, their network achieves expert-level performance across Breakout, Enduro, and Pong. It achieves human-level performance on the game Beam Rider. And for the rest of the games, it doesn't perform very well as it requires building a strategy which was not the intention of the network [26].

3.2 ViZDoom

Kempka et al. propose a similar concept of using CNNs combined with Q-learning and experience replay as mentioned in [26] to train bots in the first-person-shooter game Doom to move, shoot, and navigate mazes [18]. As the paper introduces an API that allows the user to create bots that play the game, the information on the learning process is fairly limited. They do however mention that the learning process is similar to that of [26] and that the concept is "modeled as a Markov Decision Process" with the policies being learned via Q-learning [18]. They mention using an "ϵ-greedy policy" to select a particular action. Furthermore, a custom CNN architecture is used to analyze RGB image inputs with the output of 8 units representing different combinations of the 3 possible actions: left, right, and shoot. This learning process, architecture, and policies were for their moving and shooting setup experiment. They performed another experiment for maze navigation medkit collection and propose slightly different modifications to the previous architecture and learning process to account for better spatial navigation [18].

3.3 Stanford - Deep CNNs Game AI

Yi and Chen propose using a fully supervised learning method called imitation learning in conjunction with CNNs to help computers play video games. They define imitation learning as “pure behavioral mimicry of human actions viewed through gameplay screenshots” [9]. Their goals were not to produce expert-level agents but rather to have them operate with minimal cost. In addition, another goal was to avoid the concept of reinforcement learning based on in-game score and instead rely purely on human generated data [9].

The game Super Smash Brothers for the Nintendo 64 was the candidate for this research [9], the predecessor of our research title. The reasoning behind this was the complexity of the game as opposed to previously done research on Atari titles. Yi and Chen also trained the architecture on Mario Tennis for the Nintendo 64 for further testing of cross-game portability and limitations. A custom CNN was designed to process in-game frame data, specifically 4 sequential frames at a time. The output was a softmax vector of 30 classes representing the predicted values for the possible class of the move. The ground-truth label is keyboard input visual frames that were collected during playtime by a human player. They planned on using these predicted class scores as input data during simulated gameplay to have a character agent play against a computer player. They confirmed that after training on both games, their model performed at a top-3 move-classification accuracy of 96.0% on Mario Tennis and a 96.2% on Smash Brothers. The gameplay simulations are qualitative analysis however, but the claim is that the bot performs well and is even capable of general tracking of the computer player [9].

In contrast, our research differs from the concepts proposed by these papers in that it actively tracks characters using object detection and utilizes those coordinates to control a bot. Our CNN learns to track characters based on background, animation, and position variability with limited data. The bot is hard coded using the positional information of the tracked characters.

Chapter 4

SYSTEM DESIGN AND IMPLEMENTATION

Our system pipeline is divided into 3 chronological stages: classification, detection, and the proof-of-concept bot. Figure 4.1 depicts a diagram of our various pipeline stages. The following sections cover the full design and implementation details and constraints for the classification and detection stages. The bot is detailed as a case-study in Chapter 6.

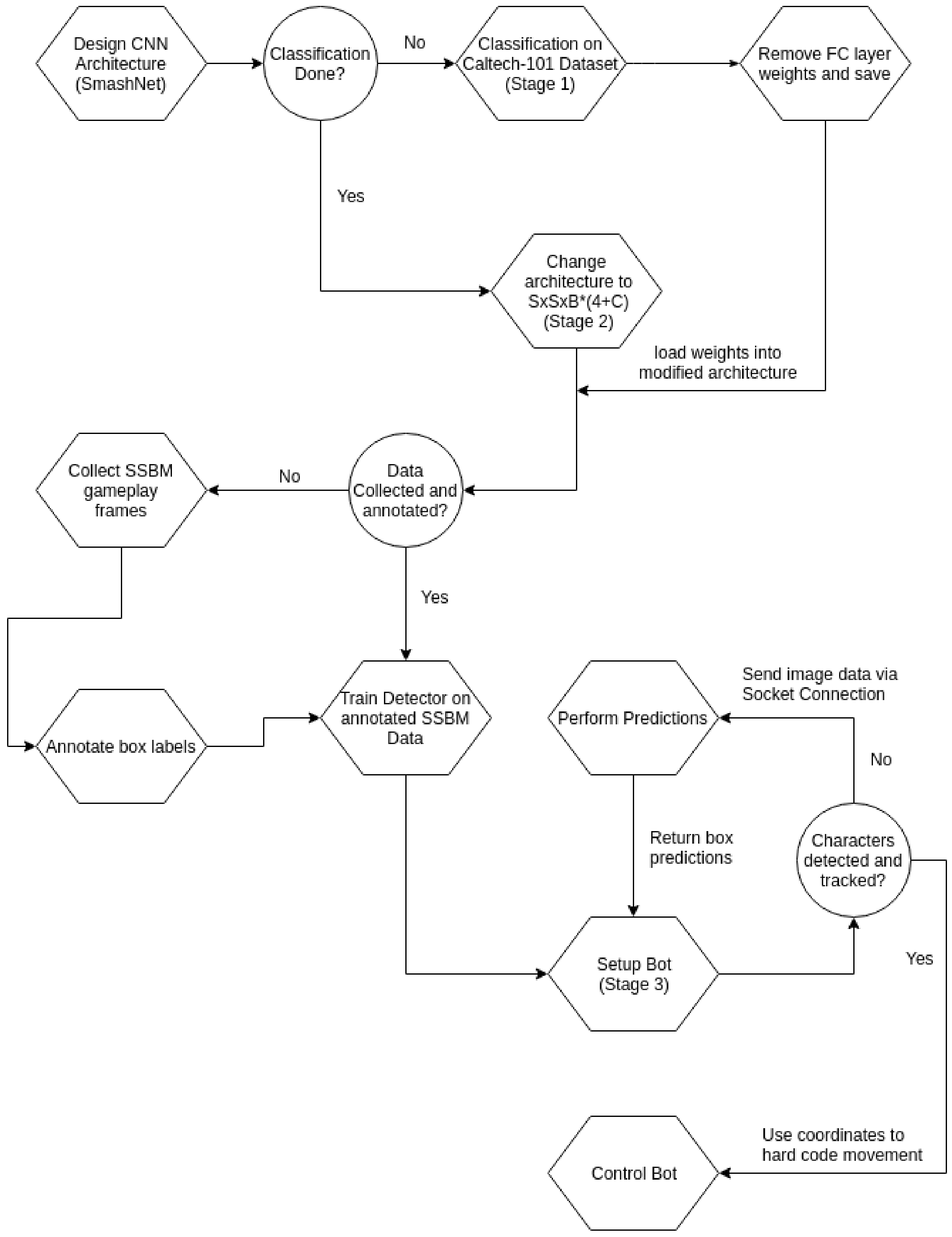

Design CNN Architecture (SmashNet)
Classification Done?
No
Classification on Caltech-101 Dataset (Stage 1)
Remove FC layer weights and save
Yes
Change architecture to SxSxB*(4+C) (Stage 2)
load weights into modified architecture
Collect SSBM gameplay frames
No
Data Collected and annotated?
Yes
Train Detector on annotated SSBM Data
Perform Predictions
Send image data via Socket Connection
No
Annotate box labels
Return box predictions
Characters detected and tracked?
Setup Bot (Stage 3)
Yes
Control Bot
Use coordinates to hard code movement

Figure 4.1: System Design Pipeline

4.1 Overall System Pipeline

The pipeline diagram is structured in the chronological way that the system was implemented. The design of a proper architecture for the purposes of use in detection came first. This architecture was pre-trained via classification on the Caltech-101 dataset in Stage 1 and then modified for detection training in Stage 2. Here it was trained on collected and labeled data from SSBM to detect and track four specific characters. Figure 4.2 displays the relationship between these 2 stages and how they fit with our system. The following section introduces the CNN design for our system.

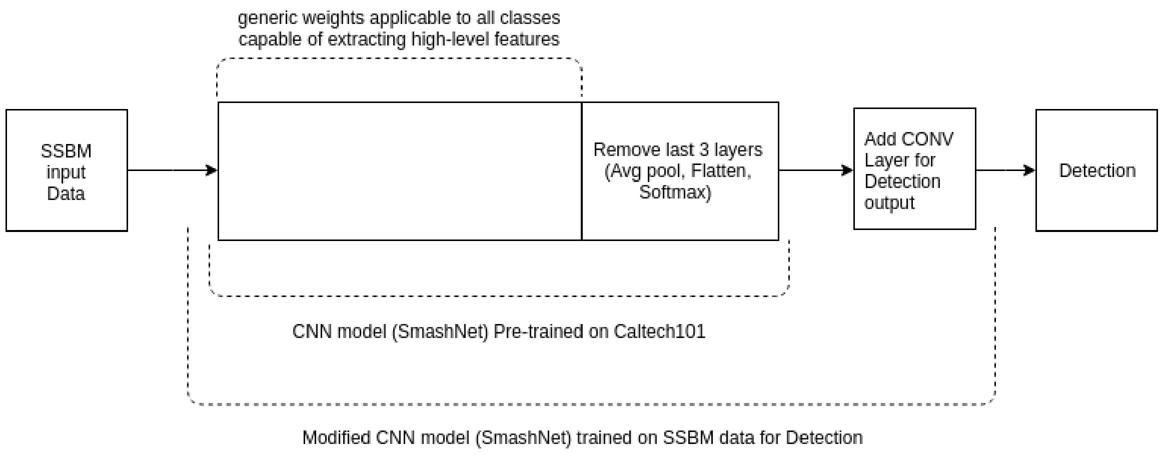

generic weights applicable to all classes capable of extracting high-level features
SSBM input Data
Remove last 3 layers (Avg pool, Flatten, Softmax)
Add CONV Layer for Detection output
Detection
CNN model (SmashNet) Pre-trained on Caltech101
Modified CNN model (SmashNet) trained on SSBM data for Detection

Figure 4.2: Visualization of classification-detection relationship with regard to our system pipeline. Classification on Caltech101 generates weights which are capable of high-level feature extraction. The model is modified to perform detection and the pre-trained weighs from classification are used for detection when training on SSBM data.

4.2 CNN Architecture - SmashNet

The first and most important step was to design a CNN architecture that met our following goals:

- **Portability**: We wanted to design a modular CNN which, *if needed*, could be changed easily to be adapted for feature extraction in several games.

- **Parameter and Computation Limitations**: Object detection on large datasets takes several days up to possible weeks to train using GPUs which are significantly stronger than our NVIDIA GTX 980. Reduced parameters means reduced floating point computations which results in a lower accuracy, however, it yields increased speed in the trade-off. As a result, we proposed a limitation of the CNN to an approximate maximum of 20 (+2 leeway) million parameters.

Our final architecture dubbed ***SmashNet*** is depicted in Figure 4.3. It was generated using an open source Python implementation known as ConvNet Drawer [41].

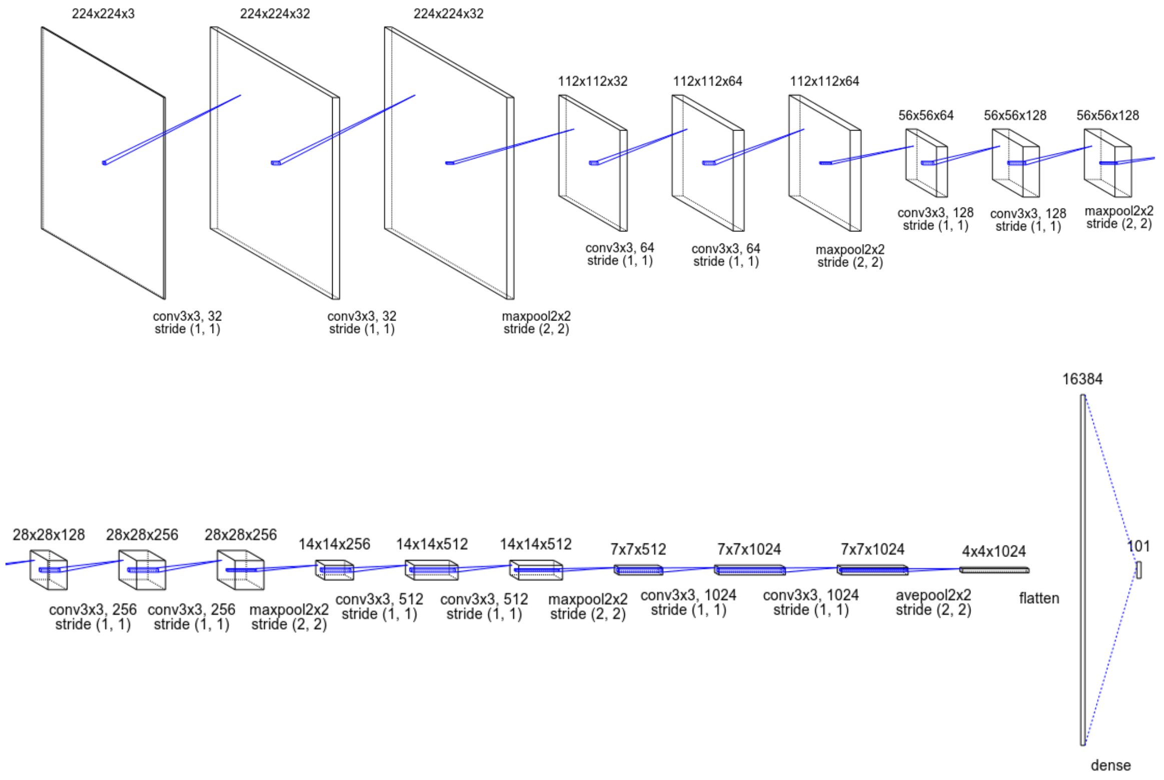


Figure 4.3: SmashNet Architecture - 13 Layers

SmashNet may be considered a 13 layer model for the purposes of this research as it contains 12 convolutional layers and a dense layer. Activation and Pooling layers are typically not counted as part of the total number of layers. SmashNet's structure and input size of 224x224x3 was heavily inspired by the VGG-16 model architecture [37]. We originally intended to allow the model to handle larger images, but decided against it as it would result in a significant increase in computation and parameters. The architecture follows the basic structure of the input layer, followed by multiple phases of 2x[CONV-RELU]-POOL, then the output layer. Our logic behind this structure is to down-sample the input image while extracting increasingly complex features across decreasing spatial sizes. Examples of complex features when performing facial recognition would be pixel representations of the eyes, nose, mouth, ears, and so on. These complex features are representative of specific classes and help during classification. In contrast, the earlier convolution layers extract features resembling edges, blobs, and other shapes that are generic to all classes. Intuitively, as the weights improve during loss function minimization and network convergence, the extracted features become better. With this in mind, we chose to double the number of filters after every down-sampling operation performed via pooling. This way, we extract more features with increasing complexity that are representative of the classes in the dataset. This follows our reasoning behind starting with 32 filters in the first set of 2x[CONV] layers as edge and blob detection is fairly simple and does not require many filters to do so. We stopped down-sampling at 4x4 input and maximized our filter count at 1024 filters in the final set of convolutions as empirically, it performed better than a 512 filter maximum while also reaching the peak of our parameter limit.

We chose to use RELU non-linear activation as it was the most commonly used activation function in a majority of CNNs. The pooling layer typically follows an activation layer in our architecture and down-samples the current input dimensions. Max Pooling with a stride of 2 and a field size of 2x2 was used for each instance of pooling except the very last. The last pooling layer performs the average pooling operation with a 2x2 window size and a stride of 2 resulting in an output size of

4x4x1024. This layer is then flattened into one dimension with all pixels of each feature map - which results in 16384 (4x4x1024) inputs across a single dimension. These 16384 inputs are fully connected to the final FC layer which uses the Softmax activation function to output a vector of predicted classes for the input.

SmashNet also draws inspiration from the VGG-16 architecture [37] in its primary use of 3x3 convolution filters. Convolution filters extract features directly proportional to the size of the filter. Other state of the art architectures use different sized convolution filters. The original inception v1 architecture used both 3x3 and 5x5 convolutions in its inception blocks [38]. A 5x5 convolution is a significant increase in the number of parameters and floating point computations. To understand this idea, an example is outlined below. Let's assume that the convolution operation across a layer is as follows:

$$\text{224x224x64} \longrightarrow \text{CONV [64 Filters, 5x5x64, Same Padding]} \longrightarrow \text{224x224x64}$$

For this convolution, the number of parameters is: 64 (input channels) *5*5*64 = 102,400 parameters. The number of floating point computations can be calculated as: (224*224*64)*(5*5*64) = 5,138,022,400 computations. In contrast, using 64 3x3x64 filters will give us 64*3*3*64 = 36,864 parameters, and (224*224*64)*(3*3*64) = 1,849,688,064 floating point computations, which are both significantly less than their 5x5 filter counterparts.

But how do we account for larger and wider features that the 5x5 gives us? The answer is that we don't. However, advances have recently been made with regard to CNN optimization using a concept called convolution factorization where larger filters (example: 5x5) can be convolved as multiple smaller filters (2 3x3) [39]. We did not use this idea as we felt that 3x3 filters perform well enough that larger filters just aren't necessary. In addition, since stacks of 3x3 filters increases the number of parameters and computations, we wanted to avoid adding layers we felt were just not needed. Networks such as VGG-16 which contain 138 million parameters perform exceptionally well in classification tasks [37], but due to the large magnitude of parameters, would take far too long for detection with our GPU. Our CNN uses a total of 20,524,933 parameters which fits appropriately with our original design constraints of the 20 (+2) million parameter requirement. The simplicity of the architecture also makes it viable to modify for various classification tasks.

4.3 Classification Pre-Training (Stage 1)

Classification is the beginning of the first stage of our pipeline. We perform this training to produce high-level feature extracting weights that are used during object detection (Figure 4.2 details this idea). Training object detection models without pre-trained weights is an extremely slow process and has several issues. Time is a major issue as the existence of pre-trained weights allows for faster network convergence, thereby decreasing the amount of training time. Without these weights, the model trains from scratch, and can take several days for it to produce potentially feasible weights. In addition, our detection dataset is minimal and only comprised of 2296 images and annotations of the different objects. Due to having a minimal training dataset, the network might never converge, resulting in potential loss values which are far too large. Pre-training our model and producing optimal weights addresses these issues and so we chose to perform classification.

4.3.1 Caltech 101 Dataset

The main rule of pre-training is to use a large dataset that has a wide variety of distinct classes so diverse feature extracting weights (filters or filter values) can be produced. The Caltech 101 dataset was chosen for our purposes as it contains 101 useful distinct classes with 40 - 800 images per class with an extra “background” class [11]. This dataset is significantly more complex than utilizing a possible dataset from SSBM as it contains a wide range of distinct classes. We deemed that using this dataset was more practical than generating our own dataset for pre-training as it would take several days to put together a diverse dataset with enough distinct images. To prepare the Caltech101 dataset for classification, we removed the background class which left us with 8677 images across the 101 remaining classes and then proceeded to the training process.

4.3.2 Training Process

Preprocessing was done by resizing the images to match our model's 224x224x3 input. One-hot encoding vectors of actual class labels were generated for each training and testing image. These images and labels were split into 85% training (7375 images) and 15% validation (1302 images) data for the purposes of training and evaluating the performance of the model. The optimization algorithm used was SGD Nesterov with a tuned learning rate α of 0.01, a momentum coefficient of 0.9, and a learning rate decay of $1e^{-6}$. Lower values of α resulted in minimal loss improvement for both training and validation loss whereas higher values of α improve training loss but not validation loss. The momentum coefficient and decay rate are the default values used by CNNs [27]. The decay rate was used to decrease α if the training loss improves rapidly with minimal validation loss improvement. The loss was set to categorical cross entropy which is the Multi-Class classification loss for a class size greater than 2. Batch size during training was set to 32 so the model performs back propagation after the loss has been calculated over an average for 32 images at a time. We chose to run the model for 20 epochs, representing the number of times the model sees the entire dataset.

Allowing the model to learn from the same data 20 times could result in overfitting where the model may learn very specific features that prevent it from being capable of correctly classifying an unseen image of the same class or even a modified variant of the same image. To prevent this, we introduce image augmentation where the images are modified via certain methods so that the model never learns multiple times from the exact same image. Two methods of augmentation were randomly applied to the data: zooming and flipping. Zooming was implemented by a factor of 20% and flipping was performed across the vertical axis. In addition, a secondary method we used to reduce overfitting is by only saving model weights where the validation loss decreases to a new minimum. This supports the idea that the model learned better feature extracting weights as the validation images are unseen data and a lower loss indicates a greater classification accuracy. Once training is completed, the final saved weights with the lowest validation loss are utilized in detection.

4.4 Detection (Stage 2)

As depicted in Figures 4.1 and 4.2, the SmashNet architecture was modified slightly for the purposes of performing detection in Stage 2. The last 3 layers of our architecture - Average Pooling, Flatten, and FC were removed from the architecture. As the input to the detection model is 416x416 sized images (as explained in Section 2.3.3), the model's resulting output after removal of the last 3 layers is 13x13x1024. Here, the following layer is added:

$$\text{13x13x1024 (input)} \longrightarrow \text{CONV [45 Filters, 1x1x1024, Same Padding] [6]}$$

A 1x1 convolution is an in-place convolution which is done purely for the purposes of altering the depth dimension (channels or feature maps) of the input. With this convolution, our input changes from a 13x13x1024 to 13x13x45 output. The 45 feature maps are representative of the 5 anchor boxes * (4 bounding box coordinates + 1 object confidence + 4 class probabilities). The overall goal was to detect 4 specific characters from SSBM, which are the 4 classes. The output is reshaped to 13x13x5x9 as explained above so that there are 5 anchor box predictions with the 9 values for every cell in the 13x13 grid. The detection model has 18,916,173 parameters which falls well under our 20 (+2) million constraint.

In addition to the model changes, we needed anchor boxes of variable size to perform multiple detections per cell. These boxes were obtained from [6] and are the anchors used for training of the popular MS COCO dataset. The anchors are: (0.57273, 0.677385), (1.87446, 2.06253), (3.33843, 5.47434), (7.88282, 3.52778), (9.77052, 9.16828) [6]. The coordinate pairs multiplied by 32 represent the width and height of each anchor box. As an example, the first coordinate pair's width is 0.57273*32 = 18.33 pixels. The reason for this computation is that each grid cell is 32 pixels as a result of dividing 416x416 images into a 13x13 grid. There are 5 pairs of coordinates and therefore 5 anchor boxes of variable sizes used to detect objects.

4.4.1 Data Collection

As mentioned in prior sections, we are working with the 2.5D fighting game SSBM. A dataset of labeled images was not readily available and so one had to be manually composed. SSBM was emulated through the use of the Dolphin[1] emulator for the purposes of data collection and bot testing. Our goal was to collect images for 4 characters. We selected Ness, Kirby, Ganondorf[2], and Fox for the following reasons:

- **Ness:** Ness is short in stature and utilizes weapons to attack which makes his character have highly variable animation frames. Figure 4.4 displays Ness from a paused gameplay frame.

[1]For the sake of fair use, we would like to mention that a legal copy of the game is owned by the author. The emulation software and game are used specifically for academic reasons - for data collection and bot testing in pursuit of Computer Vision research.

[2]Ganondorf may also be referred to as Ganon.

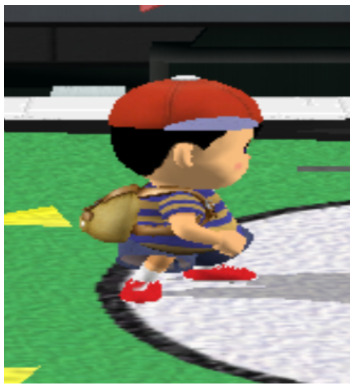


Figure 4.4: Paused gameplay frame of Ness on Pokemon Stadium (Stage)

- **Kirby:** Kirby was selected for his simplicity as a character. His small stature and round shape makes it interesting to see how easily he could be detected as opposed to a more complex character. Figure 4.5 displays Kirby from a paused gameplay frame.

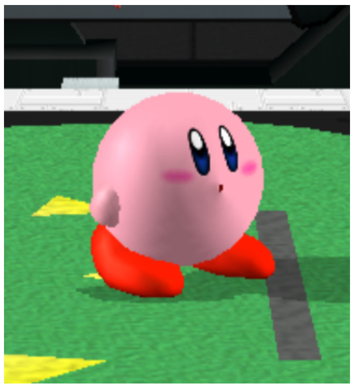


Figure 4.5: Paused gameplay frame of Kirby on Pokemon Stadium (Stage)

- **Ganondorf:** Ganondorf is arguably the largest character in the game due to his height and cape. The erratic movements of his cape and also his moves which emit purple colored auras make him a highly variable character. Figure 4.6 displays Ganondorf from a paused gameplay frame.

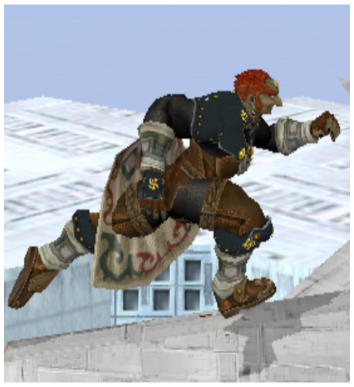


Figure 4.6: Paused gameplay frame of Ganondorf on Corneria (Stage)

- **Fox:** Fox was selected for a different purpose than the other three. While he does have a variability in his moves, the main reason he was chosen was due to his popularity as a pick by the top professional players. Figure 4.7 displays Fox from a paused gameplay frame.

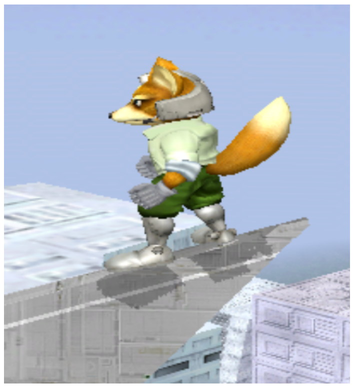


Figure 4.7: Paused gameplay frame of Fox on Corneria (Stage)

Several gameplay videos were recorded, reaching approximately an hour of footage. These videos were broken into frames[3] and then selected based on the clarity of the characters. If the characters were obscured by backgrounds, items, or parts of the stage, those respective images were omitted. The images selected for detection were labeled using a software called LabelImg which allowed drawing of bounding boxes

[3]Frames may also be referred to as "images".

across the image [3]. LabelImg generates XML files containing the respective class name of the object, followed by the bounding box coordinates of the object.

We narrowed it down to 488 images of Ness, 485 of Kirby, 439 of Ganondorf, and 488 of Fox where each of these characters were the only labeled ones in those respective images. However, 396 labeled images were used where multiple characters could be present in the image. This led to a total of 2296 labeled images. The final numbers correspond to 811 images of Ganondorf, 865 images of Ness, 848 images of Fox, and 816 images of Kirby. The final dataset took approximately 30 hours to create.

4.4.2 Training Process

The input to the model takes is a combination of images alongside the ground truth bounding box coordinates for each character present in the image. The data is split into 80% training and 20% validation data for the purposes of checking the validation loss after every epoch similar to that of the classification training process. After the model learns from the batch of images, the YOLO loss function is used to calculate the loss, and the weights are updated by back propagation. This detection training is done in two steps as detailed below.

Warm-up Training

First is the pre-training step which involves loading the classification weights into the modified detection model and then training the model on SSBM data for 4 “warm-up” epochs. However, as the last 3 layers were replaced for a new convolutional layer as outlined earlier in Section 4.4, we randomly initialize the filters for this layer along with the bias.

The training is performed 10 times indicating that each batch of images and respective annotations is processed by the model 10 times. Our 2296 images were divided into 1837 training and 459 validation data for the purpose of the 80-20 split as explained in the previous section. These 1837 images are split into batches of 16 which results in a total of 115 batches. These batches are trained 10 times and thereby the model learns from 1150 batches of data per epoch. Warm-up training was recommended as a means of improving precision and was proven to do so empirically [6]. The optimization algorithm used for training was the Adam algorithm with a tuned learning rate α of 1e^{-4}. Lower values of α took very long training times to yield proper detections and higher values of α caused issues with gradient calculation resulting in values that are too large or too small. The hyperparameters were set to the default values used by CNNs: β_1 was set to 0.9, β_2 was set to 0.999, and ϵ was set to 1e^{-8} [27]. Penalties for low confidence values of predicting the existence of an object in a grid cell was set to a factor of 5.0, and the rest - incorrectly predicting box coordinates, or class, or low confidence values for predicting that an object doesn't exist were all set to a factor of 1.0. We value the correct prediction of an object in a grid cell as significantly more important and therefore influence the loss calculation in such a way that it penalizes the loss heavily if we have low confidence values for an object that exists. Confidence values less than 60% are penalized this way. This concept is obtained from [6] and is a unique approach as the standard YOLO loss function does not seem to penalize low confidence detections of existing objects with an extra λ_{coord} factor. The IOU threshold during training and validation was set to 0.5 following the concept of a proper detection.

The goal for detection as with classification, is to decrease training loss and validation loss as the model learns. To do so, we wanted to make sure that the model was seeing variability in the images and ground-truth boxes and therefore the images were heavily augmented using the open source Python library imgaug [17]. There were several types augmentations used including different types of blurring - specifically Gaussian, Medium, and Adaptive blurs [6], [17]. The images were sharpened at random and the brightness was increased or decreased by +10 or -10 as an additive factor or 50% to 150% as a multiplicative factor [6], [17]. Gaussian Noise was yet another augmentation used [6], [17]. Dropouts were performed where 10% of pixels would be removed randomly from the image. Finally contrast normalization was performed where the contrast of the image was altered [6], [17].

Post Warm-up Training

This training was the final step for object detection predictions. The weights collected from this training are used during prediction for just standard images, pre-recorded video, and live-video for real-time bot responses. Post warm-up training is almost identical to warm-up training with the exception of a few modifications. First we load the model with the weights learned during warm-up training. Yet again we randomly initialize the final layer weights and biases. Finally, the number of epochs is changed to 20 so that the model may train longer. All other factors remain constant as the model is trained. We end up collecting data during this stage of the training and validation loss so that we may evaluate the performance of our model.

An important implementation detail to note is that our saved model weights are based directly on the training loss for both warm-up and post warm-up training and not the validation loss, which is opposite way in which we performed classification. Our reasoning behind this decision was a result of empirical testing. The model seemed to reach its lowest validation loss in earlier epochs but did not yield proper detections. However, when increasing the number of epochs to 20 and allowing the weights to save based on training loss, the model seems to perform better. The training loss decreases by a larger factor, but the validation loss only increases by a small factor indicating minimal overfitting which we concluded was acceptable.

A second implementation detail to note is that when predictions are being done on test data, we utilized a fixed Non-Max Suppression IOU threshold of 0.3, and a confidence threshold of 0.3. The resulting boxes post NMS are drawn onto the image with the respective class name and confidence value.

Chapter 5

SYSTEM EVALUATION

We evaluated our system based on the results of the implemented training methods as explained previously. The classification model was evaluated first, followed by the detection model. We only moved on to the implementation of the next stage of the design pipeline once the previous stage had been evaluated to produce what we deemed successful results. To summarize the two stages, the following steps had been performed:

- **SmashNet:** We generated a CNN model, SmashNet, for the purposes of performing detection on SSBM data.
- **Classification Pre-Training:** The SmashNet model was first subjected to pre-training via classification on the Caltech101 dataset with a goal of generating high-level feature extraction weights that could be used for detection.
- **Detection:** The SmashNet model was modified as explained in Section 4.4, to output detection results and then trained on the custom SSBM dataset of 4 classes to detect object instances of those classes during gameplay.

This section details the results and appropriate evaluation of the classification and detection stages of our pipeline.

5.1 Classification Pre-Training

As mentioned earlier, classification training using our model was done on the Caltech-101 dataset with the goal of minimizing the validation loss. The trained 20 epochs are represented as 0-19 in our analysis. The validation accuracy and training accuracy of our model is depicted in Figure 5.1.

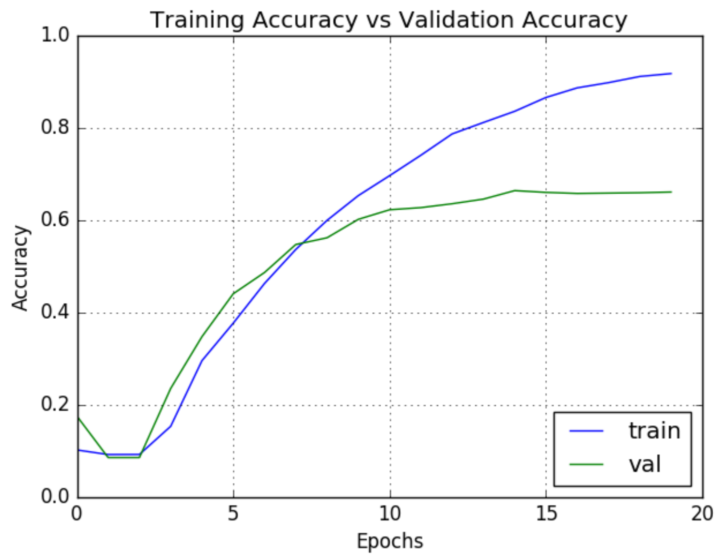


Figure 5.1: Graph of Training Accuracy vs Validation Accuracy during Classification of Caltech-101 by SmashNet

As the graph illustrates, both training and validation accuracies continue to increase as epochs increase. The validation accuracy seems to peak earlier around epoch 14 at about 66% and stabilize after. The training accuracy reaches approximately 92% by epoch 19. Accuracy can potentially be used as an indicator for model performance, however, it does not provide any indication of overfitting and as a result, can be misleading. For this reason, we analyze the training and validation losses rather than the accuracies. We do however use the training and validation accuracies as supporting evidence to check how well they correlate with their respective losses. The training and validation losses are depicted in Figure 5.2.

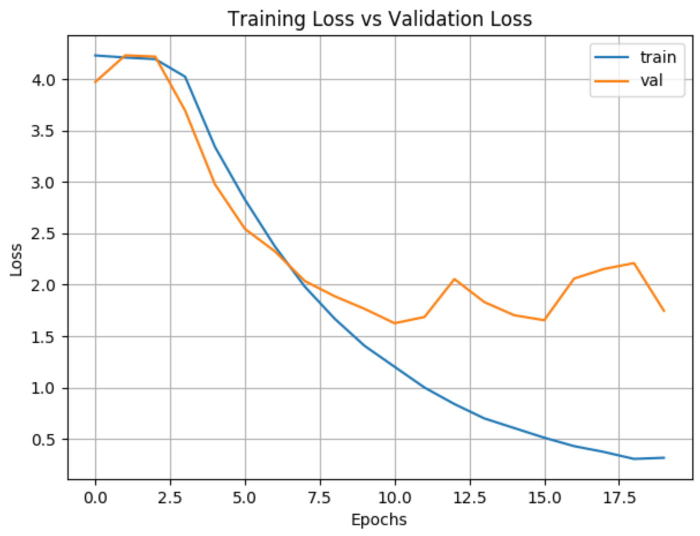


Figure 5.2: Graph of Training Loss vs Validation Loss during Classification of Caltech-101 by SmashNet

The loss graph shows that the minimum validation loss was reached at epoch 10, with a loss value of 1.62. While the training loss continues to decrease until epoch 19, the validation loss never reaches a new minimum after epoch 10. This indicates that the model is overfitting on the training data and therefore getting worse at classifying unseen data. While the validation accuracy seems to reach its absolute peak at approximately 66% at epoch 14, it actually stops drastically increasing at epoch 10, at 62.29% as indicated in Figure 5.1. The rate of increase is diminished which seems to support the theory that the model is overfitting. We reason that a few previously, incorrectly predicted inputs have been predicted correctly which results in the accuracy increase. However, of those same incorrectly predicted inputs, several predictions must have become worse, causing an increase in validation loss and clearly indicating overfitting. The saved weights of this model were at epoch 10, when the validation loss had reached its best minimum value.

Using the saved weights, we were interested in seeing the errors made by the model during classification of the Caltech dataset. Figure 5.3 depicts the top 20 classes with the highest error rates during classification of the validation data.

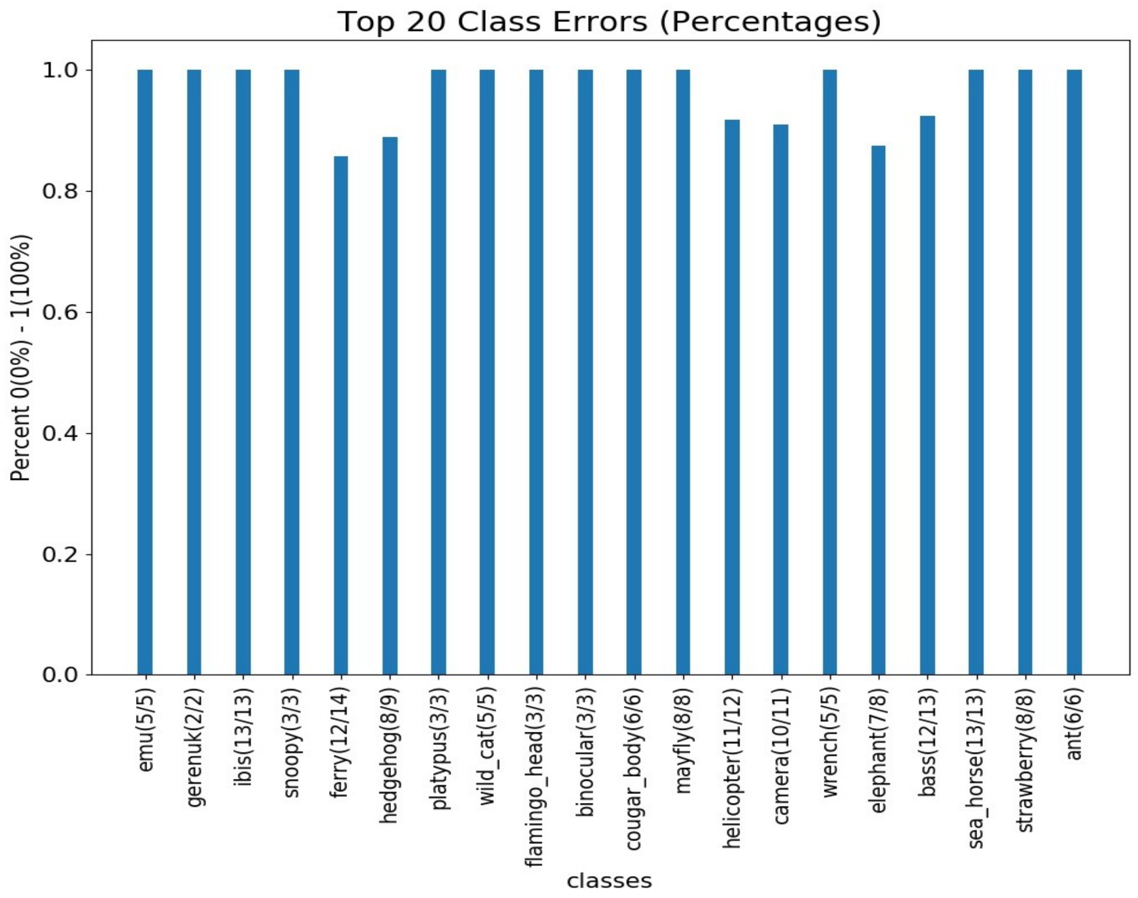


Figure 5.3: Graph of SmashNet top-20 class errors during classification of validation data

The error rate across the top 20 is very similar. While there are several instances of 100% mispredictions, Ibis and Sea_Horse were clearly the worst with 13/13 mispredictions each. Ferry was the class with the lowest in the top-10 with an error rate of 12/14 mispredictions. We expected the top two classes to be: “Faces” and “Faces_easy” due to their high similarities and complexity. However, Faces and Faces_easy both yielded only 2 mispredictions each out of 75 and 54 images respectively. Further results across all validation images can be found in Appendix A, Table A.1. Additionally, classification accuracies for all Caltech101 classes is represented in Appendix A, Figure A.1.

We were also interested to see what types of features the filters could learn. To do this, we generated several gray images with some noise. We passed these images into the model and through instances of forward propagation, obtained the activation layer loss at layers 1, 6, and 12. We performed back propagation and applied 20 steps of gradient ascent to increase the loss as much as possible. Maximizing the loss at a particular activation layer activates the filters of that layer which can be used to visualize the learned features [43]. Figure 5.4 depicts a visual of a few of our model's filters across 3 different layers.

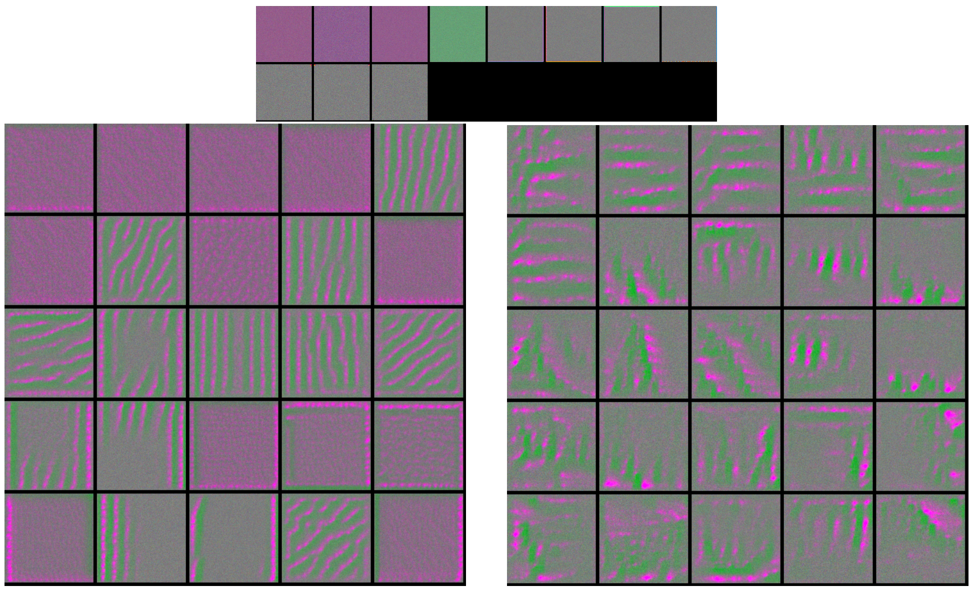


Figure 5.4: SmashNet filter visualization (Top: layer 1 filters, bottom left: layer 6 filters, bottom right: layer 12 filters)

Filters in layer 1 are simple and haven't picked up any features outside of a few color patterns. Filters in layer 6 seem to have learned several types of edges such as horizontal, vertical, and edges of other angles. Filters of layer 12 appear to have learned very specific patterns, most of which are more complicated than its layer 6 counterpart. As expected, these filters support the idea that the complexity of the learned features increases across later layers of the model. Neurons typically represent highly abstract features that are not visually representative of the input data. However, through different forms of regularization, it is possible to allow filters to have a more representative view of the input images [43]. This concept was not implemented in our research as we felt that we did not understand it well.

We also generated visualizations of the convolution and activation layer outputs of our model to understand how the input changes as it passes through the model. To do so, we passed in a single image of an accordion from Caltech101's accordion class into the model. A subset of the output feature maps for layer 1 and layer 6 convolution and activation operations are depicted in Figures 5.5 and 5.6 respectively.

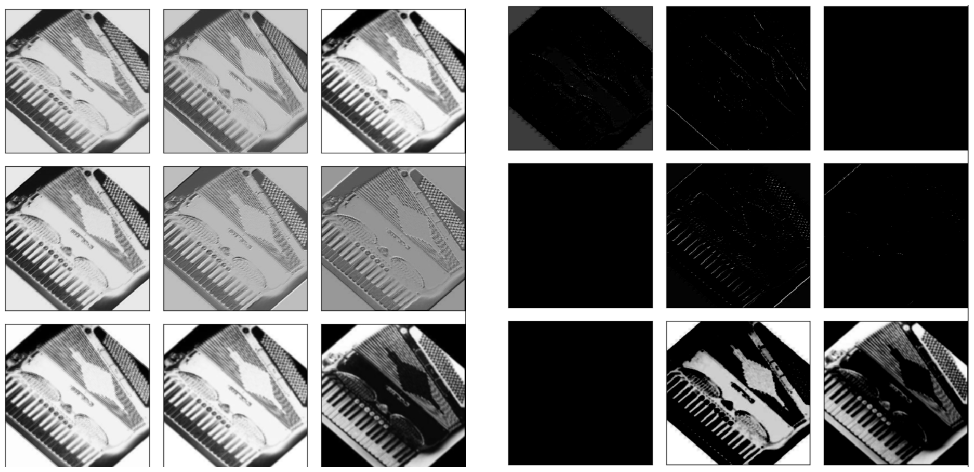


Figure 5.5: SmashNet layer 1 feature map visualizations of Accordion image (left: convolution layer, right: activation layer)

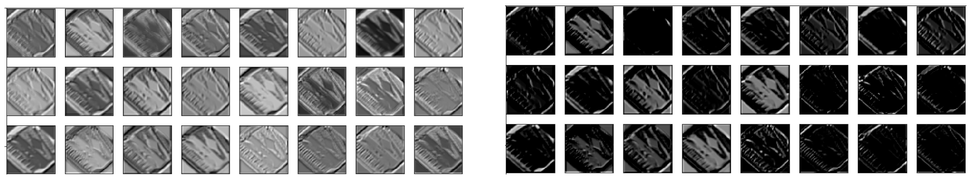


Figure 5.6: SmashNet layer 6 feature map visualizations of Accordion image (left: convolution layer, right: activation layer)

Overall, the loss curves and visualizations gave us plenty of insight on the performance of our model. We took into account the simplicity of our model and the detection task and concluded that the classification weights yielding a 62.29% validation accuracy with a 1.62 loss was acceptable. As the Caltech101 dataset is significantly more complex than our detection dataset, we did not feel that it was necessary to improve the model. However, we do suggest potential model improvements as future work in Chapter 7.

5.2 Detection

An example frame detected by our model is shown in Figure 5.7.

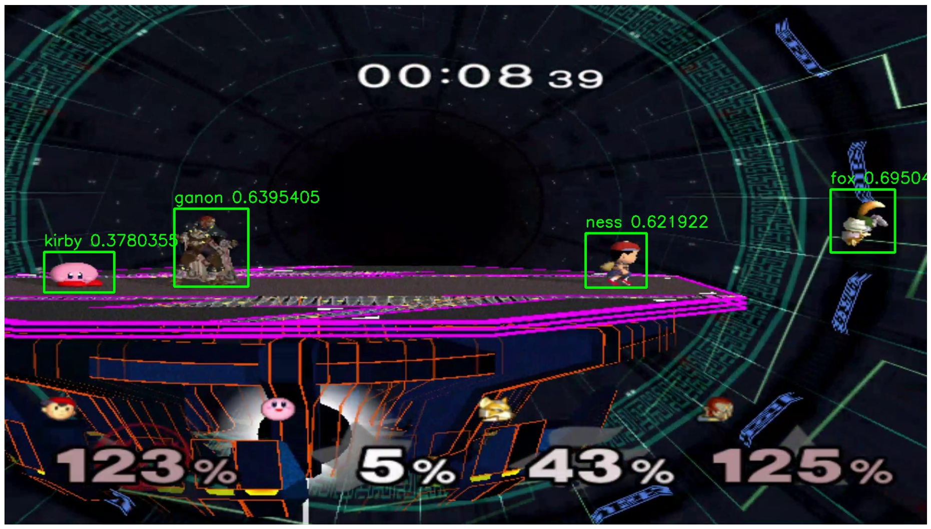


Figure 5.7: SmashNet detected frame of SSBM Characters (from left to right: Kirby, Ganondorf, Ness, and Fox)

Our detection model was trained with the goal of lowering both the training and validation loss. The trained 20 epochs are represented as 0-19 in our analysis. Figure 5.8 depicts the graph of both the training and validation loss as a result of post warm-up training.

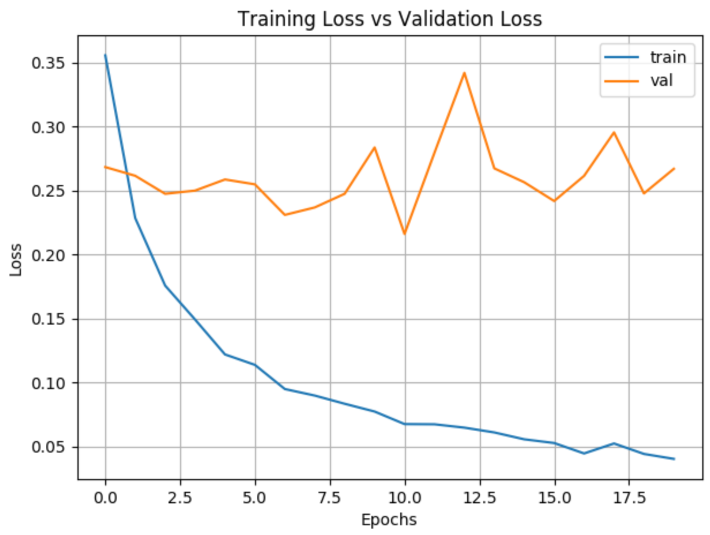


Figure 5.8: Graph of Training Loss vs Validation loss during Post Warm-Up Training of the detection model

Here, the weights yielding the best validation loss were generated at epoch 10 of training time, however, as mentioned earlier regarding the design of our detection system, we chose to save the weights based on the training loss. Since this loss continued to decrease until epoch 19, we saved the weights during epoch 19 at a minimum training loss of approximately 0.04 and a validation loss of approximately 0.27. As indicated by the curve, the validation loss increases after epoch 10 indicating a certain amount of overfitting being performed by the model. By epoch 19, the validation loss had only increased by a total of 0.05 from its last minimum of approximately 0.22 at epoch 10. We deem this minimal level of overfitting acceptable as the model visually performs better with the generated epoch 19 weights.

The rest of our analyses were performed through detection on images using our fully trained model (post warm-up trained weights). The confidence threshold was set to 0.3 therefore, all box predictions which did not meet this confidence value were removed. NMS was applied with an IOU threshold of 0.3 to filter overlapping boxes.

As explained prior, the method of evaluating a detection model is via MAP. Figure

5.9 depicts the AP for various classes from analysis of detections performed on the validation data.

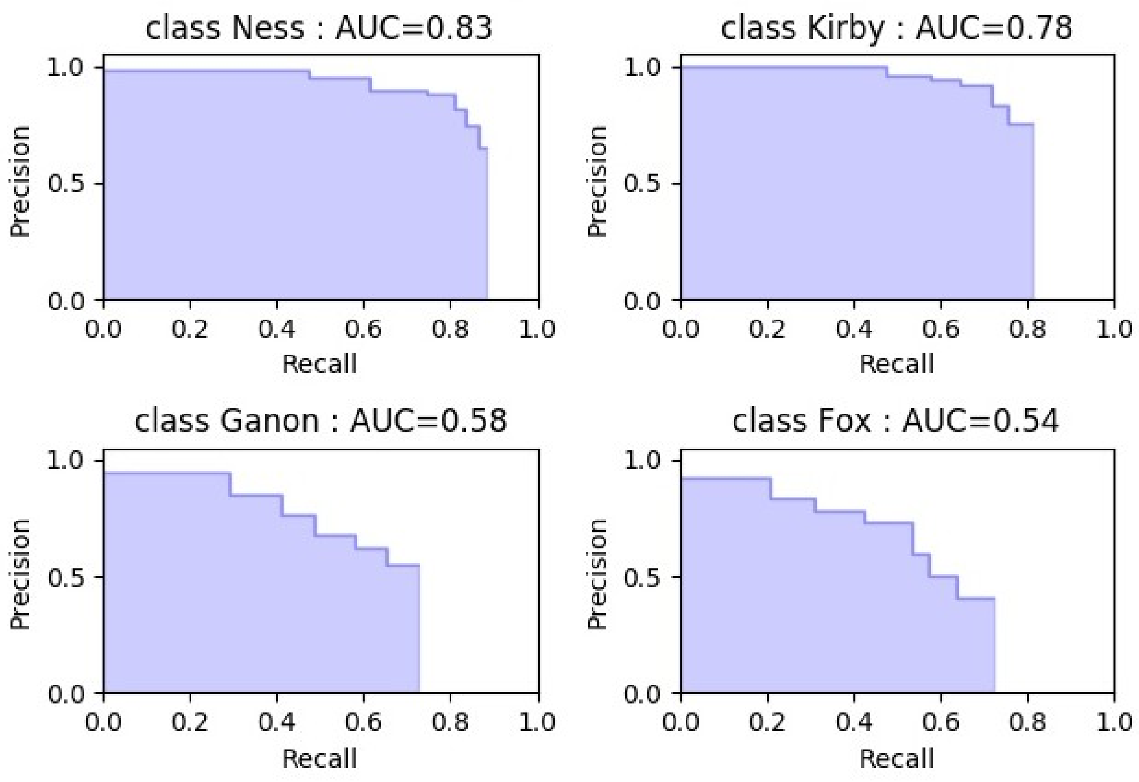


Figure 5.9: Average Precision based on detection results on the validation data for Ness, Kirby, Ganondorf, and Fox, IOU = 0.5 (standard)

The AP is clearly superior for Ness indicating that the detection system seems to recognize his character more often when analyzing the top-n detections. Kirby seems to follow second, Ganondorf as third, and Fox as last. However, all 4 characters have an unreachable recall beyond a certain point. As the recall is the factor of true positives with regard to total positives, we surmise that the detection system does not manage to detect all object instances of any individual class successfully. This is especially true with regard to Fox and Ganondorf where their individual recalls are well below 80% indicating that a significant portion of those characters are undetected or misdetected. Precision as expected decreases as recall increases. This is due to the increase of false positives as n increases with regard to the top-n

detections. The precision decreases slowly for Ness and Kirby indicating that they are typically classified with minimal false positives among the top-n detections. However, Ganondorf and Fox have a much higher rate of decrease with regard to precision. This indicates that the false positive rate is much higher as n increases for these two characters. Overall we can take the area under the curve between precision and recall as an indicator of performance accuracy for each character. Ness performs at 83%, Kirby at 78%, Ganondorf at 58%, and Fox at 54% accuracy. Taking the mean of these classes gives us the MAP value of 0.6825, indicating a total detection accuracy of approximately 68.25%.

Repeated testing with different variations of validation data yields similar results for Ness and Kirby with variances of +/-4%, however, Fox and Ganondorf vary with results of approximately +/-5-9%. This also agrees with the concept of high variability in animation frames for Fox and Ganondorf as some frames yield better detections than others. Regardless of this difference, the overall detection rate seems to differ by +4/-2%. The lowest detection rate generated was a 0.66 and the highest as 0.72. This detection accuracy range was also deemed acceptable for our purposes.

We also randomly sampled 20% of the total images in the dataset and ran them through the model to analyze the number of proper detections alongside error rates as a separate metric from MAP. Table 5.1 depicts a confusion matrix of the detection results of this experiment.

Table 5.1: Confusion Matrix of validation data detection IOU = 0.3

		Predicted				
		Ness	Kirby	Ganon	Fox	None
Actual	Ness	177	4	3	7	14
	Kirby	3	132	1	5	24
	Ganon	2	0	137	6	32
	Fox	3	1	2	152	19
	None	61	37	84	122	0

While a confusion matrix has not been accepted as a proper evaluation method for detection, we can still obtain useful information from it. An example of this is the high false positive rate for Fox and to a lesser extent, Ganondorf. It seems that the system detects both characters in several instances where no character is present (122 for Fox and 84 for Ganondorf). This is a logical reason for the rapid rate of decrease in precision for both characters as opposed to other characters such as Ness and Kirby when looking at the AP for each character (Figure 5.9). We make the assumption that the limited dataset is the reason for high variability characters like Fox and Ganondorf to produce a higher false positive detection rate. Ness and Kirby are fairly simple characters which is likely the reason for their lower false positive rate.

To derive these results, we used an IOU threshold of 0.3 to deem a proper detection. We set a lower IOU threshold as we felt that the dataset was limited as there were several unique variations of each character but not enough overall data, making it hard to learn all the necessary features during training. We also determined empirically that a threshold of 0.3 performed significantly better than the standard threshold of 0.5. IOU thresholds of 0.5 yielded detection accuracies barely above 65% as indicated in Figure 5.9 via the recall curve for both Fox and Ganondorf which seemed to correlate with several instances of lower confidence detection. This is a logical conclusion as decreasing the IOU threshold increased the recall value for both characters to well over 65% based on table 5.1 when comparing their respective true positives and false negatives. Fox has a recall value of 85.9% and Ganondorf has a recall value of 77.4%. This agrees with our earlier assumption as characters like Ganondorf and Fox seem to be detected but with less confidence due to their higher variability in animation frames. So lowering the IOU gave us a larger percentage of accurate detections.

For the purposes of the bot, we increase the confidence and NMS thresholds to 0.45 which appears to make a visual difference of less false positive detections. This was done to remove false positive detections of Ganondorf and Fox as they typically are recognized with lower confidence than Ness and Kirby. Admittedly, it likely misses a few detections of Ness and Kirby as well, but the new AP indicates, with a 0.79 for Ness and 0.74 for Kirby, that it doesn't affect their success rate by much.

From evaluating our model we come to two conclusions. First, with the very limited data we have on the 4 SSBM characters, we feel that this detection rate of 68.25% across all 4 characters is acceptable. Second, we feel that this training pipeline and model can be applied to any detection task with regard to games as long as there is enough data to train the detection model. The phrase "enough data" is fairly subjective but we feel that using over 2000 annotated object instances per class is likely enough for high detection rates (>80%). The other option, if there is a lack of data, is to pre-train the model on a large detection task like MS COCO rather than a classification task as we did. This however requires significantly more computational power as MS COCO has 80 classes with over 330,000 images, and 1.5 million object instances [24].

5.3 Comparisons between Classification and Detection Results

We decided to investigate the relationship between the classification and detection results to see if there were any possible correlations between better classified classes and Ness and Kirby's high detection rate. We felt that this was a useful study as the pre-trained weights used in detection are generated via classification. Upon further analysis of the classification results (Appendix A, Table A.1), we noted a few observations. The classes yin_yang and trilobite are classified with 90.9% and 85.7% accuracies respectively. These two classes are both highly similar in shape with a large round figure. It stands to reason that since the pre-trained classification weights performed well in classifying them, that they would be able to extract ideal high-level features for Kirby during detection. This could also be supported by the high classification accuracies for Faces_easy and Faces at 96.3% and 97.3% as they also consist of round shapes. With Ness, it is harder to pinpoint the reasoning behind the detection model's success and classification results. We attribute the success to a combination of a multitude of classes. Ness also possesses a round head which correlates with the same classes that were mentioned for Kirby. His body forms a fairly square-like object with two small legs. This structure seems to most closely resemble the windsor_chair class which produces a fairly high classification accuracy of 77.8%. Ganondorf and Fox on the other hand are fairly complicated characters with diverse body shapes. Ganondorf's shape seems most comparable to that of the kangaroo class which only yields a poor classification accuracy of 25%. These classes of similar shape were determined by comparing silhouettes of Caltech101 classes and the SSBM character shapes. We could not point to any specific combination of accurately classified classes that correlate to Fox. We attribute the lower detection accuracy of these two highly variable classes to two possible reasons: poor classification weights or insufficient feature coverage by the Caltech101 dataset.

Chapter 6

CASE STUDY: PROOF-OF-CONCEPT BOT

This section details the third and final stage of our design pipeline - the SSBM Bot. We designed this bot as a proof of concept for the possibility of using real-time object detection to track characters and produce an adaptive game AI. We used an open source Python implementation called libmelee [4] to allow the Dolphin emulator to accept programmable controller inputs thereby allowing us to control characters in game. For the sake of simplicity, we chose to have only two characters fight - Ness (enemy player) and Kirby (bot). Having the maximum possible of 4 characters introduces multiple variables and options for the bot which we did not want to explore. The goal wasn't to design an ideal bot, but rather, as mentioned before, a proof of concept.

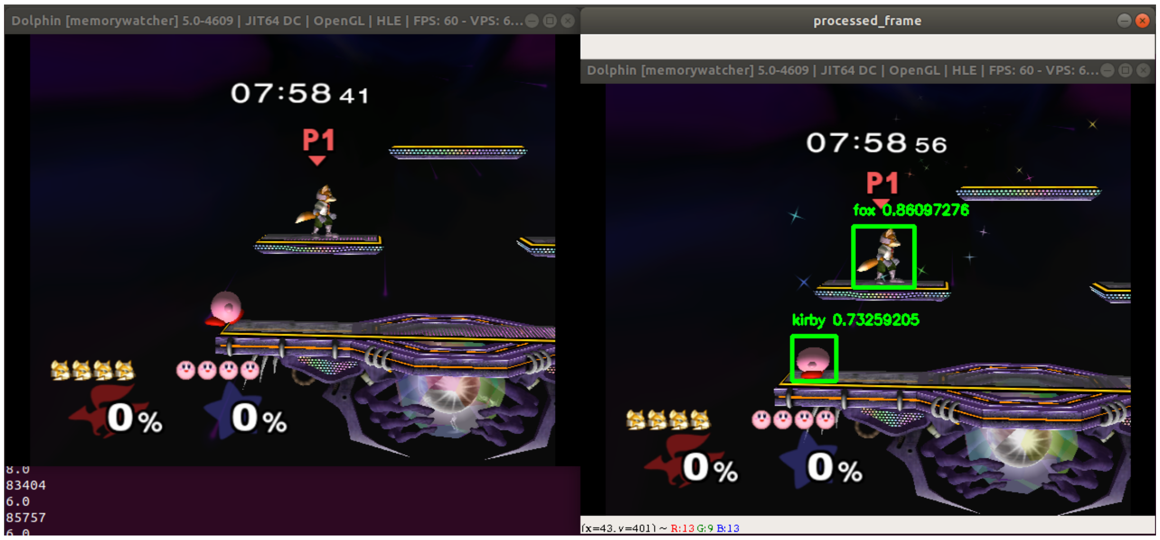
6.1 Frame Capture System

Since the bot was to function in real-time, the first step was to implement a real-time frame capture system. We used a Python library called MSS to capture frames [5] and defined a window size of 639x507 pixels (width by height) as it encompassed the emulated game window. The frames were captured in real-time at approximately 180-200 fps. However, the capture rate drops significantly to 24-26 fps when the captured frames are also displayed in a side window. This is an acceptable rate for a real-time bot, however, as we have computational limitations, namely a low-performance GPU, the character tracking is performed remotely over a server.

6.2 Real-Time Character Tracking

This is the fundamental step of the SSBM bot as it is what allows it to act according to specific responses from the enemy character. As mentioned before, this tracking needs to be performed over a server and therefore we opted to set up a TCP connection with a client-server architecture. The client has an Intel i7-5500U 2.4 GHz processor with an Intel HD Graphics 5500 GPU and the server has an Intel Xeon CPU E5-2695 v3 2.30GHz processor with a Nvidia GeForce GTX 980 GPU. The client GPU performs real-time detection at an extremely slow rate of 2 fps which is the reason we opted to perform detection remotely on a more powerful GPU.

We set up the client to send the captured frames 4096 bytes at a time. In addition, a header byte of '0', '1', or '2' indicating start, middle, or end, alongside a 4 byte packet length is sent prior to each of the 4096 bytes. On the server side, the detection model along with the post warm-up training weights are loaded prior to the connection being established. Once established, the server first receives the header byte and the 4 byte packet length. The packet length tells it how many bytes to receive. The reasoning behind sending a header byte and the packet length is to know when the client sends the final set of bytes (final packet) of the image. Typically the final packet is less than 4096 bytes and therefore, the server needs to know the length of the packet so it can save the completed image after receiving the appropriate amount of bytes and move on to detection. After saving the image, the server runs it though the detection model using a confidence threshold of 0.45 and an NMS threshold of 0.4 to filter the predicted boxes. The detected box coordinates are converted to a string and sent back to the client using the same method of sending 4096 bytes at a time. There is no need to send the image back to the client as a copy of the image was saved on the client side prior to sending bytes across the connection. The return string is decoded, the boxes are drawn on the saved image, and the image is shown in a side window (Figure 6.1). The side window displays detections at a rate of 6-10 fps depending on network strength.


Dolphin [memorywatcher] 5.0-4609 | JIT64 DC | OpenGL | HLE | FPS: 60 - VPS: 6...
processed_frame
07:58 41
07:58 56
P1
fox 0.86097276
kirby 0.73259205
0%
0%

```
8.0
83404
6.0
85757
6.0
```

(x=43, y=401) ~ R:13 G:9 B:13

Figure 6.1: A visual of our implementation for the real-time character tracking system during gameplay. SSBM is running on the left and the side window of detections is displayed on the right.

6.3 Bot Programming

The bot was set up as a two-thread system where the first thread controls the character within the game, and the second runs the frame capture and tracks the characters. So while the frame display may run at approximately 6-10 fps, the game is not hindered by this process and will always run at the full 60 fps. We do note however that if the side window containing the detection visualizations is not shown, the bot operates between a consistent 12 - 14 fps. We decided to have the enemy player control Ness while the bot controls Kirby. The stage selection is a hard-coded process which is controlled by the bot.

The algorithm pseudocode for the bot, dubbed *KirbyBot*, is outlined as follows:

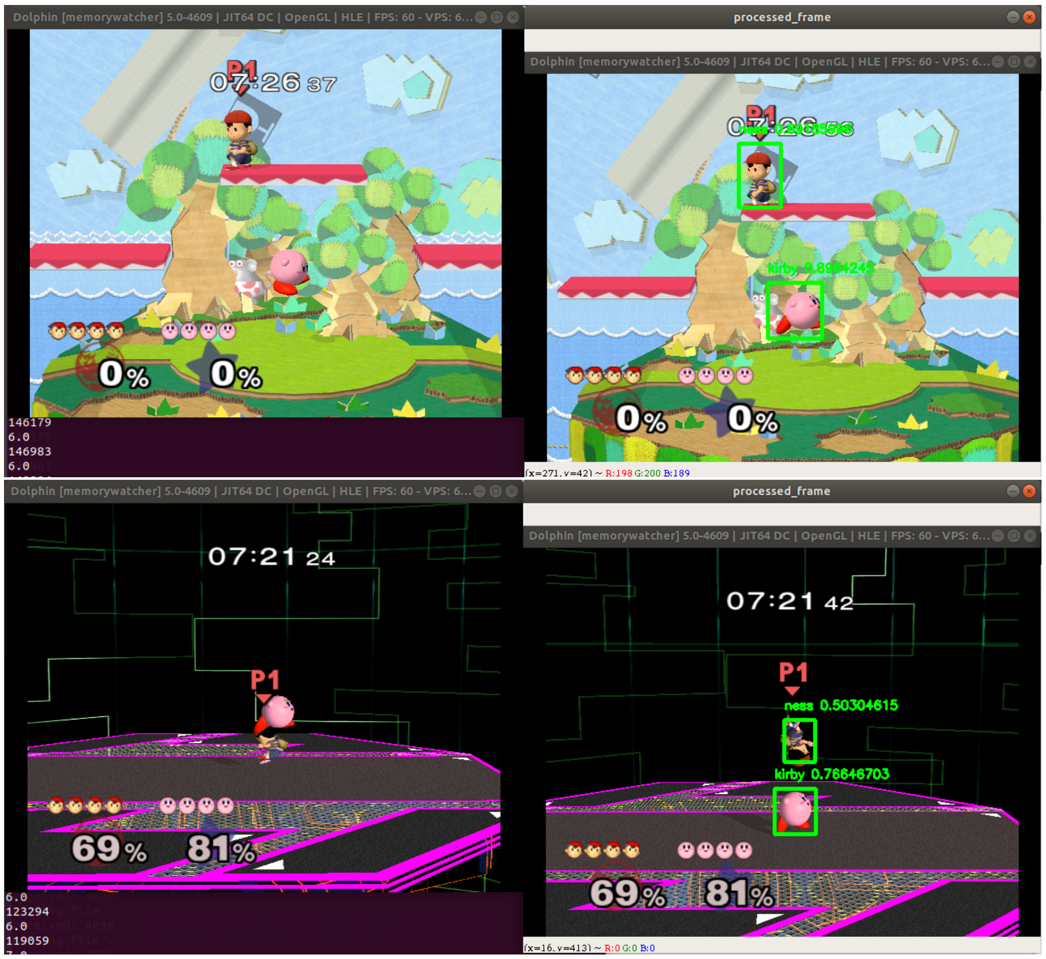
```
while SSBM is running do
    if both Ness and Kirby boxes exist then
        if Ness xmax + 15 pixels < Kirby xmin then
            Kirby moves left;
        else if Kirby xmax + 15 pixels < Ness xmin then
            Kirby moves right;
        else if Ness ymax + 15 pixels < Kirby ymin then
            Kirby jumps;
        else if Kirby ymax + 15 pixels < Ness ymin then
            Kirby comes down;
        else
            Kirby attacks;
    else
        Kirby doesn't move;
end
```

Algorithm 1: KirbyBot Pseudocode

The intuitive understanding of our algorithm is that KirbyBot simply follows Ness across the fighting stage. We decided to leave an extra 15 pixel distance so that when KirbyBot moves, it avoids running past Ness in a majority of cases. The final else statement handles the case where the bounding boxes of both characters overlap. In this case, KirbyBot launches a powerful attack which displaces Ness. KirbyBot has no concept of recovery, offensive or defensive maneuvers, or even independent movement. It relies purely on the location of the enemy Ness for instructions. This was our goal as a proof of concept - to design an adaptive AI that responds to enemy movements by tracking them in real time.

6.4 Experiment - Close Combat Interaction Rate

We were interested in collecting information on how often KirbyBot was fighting in close combat where the two characters were within 15 pixels of each other. To do so, we compared the number of frames that KirbyBot used the powerful attack with regard to the number of frames that both characters were detected. This experiment was done on 2 stages - Yoshi's Story and Final Destination, with data collected for approximately 1 minute on each. Figure 6.2 shows the two stages and Table 6.1 depicts the results collected from this experiment.


Dolphin [memorywatcher] 5.0-4609 | JIT64 DC | OpenGL | HLE | FPS: 60 - VPS: 6...
processed_frame
P1
07:26 37
0%
0%
kirby 0.8884245
07:21 24
07:21 42
ness 0.50304615
kirby 0.76646703
69%
81%
(x=271, y=42) ~ R:198 G:200 B:189
(x=16, y=413) ~ R:0 G:0 B:0

```
146179
6.0
146983
6.0
```

```
6.0
123294
6.0
119059
```

Figure 6.2: Close Combat Experiment conducted on Yoshi's Story (Top) and Final Destination (Bottom)

Table 6.1: Table of results for KirbyBot's performance over 2 minutes of Gameplay across 2 Stages (approximately 1 minute each).

Stage	Close Combat Frames	Detected Frames (both chars)	Total Frames
Final Destination	44	163	412
Yoshi's Story	36	256	425

There are two interesting conclusions that can be made from this data. First, it seems that under 40% of the frames were detected as containing both characters for Final Destination. This was expected as we allowed KirbyBot to knock Ness off of the fighting stage. SSBM contains a damage indicator which is directly proportional to the amount of knock-back a character receives when hit by an attack. After a certain threshold of damage is built up, when KirbyBot launches a powerful attack, most frames actually don't contain Ness as he usually is too high for the camera to capture. This is the reasoning for the low detection rate of both characters. A second conclusion we can draw from this data is that the two characters were within 15 pixels of each other approximately 27% of the time on Final Destination, as opposed to the 14% on Yoshi's Story. This is logical as Yoshi's Story contains several aerial platforms that we used to dodge KirbyBot. Final Destination on the other hand is a flat, one-level stage so it was harder to avoid KirbyBot. This table gives us insights into player habits. One could conclude without much trouble that the enemy Ness focused on avoiding KirbyBot as much as possible as the close combat rate is low for both stages. In addition, one could also conclude that the enemy Ness was more successful on Yoshi's Story as there is a lower close combat rate which indicates the use of the platforms. Logically, this information is subjective and dependent entirely on the play-style of the enemy Ness and can easily vary between players across each stage. However, this experiment was performed to demonstrate that data collection on player habits is also a feasible option using KirbyBot.

Chapter 7

FUTURE WORK

7.1 Improvements to SmashNet

A logical place to start making improvements is the SmashNet model. As it only yields a 62.29% accuracy on Caltech101 and a 68.25% accuracy on the annotated SSBM data, it leaves much room for improvement. We propose swapping the last average pooling and flatten operations with a single global average pooling operation. This pooling operation is performed using a field size of the entire spatial dimensions of the input which results in the following output:

$$7\text{x}7\text{x}1024 \longrightarrow \text{[Global average pooling]} \longrightarrow 1\text{x}1\text{x}1024$$

The operation produces a 1x1 output feature map for each respective input feature map by taking an average across all the values. This idea was adopted from Lin et al.'s paper "Network In Network" where they propose the use of global average pooling rather than multiple FC layers at the end of the CNN architecture to prevent over fitting during classification [23]. The 1024 channel output feeds into a final FC layer which uses the Softmax activation function to output a vector of predicted classes for the input. This improvement to SmashNet is encouraged over a standard average pooling $\longrightarrow$ flatten layers as it produced a 4% increased classification accuracy of approximately 66.8% on the Caltech101 dataset. With minimal overfitting, it reaches accuracies over 70%.

Other possible improvements to SmashNet include the use of 5x5 convolutions with increased stride values to account for larger feature extraction. Under the assumption that the 20(+2) million parameter requirement can be removed, the model's depth can be increased via extra convolutional layers for more feature extraction and better overall accuracy.

7.2 Improvements to datasets

A method to indirectly improve the accuracy of the model is through improving the dataset. SmashNet could be trained on a larger, more diverse dataset such as ImageNet for classification weights. These weights would likely extract better features during object detection. Additionally, the SSBM dataset is also a possible area of improvement. Our current SSBM dataset is only comprised of 2296 annotated images. A larger dataset would offer significantly more features for training, thereby improving the overall detection accuracy.

7.3 Improvements to KirbyBot

The major issue with KirbyBot is the 6-10 fps rate. This is mainly a result of the real-time character tracking system as our client was incapable of running the detection model at a rate greater than 2 fps. As a result, the captured frames are transferred to a remote machine that is capable of running the detection model faster which then sends the results back to the client. The overhead and latency from this back and forth transfer of data causes the bot to operate at a lower fps rate. The simple improvement to increase the fps rate is to run both the bot and detection model on the same machine. Unfortunately, this was not an option for us as the remote machine was incapable of running the emulator.

7.4 Other Interesting Future Work

Recent research suggests using real-time video detection to gather information about when two objects are a certain distance from each other [7]. It expands on this concept by suggesting a method to collate this information using SQL databases and using it for practical purposes such as detecting the position of a car relative to the stop line when the traffic light signals red [7]. Our experiment performed on KirbyBot in section 6.4 utilizes this general concept to collect practical information on player habits. We however feel that this system, if trained on more data with regard to moves and items, can be expanded significantly to collect information on aerial interaction rates, affinity to staying above ground, location preferences with respect to stage, move preferences, character preferences, item drop rates, among many more. We feel that this information can be used both casually and competitively as a way of improving the gameplay experience.

Additionally, an interesting area of research is to improve efficiency on training time for our model while also improving the weights obtained. Our current system back-propagates across the whole detection model during training, even with the pre-trained classification weights. This is not necessary as typically high-level features learned are similar across all objects and should not need to be retrained. Determining the optimal layer from which to start training the detection model is the ideal goal.

Chapter 8

CONCLUSION

In this research we developed a real-time character tracking system for the popular game SSBM utilizing a custom CNN model - SmashNet. We integrated classification pre-training and object detection training as parts of a 3-stage pipeline to develop this system. The model is completely trained from scratch and is capable of tracking 4 characters across multiple locations producing an overall detection accuracy of 68.25%.

This system led to the design of KirbyBot, a hard-coded proof of concept bot capable of playing SSBM at a basic level by following and attacking the real-time tracked enemy character. The bot is a black-box implementation as it inherently knows nothing about the game and relies entirely on the tracked locations of both Kirby and the enemy Ness to perform its functions. KirbyBot performs at an adequate 6-10 fps and exists as a demonstration of a potential use for a real-time tracking system in games.

Through this pursuit, we addressed the practical uses of real-time object detection research in games. Adaptive game AI and data collection on player habits are two such uses as demonstrated by KirbyBot, a simple bot that is far from complete and can be enhanced in multiple ways, especially through white-box functionality. We strongly believe that object detection research using CNNs opens new avenues of improving the casual and competitive gaming scene, and that exposure to these concepts will benefit the game industry as a whole.

BIBLIOGRAPHY

[1] CS231n Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io/convolutional-networks/ and http://cs231n.github.io/neural-networks-1/ and http://cs231n.github.io/neural-networks-2/ and http://cs231n.github.io/neural-networks-1/#actfun.

[2] Keras: The Python Deep Learning library. https://keras.io/.

[3] LabelImg. https://github.com/tzutalin/labelImg.

[4] Libmelee. https://github.com/altf4/libmelee.

[5] mss. http://python-mss.readthedocs.io/examples.html.

[6] H. N. Anh. Basic Yolo Keras. https://github.com/experiencor/basic-yolo-keras.

[7] P. Bailis et al. Data Infrastructure for the DAWN of Widespread ML. https://www.dropbox.com/s/5tdajv9m83t8afa/norcaldb-pb.pdf?dl=0, 2018.

[8] X. Cao, C. Wu, P. Yan, and X. Li. Linear SVM classification using boosting HOG features for vehicle detection in low-altitude airborne videos. In *Image Processing (ICIP), 2011 18th IEEE International Conference on*, pages 2421–2424. IEEE, 2011.

[9] Z. Chen and D. Yi. The Game Imitation: Deep Supervised Convolutional Networks for Quick Video Game AI. *arXiv preprint arXiv:1702.05663*, 2017.

[10] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[11] L. Fei-Fei, R. Fergus, and P. Perona. Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. *Computer vision and Image understanding*, 106(1):59–70, 2007.

[12] M. Garon. Mean Average Precision. `https://github.com/MathGaron/mean_average_precision`.

[13] R. Girshick. Fast r-cnn. *arXiv preprint arXiv:1504.08083*, 2015.

[14] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[15] G. Hinton. The Momentum Method. `https://www.coursera.org/learn/neural-networks/lecture/Oya9a/the-momentum-method`.

[16] G. Hinton, N. Srivastava, and K. Swersky. rmsprop: A mini-batch version of rprop. `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`, 2016.

[17] A. Jung. imgaug. `https://github.com/aleju/imgaug`.

[18] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski. Vizdoom: A doom-based ai research platform for visual reinforcement learning. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pages 1–8. IEEE, 2016.

[19] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[20] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[21] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng. Unsupervised learning of hierarchical representations with convolutional deep belief networks. *Communications of the ACM*, 54(10):95–103, 2011.

[22] F.-F. Li, A. Karpathy, and J. Johnson. Lecture 8: Spatial Localization and Detection. `http://cs231n.stanford.edu/slides/2016/winter1516_lecture8.pdf`, 2016.

[23] M. Lin, Q. Chen, and S. Yan. Network In Network. *CoRR*, abs/1312.4400, 2013.

[24] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: Common Objects in Context. *CoRR*, abs/1405.0312, 2014.

[25] C. C. T. Mendes, V. Frémont, and D. F. Wolf. Exploiting fully convolutional neural networks for fast road detection. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 3174–3179. IEEE, 2016.

[26] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[27] A. Y. Ng. Deep Learning Specialization. `https://www.coursera.org/specializations/deep-learning`, 2017.

[28] I. Nintendo/Hal Laboratory. Super smash bros. melee. [Console CD], 2001.

[29] A. H. Olsen. The Evolution of eSports: An Analysis of its origin and a look at its prospective future growth as enhanced by Information Technology Management tools. *CoRR*, abs/1509.08795, 2015.

[30] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *CoRR*, abs/1506.02640, 2015.

[31] J. Redmon and A. Farhadi. YOLO9000: Better, Faster, Stronger. *CoRR*, abs/1612.08242, 2016.

[32] S. Ren, K. He, R. B. Girshick, and J. Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *CoRR*, abs/1506.01497, 2015.

[33] A. Rosebrock. Intersection over Union (IoU) for object detection. `https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/`. Accessed: 2018-04-15.

[34] S. Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[35] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[36] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[37] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[38] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going Deeper with Convolutions. *CoRR*, abs/1409.4842, 2014.

[39] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the Inception Architecture for Computer Vision. *CoRR*, abs/1512.00567, 2015.

[40] M. Thoma. Analysis and Optimization of Convolutional Neural Network Architectures. Masters‘s thesis, Karlsruhe Institute of Technology, Karlsruhe, Germany, June 2017.

[41] Y. Uchida. convnet-drawer. `https://github.com/yu4u/convnet-drawer`.

[42] B. Wu, F. Iandola, P. H. Jin, and K. Keutzer. Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. *arXiv preprint arXiv:1612.01051*, 2016.

[43] J. Yosinski, J. Clune, A. M. Nguyen, T. J. Fuchs, and H. Lipson. Understanding Neural Networks Through Deep Visualization. *CoRR*, abs/1506.06579, 2015.

APPENDICES

Appendix A

RESULTS

Table A.1: Full results of SmashNet classification on validation data (1302 images)

Class	Recall	Recall %	Precision	Precision %	Error	Error %
strawberry	0/8	0.0	0/5	0.0	8/8	1.0
cougar_body	0/6	0.0	0/1	0.0	6/6	1.0
emu	0/5	0.0	0/7	0.0	5/5	1.0
platypus	0/3	0.0	0/0	N/A	3/3	1.0
wrench	0/5	0.0	0/1	0.0	5/5	1.0
flamingo_head	0/3	0.0	0/6	0.0	3/3	1.0
mayfly	0/8	0.0	0/0	N/A	8/8	1.0
wild_cat	0/5	0.0	0/5	0.0	5/5	1.0
gerenuk	0/2	0.0	0/4	0.0	2/2	1.0
binocular	0/3	0.0	0/1	0.0	3/3	1.0
ant	0/6	0.0	0/2	0.0	6/6	1.0
sea_horse	0/13	0.0	0/0	N/A	13/13	1.0
snoopy	0/3	0.0	0/1	0.0	3/3	1.0
ibis	0/13	0.0	0/0	N/A	13/13	1.0
bass	1/13	0.07692	1/5	0.2	12/13	0.9231
helicopter	1/12	0.08333	1/2	0.5	11/12	0.9167
camera	1/11	0.09091	1/1	1.0	10/11	0.9091
hedgehog	1/9	0.1111	1/6	0.1667	8/9	0.8889

Class	Recall	Recall %	Precision	Precision %	Error	Error %
elephant	1/8	0.125	1/3	0.3333	7/8	0.875
crocodile	1/7	0.1429	1/3	0.3333	6/7	0.8571
ferry	2/14	0.1429	2/5	0.4	12/14	0.8571
crayfish	1/6	0.1667	1/4	0.25	5/6	0.8333
rhino	2/12	0.1667	2/7	0.2857	10/12	0.8333
garfield	1/6	0.1667	1/2	0.5	5/6	0.8333
hawksbill	2/12	0.1667	2/6	0.3333	10/12	0.8333
llama	2/10	0.2	2/2	1.0	8/10	0.8
anchor	1/5	0.2	1/6	0.1667	4/5	0.8
dolphin	2/10	0.2	2/2	1.0	8/10	0.8
chair	2/9	0.2222	2/8	0.25	7/9	0.7778
umbrella	2/8	0.25	2/3	0.6667	6/8	0.75
water_lilly	2/8	0.25	2/4	0.5	6/8	0.75
dalmatian	3/12	0.25	3/3	1.0	9/12	0.75
beaver	1/4	0.25	1/2	0.5	3/4	0.75
scorpion	3/12	0.25	3/10	0.3	9/12	0.75
metronome	1/4	0.25	1/2	0.5	3/4	0.75
kangaroo	2/8	0.25	2/29	0.06897	6/8	0.75
cup	4/13	0.3077	4/11	0.3636	9/13	0.6923
ceiling_fan	2/6	0.3333	2/7	0.2857	4/6	0.6667
pigeon	2/6	0.3333	2/3	0.6667	4/6	0.6667
electric_guitar	4/12	0.3333	4/8	0.5	8/12	0.6667
stapler	1/3	0.3333	1/4	0.25	2/3	0.6667
stop_sign	4/11	0.3636	4/5	0.8	7/11	0.6364
sunflower	5/13	0.3846	5/10	0.5	8/13	0.6154
cellphone	5/13	0.3846	5/5	1.0	8/13	0.6154

Class	Recall	Recall %	Precision	Precision %	Error	Error %
crocodile_head	2/5	0.4	2/38	0.05263	3/5	0.6
pyramid	2/5	0.4	2/4	0.5	3/5	0.6
barrel	4/10	0.4	4/9	0.4444	6/10	0.6
saxophone	4/10	0.4	4/5	0.8	6/10	0.6
tick	4/10	0.4	4/4	1.0	6/10	0.6
nautilus	3/7	0.4286	3/6	0.5	4/7	0.5714
buddha	6/14	0.4286	6/12	0.5	8/14	0.5714
ketch	6/14	0.4286	6/9	0.6667	8/14	0.5714
flamingo	5/11	0.4545	5/9	0.5556	6/11	0.5455
euphonium	7/15	0.4667	7/14	0.5	8/15	0.5333
ewer	10/20	0.5	10/13	0.7692	10/20	0.5
lamp	5/10	0.5	5/6	0.8333	5/10	0.5
starfish	4/8	0.5	4/5	0.8	4/8	0.5
panda	2/4	0.5	2/2	1.0	2/4	0.5
inline_skate	4/7	0.5714	4/4	1.0	3/7	0.4286
cannon	4/7	0.5714	4/11	0.3636	3/7	0.4286
crab	4/7	0.5714	4/25	0.16	3/7	0.4286
wheelchair	3/5	0.6	3/16	0.1875	2/5	0.4
soccer_ball	8/13	0.6154	8/10	0.8	5/13	0.3846
lotus	2/3	0.6667	2/5	0.4	1/3	0.3333
lobster	2/3	0.6667	2/20	0.1	1/3	0.3333
rooster	8/12	0.6667	8/9	0.8889	4/12	0.3333
bonsai	10/15	0.6667	10/20	0.5	5/15	0.3333
schooner	8/12	0.6667	8/12	0.6667	4/12	0.3333
brontosaurus	6/9	0.6667	6/16	0.375	3/9	0.3333
okapi	4/6	0.6667	4/9	0.4444	2/6	0.3333

Class	Recall	Recall %	Precision	Precision %	Error	Error %
brain	13/19	0.6842	13/18	0.7222	6/19	0.3158
joshua_tree	7/10	0.7	7/15	0.4667	3/10	0.3
accordion	7/10	0.7	7/16	0.4375	3/10	0.3
gramophone	7/10	0.7	7/7	1.0	3/10	0.3
laptop	7/10	0.7	7/16	0.4375	3/10	0.3
chandelier	13/18	0.7222	13/21	0.619	5/18	0.2778
stegosaurus	3/4	0.75	3/8	0.375	1/4	0.25
car_side	9/12	0.75	9/14	0.6429	3/12	0.25
butterfly	9/12	0.75	9/26	0.3462	3/12	0.25
mandolin	6/8	0.75	6/14	0.4286	2/8	0.25
grand_piano	16/21	0.7619	16/26	0.6154	5/21	0.2381
dragonfly	7/9	0.7778	7/10	0.7	2/9	0.2222
windsor_chair	7/9	0.7778	7/8	0.875	2/9	0.2222
octopus	4/5	0.8	4/5	0.8	1/5	0.2
revolver	8/10	0.8	8/12	0.6667	2/10	0.2
cougar_face	12/15	0.8	12/28	0.4286	3/15	0.2
pizza	8/10	0.8	8/14	0.5714	2/10	0.2
watch	27/33	0.8182	27/43	0.6279	6/33	0.1818
trilobite	12/14	0.8571	12/17	0.7059	2/14	0.1429
scissors	6/7	0.8571	6/9	0.6667	1/7	0.1429
headphone	6/7	0.8571	6/8	0.75	1/7	0.1429
dollar_bill	9/10	0.9	9/10	0.9	1/10	0.1
yin_yang	10/11	0.9091	10/10	1.0	1/11	0.09091
minaret	12/13	0.9231	12/13	0.9231	1/13	0.07692
airplanes	114/123	0.9268	114/121	0.9421	9/123	0.07317
Leopards	27/29	0.931	27/43	0.6279	2/29	0.06897

Class	Recall	Recall %	Precision	Precision %	Error	Error %
menorah	14/15	0.9333	14/15	0.9333	1/15	0.06667
Faces_easy	52/54	0.963	52/55	0.9455	2/54	0.03704
Faces	73/75	0.9733	73/77	0.9481	2/75	0.02667
pagoda	5/5	1.0	5/5	1.0	0/5	0.0
Motorbikes	112/112	1.0	112/124	0.9032	0/112	0.0

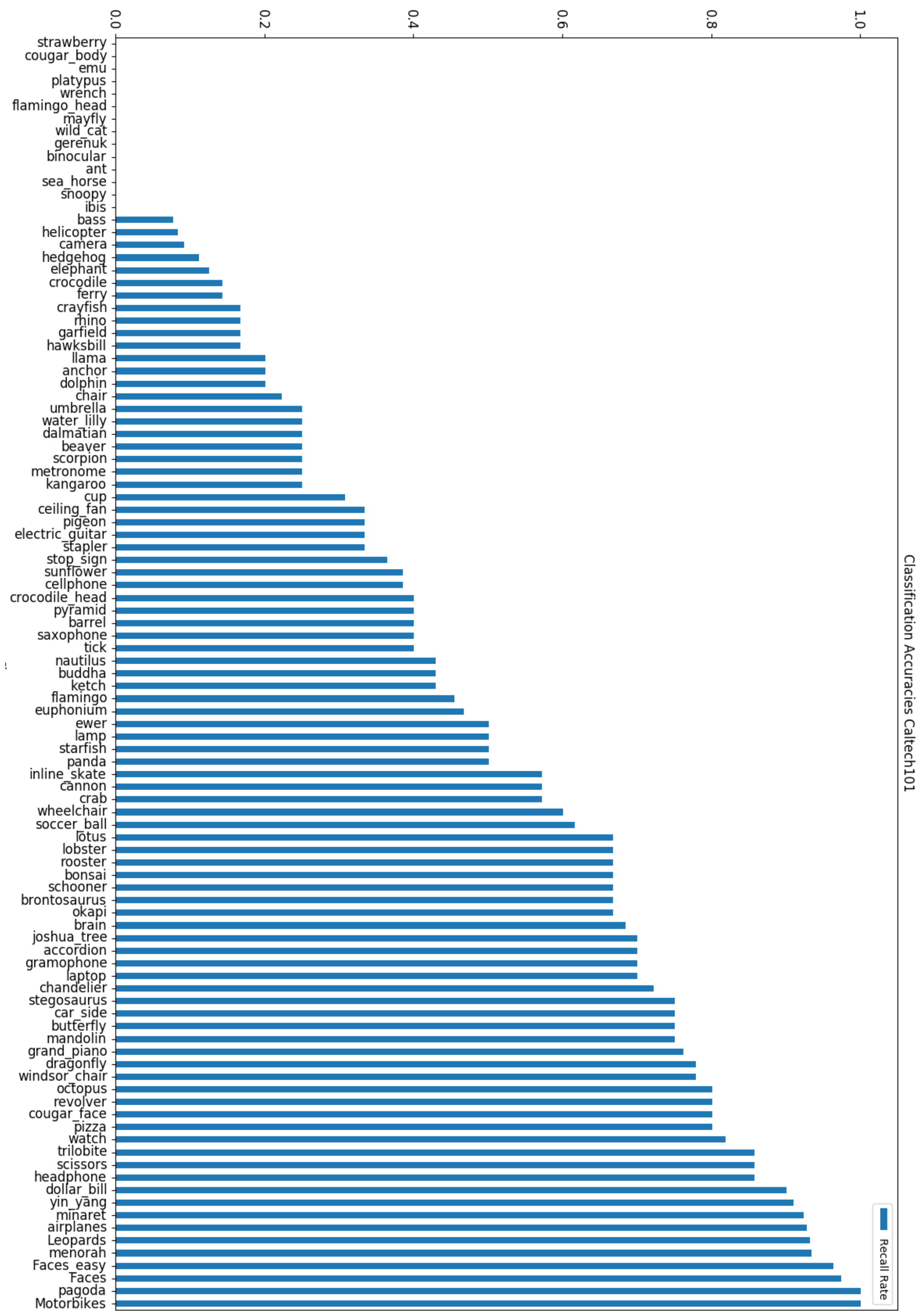


Figure A.1: Caltech101 Classification Accuracies (Classes vs Recall %)