3-2021

Bayesian Augmentation of CNN-LSTM for Video Classification with Uncertainty Measures

Emmie K. Swize

Bayesian Augmentation of Convolutional Neural Network - Long Short Term Memory for Video Classification with Uncertainty Measures

THESIS

Emmie K. Swize, 2d Lt, USAF

AFIT-ENS-MS-21-M-186

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENS-MS-21-M-186

BAYESIAN AUGMENTATION OF CONVOLUTIONAL NEURAL NETWORK - LONG SHORT TERM MEMORY FOR VIDEO CLASSIFICATION WITH UNCERTAINTY MEASURES

THESIS

Presented to the Faculty
Department of Operational Sciences
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Master of Science of Operations Research

Emmie K. Swize, BS
2d Lt, USAF

March 25, 2021

AFIT-ENS-MS-21-M-186

BAYESIAN AUGMENTATION OF CONVOLUTIONAL NEURAL NETWORK - LONG SHORT TERM MEMORY FOR VIDEO CLASSIFICATION WITH UNCERTAINTY MEASURES

THESIS

Emmie K. Swize, BS
2d Lt, USAF

Committee Membership:

Dr. Lance E. Champagne
Advisor

Dr. Bruce A. Cox
Co-Advisor

Dr. Trevor Bihl
Member

Capt Phillip R. Jenkins, PhD
Reader

AFIT-ENS-MS-21-M-186

Abstract

The success of Department of Defense missions relies heavily on intelligence, surveillance, and reconnaissance capabilities, which supply information about the activities and resources of an enemy or adversary. To secure this information, satellites and unmanned aircraft systems collect video data to be classified by either humans or machine learning networks. Traditional automated video classification methods lack measures of uncertainty, meaning the network is unable to identify those cases in which its predictions are made with significant uncertainty. This leads to misclassification, as the traditional network classifies each observation with the same amount of certainty, no matter what the observation is. Bayesian neural networks offer a remedy to this issue by leveraging Bayesian inference to construct uncertainty measures for each prediction. Because exact Bayesian inference is typically intractable due to the large number of parameters in a neural network, Bayesian inference is approximated by utilizing dropout in a convolutional neural network. This research compared a traditional video classification neural network to its Bayesian equivalent based on performance and capabilities. The Bayesian network achieves higher accuracy and is able to produce uncertainty measures for each classification.

For my mom and dad,

Who believed in me even when success felt 'inconceivable.'

Acknowledgements

I would first like to thank God, without whom nothing is possible.

I would like to thank my advisor, Dr. Lance Champagne, for his trust and guidance throughout the past year.

And finally, thank you to my husband. Without your patience and computer, I wouldn't have been able to complete this research.

Emmie K. Swize

Table of Contents

Page

List of Figures

List of Tables

BAYESIAN AUGMENTATION OF CONVOLUTIONAL NEURAL NETWORK - LONG SHORT TERM MEMORY FOR VIDEO CLASSIFICATION WITH UNCERTAINTY MEASURES

I. Introduction

Video classification allows for intelligence, surveillance, and reconnaissance (ISR) platforms to detect and identify objects and activities in surveillance videos autonomously. Image classification is one of the many functions of neural networks. Convolutional neural networks (CNN) in particular are often used for image classification. Recurrent neural networks (RNN) are well equipped to handle time series data, such as a video broken into a series of images. These two capabilities combined allow for the classification of videos. The remainder of this chapter includes the problem statement, background, motivation for this research, and a brief outline of the remainder of this thesis.

1.1 Problem Statement

The objective of this research is to develop a blend of neural networks for video classification for the purpose of ISR. The blended network consists of a CNN at the front end of a RNN, as well as an inserted Bayesian neural network (BNN) to update the priors of the CNN.

1.2 Background and Motivation

ISR missions involve using artificial intelligence (AI) and machine learning (ML) algorithms to identify objects and build a sensor-aware operating picture. However, the current technology is not flexible enough in nature to handle novelty data. For example, currently available classification algorithms are largely static in nature once trained and context from recently observed data is not considered in making decisions, meaning such algorithms are unequipped to handle uncertain and unexpected situations. The remedy for this is to create classification algorithms that are aware of their own uncertainty and therefore able to identify unexpected data. (Bihl and Talbert, 2020)

1.3 Organization of the Thesis

Chapter 2 reviews past and current research into BNNs as well as video and image classification using neural networks. Chapter 3 explains the methodology used to develop the model as well as its evaluation criteria. Chapter 4 presents the results of the analysis. Chapter 5 covers the conclusions and presents possible areas for future research.

II. Literature Review

This chapter discusses past research into the use of neural networks for video classification. It is partitioned by sections as appropriate to provide structure for discussion of material being reviewed.

2.1 Artificial Neural Networks

Artificial neural networks (ANNs) are machine learning models loosely based on the structure of the brain's biological network, in which biological neurons pass information through connections when triggered (Géron, 2019). ANNs are versatile machine learning tools that can handle large and complex tasks, including image recognition. Furthermore, constructed appropriately, ANNs are a provably optimal approach to learning patterns in data (Géron, 2019). The first ANN was designed by McCulloch and Pitts in 1943. In their paper "A logical calculus of the ideas immanent in nervous activity," McCulloch and Pitts present an artificial neuron that is activated by binary inputs to produce a binary output (Mcculloch and Pitts, 1943). By constructing a network of such artificial neurons, the authors show that ANNs can perform logical computations.

In 1957, Frank Rosenblatt designed the Perceptron, an ANN composed of a single layer of threshold logic units (TLU) (Géron, 2019). Like McCulloch and Pitts' artificial neuron, TLUs are neurons activated by the inputs passed to them. However, each TLU receives a numerical value as input and produces a numerical value as output. Additionally, as demonstrated in Figure 1, the connections surrounding TLUs are weighted, meaning each unit computes the weighted sum of the inputs passed to it according to the connection weights. The output of each TLU is the result of a step function applied to the weighted sum of the inputs. Figure 2 demonstrates the structure of a Perceptron with two inputs and three TLUs. Werbos (1974) proposed a new weight-training method called backpropagation, which lead to ANNs using differentiable activation functions, such as the sigmoid activation function.

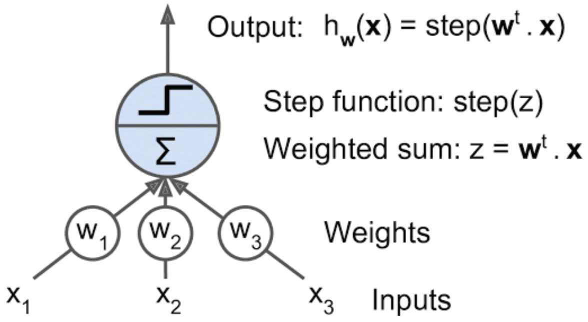
Output: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(\mathbf{w}^t \cdot \mathbf{x})$
Step function: step(z)
Weighted sum: $z = \mathbf{w}^t \cdot \mathbf{x}$
Σ
w_1
w_2
w_3
Weights
x_1
x_2
x_3
Inputs

Figure 1. Threshold Logic Unit (Géron, 2019))

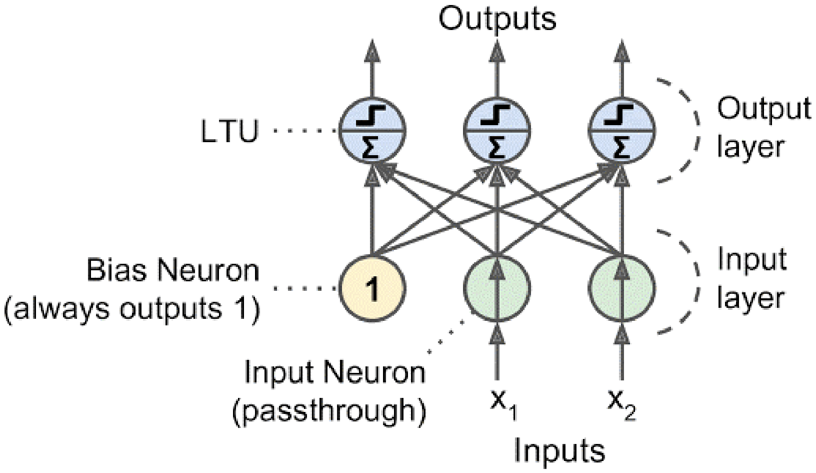

Figure 2. Perceptron (Géron, 2019)

The capabilities of such a model improve with depth, leading to a Multilayer Perceptron (MLP), which is an ANN composed of multiple layers of TLUs (Géron, 2019). The middle layers are referred to as the hidden layers and the last layer is referred to as the output layer. A Deep Neural Network (DNN) builds on the structure of the MLP, consisting of a deep stack of hidden layers, typically three or more.

2.2 Convolutional Neural Networks

Recently, DNNs have led to developmental milestones in the capabilities of deep learning, including the processing of images, video, and audio. These milestones allow for improved recognition and classification of objects and actions in both images and video. CNNs are a variant of DNNs that are particularly useful in processing and categorizing 2-dimensional visual data, such as images and handwriting (Géron, 2019). The earliest stage of the CNN, the Neocognitron, was proposed by Kunihiko Fukushima in 1980 (Fukushima, 1980). The Neocognitron contains simple cell operations for the feature extraction of an image and complex cell operations that pool the simple cell results to provide spatial invariance. This early stage CNN model inspired the LeNet-4 model in 1995, which developed into the LeNet-5 model in 1998 (Lecun et al., 1998). The LeNet-5 model is considered a milestone because it introduced convolutional layers and pooling layers, the backbones of the modern-day CNN. Convolutional layers apply a learned filter, called the kernel, to the input arrays to detect co-occurrences and spatial information in the input (Géron, 2019). In doing so, small "neighborhoods" of the image are examined, revealing each neuron's area of influence based on its location. This lends well to image and video classification, in which pixels closer together are typically more correlated than pixels farther apart. Pooling layers create a summarized version of the features identified by the preceding convolutional layer, reducing the dimensionality. With convolutional and pooling layers, CNNs assemble simple features into increasingly more complex features with each hidden layer (Géron, 2019).

Figure 3 demonstrates the layout of the LeNet-5 architecture. Each convolutional layer outputs one feature map for each filter, each of which emphasizes the image locations that activate the respective filter the most. By applying multiple filters to the inputs, a convolutional layer is able to extract multiple features at each location.

The sub-sampling layers represent the pooling layers, which reduce the sensitivity of the outputs to shifts and distortions in the image. This gives the CNN the powerful capability of recognizing a learned pattern in any location in the image, not just where the original pattern instance occurred (Lecun et al., 1998). As a final classifier, the LeNet-5 architecture includes an MLP at the end consisting of fully connected layers and an output layer.

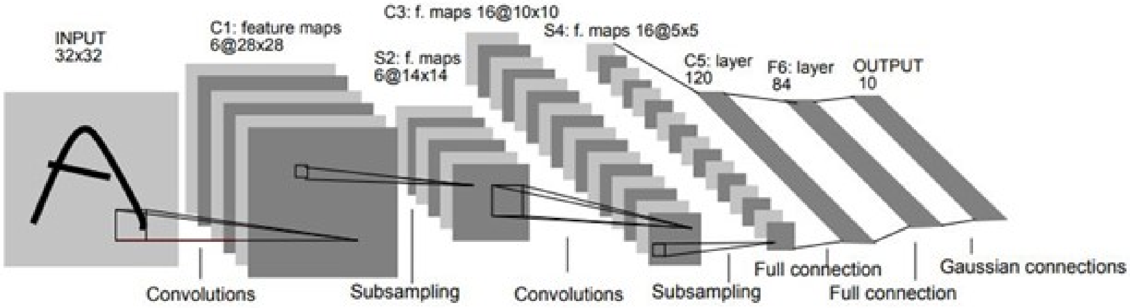

Figure 3. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical. (Lecun et al., 1998)

2.3 Recurrent Neural Networks

While CNNs are useful at processing and categorizing individual images, RNNs are another variant of DNNs that can process sequential data, such as a video broken into a series of images. RNNs possess a type of memory in the form of a hidden state, which passes previous output information as additional inputs to future time steps in the network (Géron, 2019). The Hopfield Network, proposed by John Hopfield in 1982, was the first DNN to incorporate a form of associative memory into the network (Gal, 2016). In 1985, David Rumelhart expanded on this early form of the RNN by incorporating Backpropagation Through Time (BPTT) (Gal, 2016). Backpropagation is a procedure that repeatedly updates a network's connection weights according to the error of the network's output. BPTT is the application of backpropagation to each time step of an RNN or RNN variant (Gal, 2016). Figure 4 demonstrates a layer of recurrent neurons unrolled through time. At each step, the layer receives the input x_i and the output of the previous time step y_{t-1} as inputs.

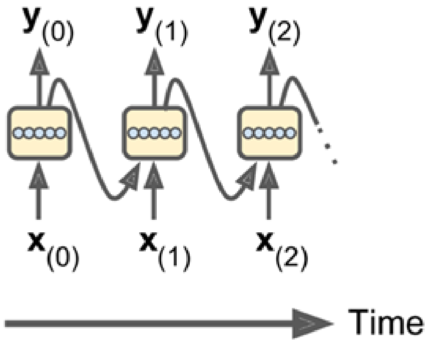

Figure 4. RNN Layer Unrolled through Time (Géron, 2019)

Standard RNNs, although able to process sequential data, can suffer from short-term memory, meaning the network's memory is limited in the amount of past outputs it can represent clearly. In 1997, Hochreiter and Schmidhuber presented an evolved version of the standard RNN, the Long Short-Term Memory (LSTM) cell, as a solution (Hochreiter and Schmidhuber, 1997). LSTM cells outperform standard RNNs by converging more quickly and by detecting long-term dependencies in sequential data. Figure 5 shows that LSTMs possess not only a hidden state, but a cell state as well. The hidden state is similar to that of the standard RNN and represents the short-term information, while the cell state represents the long-term information. LSTM cells also utilize three gate controllers that are responsible for adding information to the stored memory or erasing information from the stored memory (Géron, 2019). These controllers are the forget gate ($f_{(t)}$), the input gate ($i_{(t)}$), and the output gate ($o_{(t)}$), as labeled in Figure 5. With the use of these gates, the LSTM cell is able to discern which content should be stored in its memory and which content it should forget.

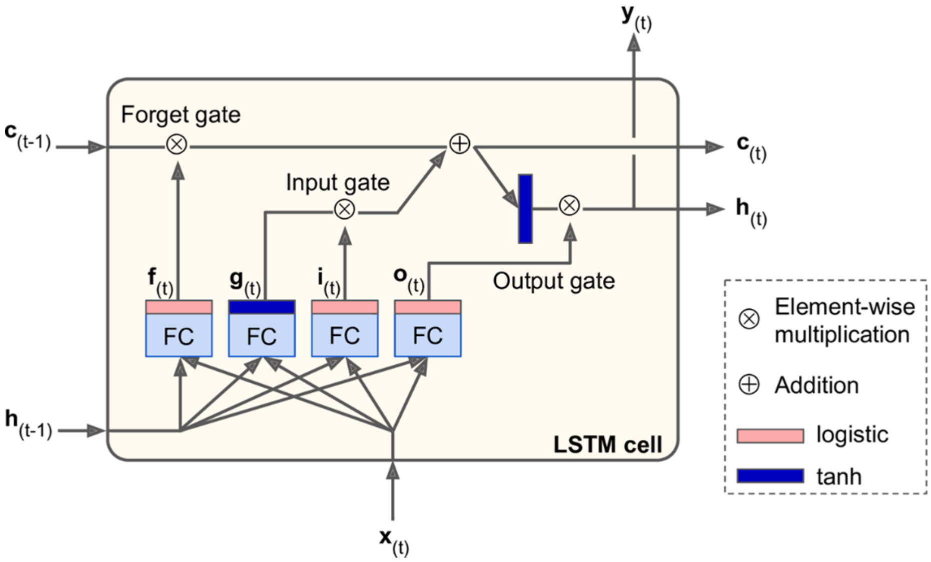

Figure 5. LSTM Cell (Géron, 2019)

2.4 Bayesian Neural Networks

Standard ANNs and DNNs learn point estimates for network weights and produce point estimates for predictions. These networks do not have a measure of certainty or confidence in the parameters and predictions, making the results potentially difficult to trust. In other words, standard DNNs output point estimate predictions, but no measure of respective uncertainty. Using Bayesian inference, a BNN incorporates a measure of uncertainty by learning parameters as distributions instead of point estimates (Gal, 2016). Bayesian inference is a type of statistical inference that uses Bayes' theorem, represented in Equation 1.

$$p(\Theta|D) = \frac{p(D|\Theta)p(\Theta)}{p(D)} \tag{1}$$

Here, $p(\theta)$ is the prior probability of model parameters θ before having seen the evidence, or data, D. $p(D|\Theta)$ is the likelihood of the occurrence of evidence D given the model parameters Θ. The theorem is used to update the inferred weight distributions as more information becomes available to the network. Any kind of network can become a BNN by treating the parameters in a Bayesian manner (Gal, 2016). Figure 6 presents a visual of the weight-learning difference between a standard CNN (left) and BNN (right).

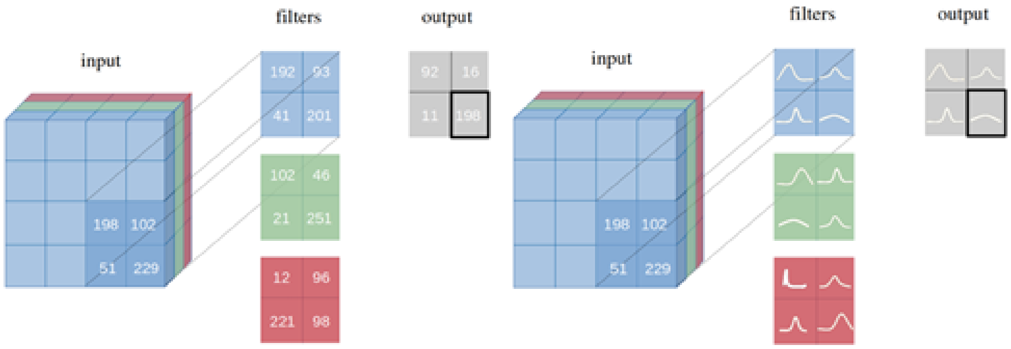

Figure 6. Input image with exemplary pixel values, filters, and corresponding output with point estimates (left) and probability distributions (right) over weights. (Shridhar et al., 2019)

Denker et al. (1987) proposed the first usage of a prior distribution for the weights of a network. By integrating over the weights, Denker et al. obtained a marginal probability for each outcome. Tishby et al. (1989). expanded upon this proposal and conceptualized the first BNN. Tishby et al. defined a prior distribution over the network weights and showed that with Bayes' Theorem, Bayesian inference can be performed to identify the optimal network architecture. Denker and LeCun (1990) built on this concept by transforming the network outputs into probability distributions. The concept of the BNN is similar to that of the Probabilistic neural network (PNN), which was proposed by Specht (1990). PNNs, another version of ANNs that are derived from BNNs, utilize an exponential activation function to approximate the Bayes optimal solution. Buntine and Weigend (1991) presented the concept of Bayesian back-propagation, a Bayesian inference method that combines the concepts of backpropagation and Bayes' Theorem. In 1993, Neal (2012) showed that Bayesian inference methods avoid overfitting and poor generalization using the first Markov Chain Monte Carlo (MCMC) sampling algorithm, which relies on Bayesian inference.

Neal (1995) expanded his contribution to the growth of BNNs the following year by establishing a link between BNNs and Gaussian processes, which are stochastic processes in which every finite linear combination of random variables is normally distributed. Neal showed that a single-layered network with infinitely many units and Gaussian priors for the weights is equivalent to a Gaussian process. This equivalence means that exact Bayesian inference can be conducted on infinite neural networks by evaluating the corresponding Gaussian process. Although the Gaussian process properties do not translate easily to finite neural networks, Bayesian inference can be approximated for finite neural networks that have Gaussian priors for the weights.

As pointed out by Shridhar et al., even for a network with few parameters, performing exact Bayesian inference to determine a network's posterior is a lengthy and difficult task (Shridhar et al., 2019). For this reason, Bayesian inference is often approximated using variational inference, a method that fits a Gaussian distribution as closely as possible to the true posterior distribution (Gal, 2016). This is done by minimizing the Kullback-Leibler (KL) divergence, a measure of how much information is lost in the approximation. However, because variational inference significantly increases the number of model parameters, it comes at a high computational cost.

2.5 Dropout as a Bayesian Approximation

A simple method of Bayesian inference approximation that does not sacrifice computational complexity is dropout. In 2016, Gal and Ghahramani demonstrated that applying dropout before every weighted layer in a network is mathematically equivalent to a Bayesian approximation of a Gaussian process (Gal and Ghahramani, 2016). Their work shows that the application of dropout minimizes the KL divergence between an approximate distribution and a Gaussian process posterior distribution.

Dropout, a popular regularization technique that prevents a model from overfitting training data, is a process that randomly omits neurons and their associated connections from the neural network according to a fixed probability p. In other words, at each step, each neuron will be retained in the network at that step with probability p. Figure 7, created by Srivastava et al., demonstrates the difference between standard neural network layers and neural network layers with dropout. This technique essentially samples a thinned version of the full network for each training case. For a neural network with n neurons, applying dropout to training amounts to a collection of 2^n possible networks (Srivastava et al., 2014). Additionally, dropout prevents individual neurons from relying on other specific neurons to supplement their contributions to the network (Hinton et al., 2012). This causes the contribution of each neuron to become more helpful regarding correct network predictions.

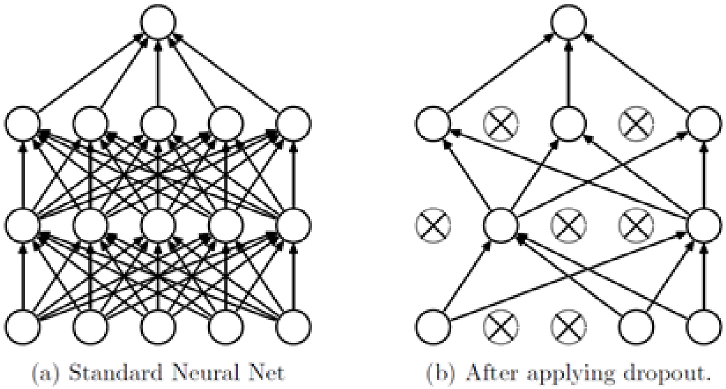

Figure 7. Dropout Neural Net Model. Left: A standard neural net with 2 hidden layers. Right: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped. (Srivastava et al., 2014)

A typical dropout procedure is only implemented during the training stage, causing predictions during the testing stage to be deterministic. Monte Carlo dropout (MCD), proposed by Gal and Ghahramani in 2016, applies dropout to both the training and testing stages of a network. In statistics, Monte Carlo refers to algorithms that utilize repeated random sampling to infer distributions for a numerical quantity. Implementing dropout during the testing stage means that the model output can be treated as a random sample generated from the posterior predictive distribution. The model uncertainty can therefore be estimated with the distribution of repeated predictions for an instance, constructing a distribution of probabilities for every class. With this distribution of multiple predictions, the average and the variation can reveal the networks uncertainty in its predictions. Gal and Ghahramani show that not only is this procedure simple in execution, but that it has no negative impact on model performance (Gal and Ghahramani, 2016).

III. Modeling and Methodology

This chapter presents the methodology employed to build, train, and test the desired blend of neural networks. First, the chapter details the contents, structure, and preparation of the data set used for training and testing. Following the data description, the model architecture and evaluation are covered.

3.1 Blend of Networks

This research explores the effects of utilizing a blend of the networks discussed above to gain synergistic effaaAZects. The CNN is responsible for classifying individual frames of a video, which is then passed to the RNN as a sequence of frames for classification of the video as a whole. The CNN parameters are treated in a Bayesian manner, making it a BCNN. The front-end network is a BCNN and the back-end network is an RNN. The goal of this network architecture is to classify videos and provide a measure of uncertainty in each video's frame predictions.

3.2 Data Description

The data set used in this research is the UCF101 Action Recognition Data Set (Soomro et al., 2012). The University of Central Florida (UCF) introduced this dataset in 2012 as the largest compilation of human action videos, comprising of 13,320 videos that span 101 classes in total. UCF101 offers a large amount of variety regarding actions, camera motion and viewpoint, object pose and scale, background appearance, and lighting conditions. The 101 action classes are divided into five types: Human-Object Interaction, Body-Motion Only, Human-Human Interaction, Playing Musical Instruments, and Sports.

Table 1 provides the summary statistics of the data set structure and Figures 8 and 9 provide visual representations of the data set. Figure 8 contains frames of an example video from each of the 101 action classes along with corresponding class and class type labels. Figure 9 offers a visual of the number of videos in each class, as well as the distribution of video duration.

Each class is also subsequently divided into 25 groups based on common features, such as background or viewpoint. For all classes, the 25 groups contain anywhere from 4 to 7 videos each. Although the videos of roughly half of the action categories also contain audio, this feature is not taken into consideration in this research. As preparation for training and testing, each video is partitioned into individual frames sized 224 x 224 pixels, with each pixel containing values for 3 color channels. The images are converted to three-dimensional arrays, sized 224 x 224 x 3. These arrays are normalized by dividing all values by 255 to have all pixel values range from 0 to 1.

Table 1. UCF101 Summary Statistics (Soomro et al., 2012)

Actions	101
Clips	13320
Groups per Action	25
Clips per Group	4-7
Mean Clip Length	7.21 sec
Total Duration	1600 mins
Min Clip Length	1.06 sec
Max Clip Length	71.04 sec
Frame Rate	25 fps
Resolution	320x240
Audio	Yes (51 actions)

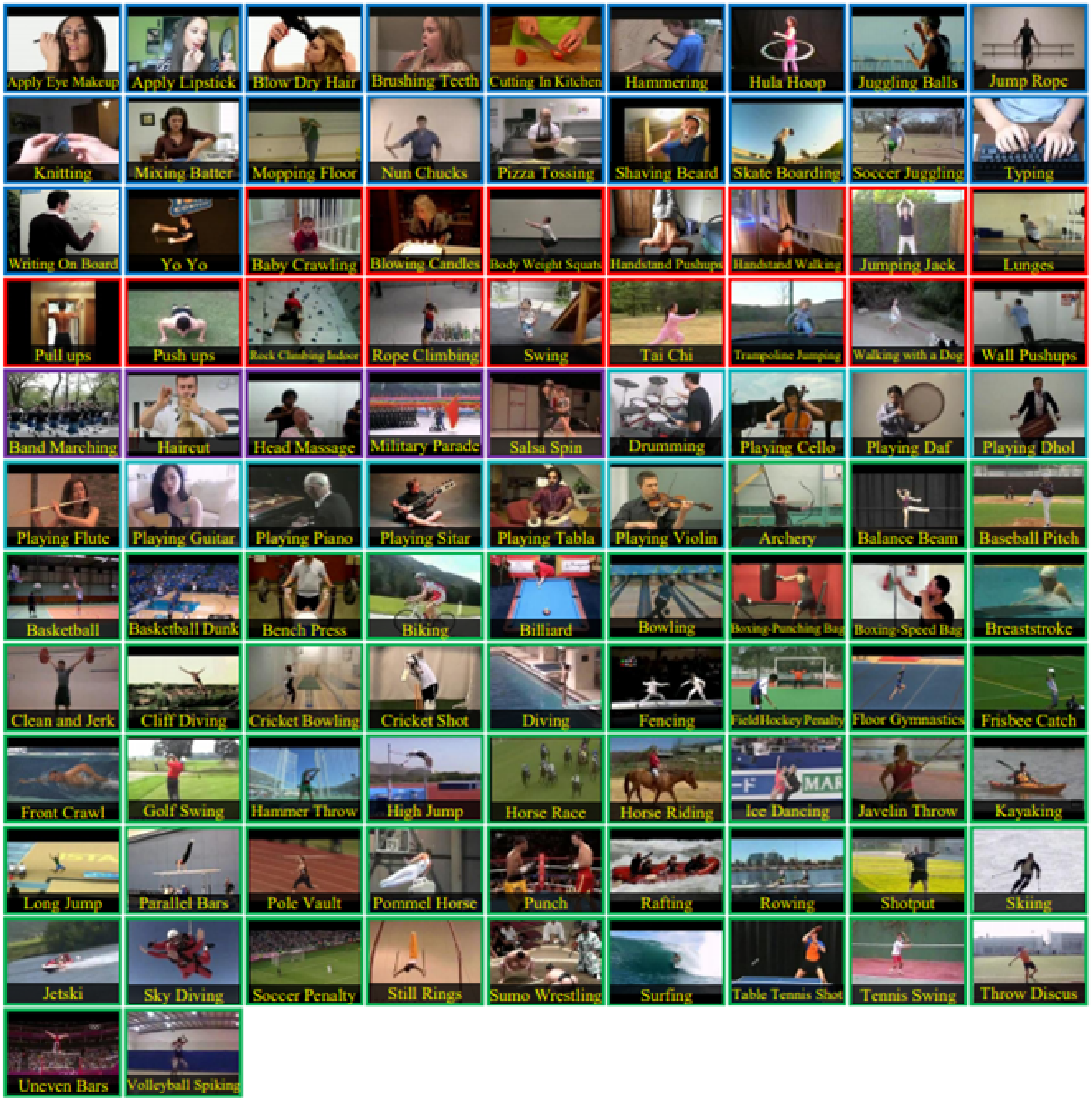

Figure 8. Sample frames of each of the 101 action classes in UCF101. The color of each frame border corresponds to the respective action type: Human-Object Interaction, Body-Motion Only, Human-Human Interaction, Playing Musical Instruments, and Sports. (Soomro et al., 2012)

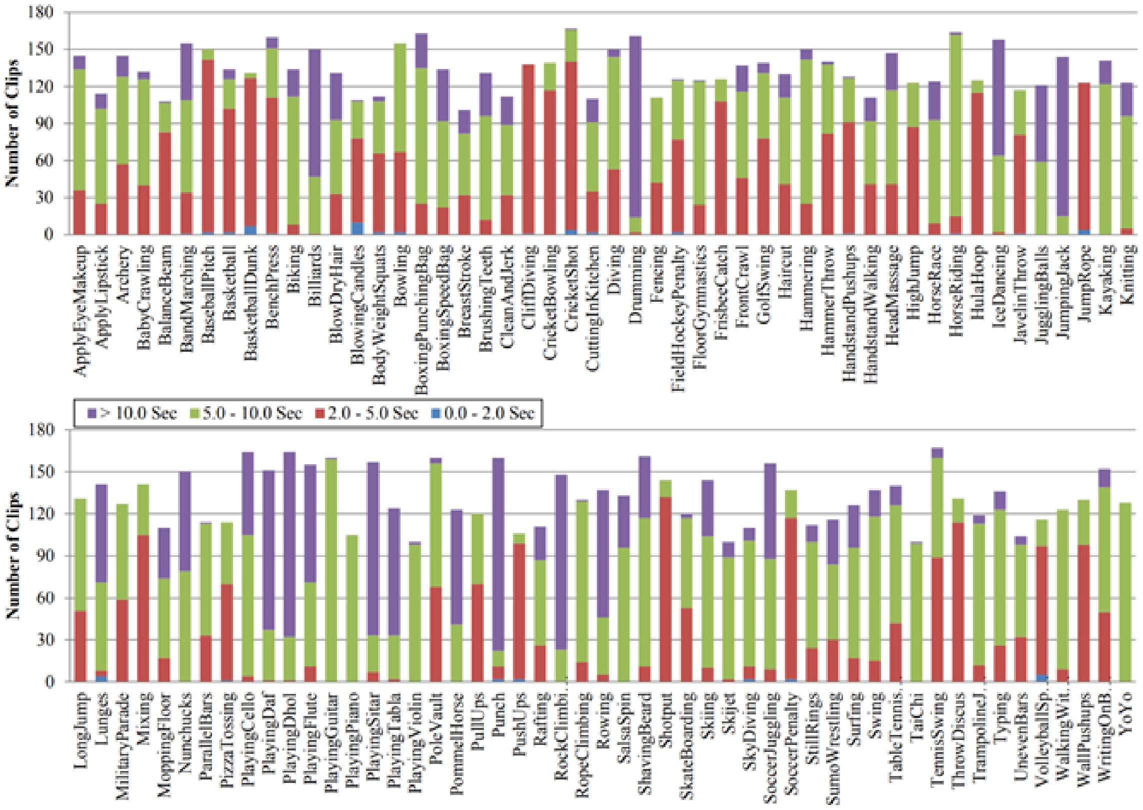

Figure 9. Number of clips per class with clip duration illustrated by color. (Soomro et al., 2012)

3.3 Model Architecture

The entire research model consists of two networks: a BCNN as the front-end network that feeds into an RNN as the back-end network. Initially, the front-end BCNN was intended to utilize the architecture of the VGG16 network, a CNN created by the Visual Geometry Group from Oxford in 2014 (Simonyan and Zisserman, 2014). This is because the VGG16 network achieves 92.7% top-5 accuracy on the ImageNet data set, which contains over 14 million images that span 1,000 classes. This means that the VGG16 model includes the correct class label in its five highest output values with 92.7% accuracy. Although this would provide a good starting point for this research, due to limited computational resources, the front-end BCNN is constructed from a simpler CNN consisting of two weighted layers with dropout applied before every weighted layer. This is following Gal's finding that applying dropout before every weighted layer is an approximate Bayesian inference. Both dropout layers utilize an identical dropout rate of 0.5 according to the precedent set by Gal and Ghahramani (2016). This reduction in the front-end model's size does not impact this research, as its aim is to show a proof of concept.

Figure 10 demonstrates the architecture of the front-end model. The fixed-size input, a 224 x 224 RGB image, passes through two convolutional layers. Both convolutional layers use 32 filters and a kernel with size 3 x 3, which is the smallest kernel size that can capture the concept of left and right, up and down, and center. Each convolutional layer uses a stride of 1 pixel, which is the amount by which the filter shifts, and a zero-padding of 1 pixel around the edges of the input images, which allows the original input size to be preserved at each layer. The next layer is a max-pooling layer that uses a pool size of 3 x 3. Following this series of convolutional and pooling layers, the model has two fully connected layers, one with 50 units and one with 101 units, which is the number of classes in the UCF101 dataset. The activation function of all the layers, aside from the final fully connected layer, is the rectified linear unit activation function (ReLU), which is a piecewise linear function that outputs the input if it is positive and zero otherwise. The final fully connected layer uses a softmax activation function, which converts the input to a vector of categorical probabilities, each between 0 and 1, that sum to 1. However, as shown by Gal and Ghahramani (2016), the output vector of probabilities from the softmax function alone cannot be interpreted as model confidence or uncertainty.

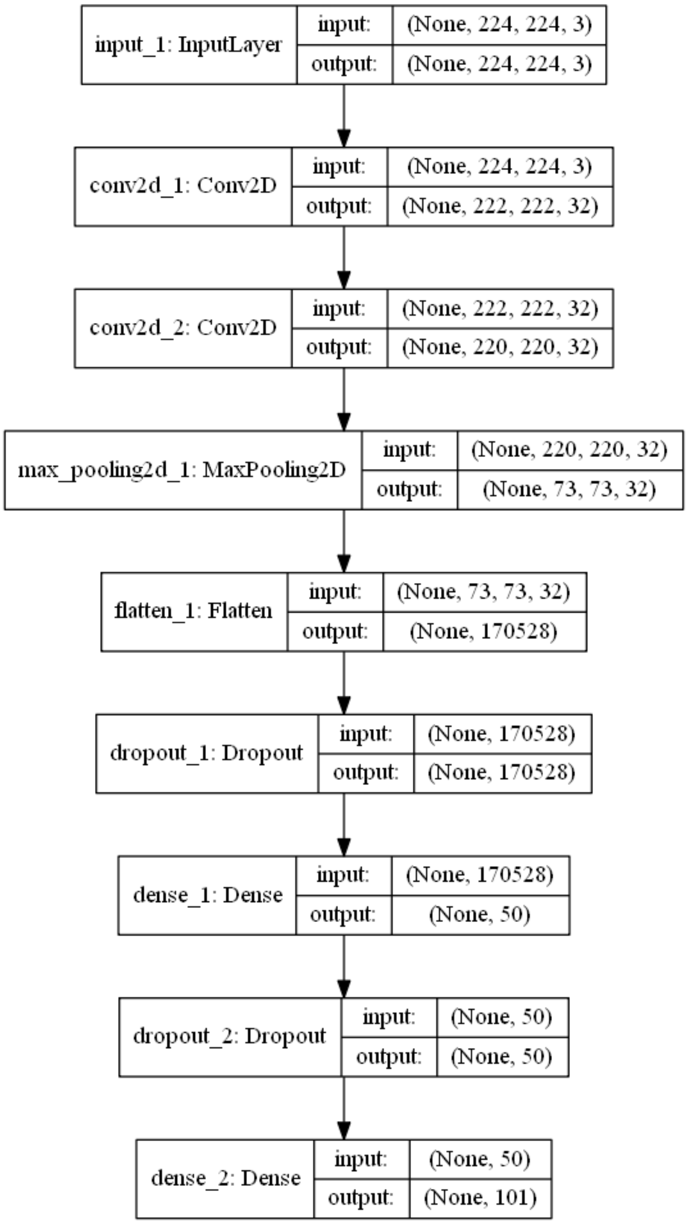

Figure 10. Front-end BCNN Architecture: The dimensions of the inputs and outputs of each layer are preceded by 'None,' signifying that the network can receive any number of images to classify at a time (hyperparameters are discussed in the following chapter).

The output of the BCNN is a list of matrices, one for each video that is classified by the front-end network. Initially, each matrix contains as many rows as a video has frames and as many columns as the data set has classes. Given a video index, the matrix corresponding to that index in the list contains the MCD predicted probabilities for each of the 101 classes for each frame of the video. However, the Keras LSTM layers used to build the back-end network require that all sequences within each batch have the same number of timesteps. To standardize the number of frames across the data passed to the back-end network, all matrices in the list outputted by the front-end network are padded with arrays of zeros to match the highest number of frames that occurs in the data. Using zero padding allows for the list to preserve the original content of the data. This list of matrices, the BCNN output, is fed into an RNN consisting of two LSTM layers, each with 50 units and a softmax activation function. These two LSTM layers are followed by two fully connected layers that contain 50 units and 101 units, respectively. The architecture of the back-end network is represented in Figure 11.

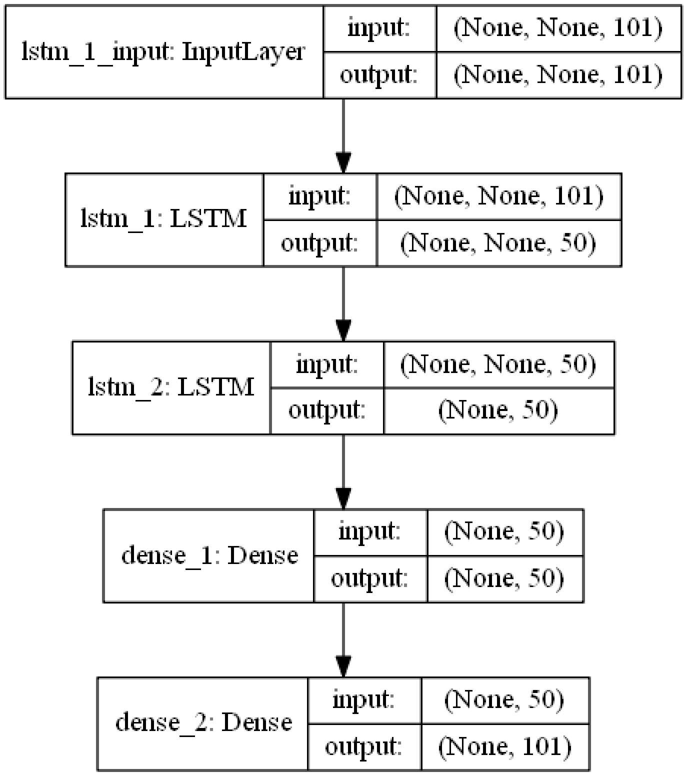

Figure 11. Back-end RNN Architecture

The performance and capabilities of this Bayesian model will be compared to a non-Bayesian, baseline model consisting of the same front-end CNN and back-end RNN structure, but with the dropout layers removed from the front-end network during the testing phase. The Bayesian and baseline front-end networks are identical other than the employment of dropout during the testing phase for the Bayesian front-end network. The same back-end RNN is used after each of these front-end networks.

3.4 Model Parameter Tuning

Because the performance and required training time of a model depend on the specified hyperparameter values, it is important to tune the parameters to find their optimal values. The model parameters include the batch size, epochs, optimizer, weight initialization method, and size of hidden layers. The batch size is the number of data observations that are shown to the network before updating the weights (Géron, 2019). RNNs and CNNs are particularly sensitive to the batch size. The number of epochs is the number of times that the entire training data set is shown to the network during training (Géron, 2019). The optimizer is the algorithm used to update the model weights in response to the loss function results (Géron, 2019). The weight initialization method determines with which random distribution, if any, the network weights are initialized, which heavily affects network training time (Géron, 2019). The size of a hidden layer refers to the number of neurons it contains, which controls the representational capacity of the network at that layer (Géron, 2019).

3.5 Uncertainty Thresholds

The appeal of measuring the network's uncertainty lies in the network's ability to know what it does not know. However, this ability is not useful unless the network is also able to request clarification for those cases about which it is uncertain. To address this, both front-end networks, baseline and Bayesian, will 'flag' any image and its associated video that corresponds to high uncertainty based on respective network thresholds. This allows for a human to inspect and determine the correct class of the flagged images and associated videos, thus avoiding misclassification. This is particularly useful for situations in which ISR resources are identifying a target that may require high certainty before proceeding. All flagged images and their associated videos will be treated as 'non classified,' as the networks will request human intervention for these cases. In this sense, it is expected that the networks will improve in classification accuracy by choosing to classify only the images and videos about which they are more certain.

3.5.1 Bayesian Model Uncertainty Thresholds

For the Bayesian front-end network, the distribution of MCD predictions for an image is used to determine whether the network will classify the image or leave it non-classified. There are many ways to set the network uncertainty threshold for flagging an image, some of which will suit certain data and situations more than others. For the purposes of this research, the uncertainty thresholds are set as follows:

1. An image and its video are flagged if the maximum mean predicted class probability is less than a determined cutoff value, ϕ_1.

2. An image and its video are flagged if there are two or more mean predicted class probabilities greater than another determined value, ϕ_2.

3. An image and its video are flagged if the maximum standard deviation for any of the predicted class probabilities is greater than a determined value, ϕ_3.

These values and their respective sensitivity analyses are specified in the following chapter. These flag thresholds are chosen to ensure reasonable network certainty in the unflagged images.

3.5.2 Baseline Model Uncertainty Thresholds

The baseline front-end network cannot provide the same distribution of predictions that the Bayesian front-end network can. For this reason, in order to gain a semblance of the baseline front-end network's uncertainty, this research applies a frequentist methodology, which treats uncertainty as a probability that is the limit of the relative frequency of an event after many trials (Ambaum, 2012). To accomplish this, augmented data is created from the images and used to construct a distribution of correct predictions and incorrect predictions for each class (Cerliani, 2020). The data is augmented using the Image Data Generator class in Keras (Chollet et al., 2015). The baseline 'non-classified' threshold for each class is found using the same determined value as mentioned in the first Bayesian threshold as a percentile cutoff in the correct predictions distribution for that class. During the testing stage, if the softmax output for the predicted class is below the threshold of that class, then the image is rendered 'non-classified.'

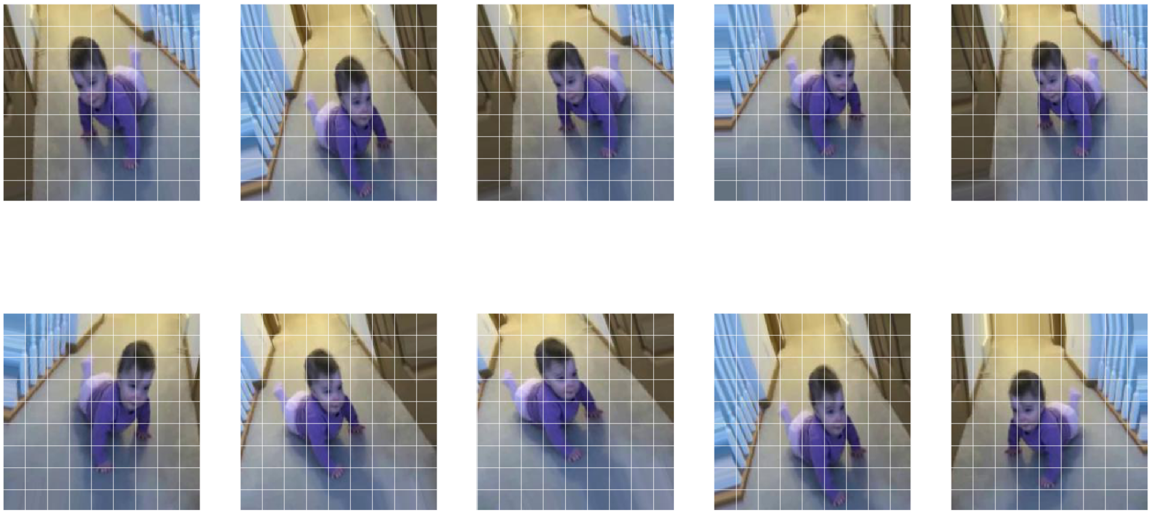

Figure 12. 10 examples of augmented video frames belonging to the class 'Baby Crawling'

3.6 Model Evaluation

For both front-end networks, baseline and Bayesian, as well as for the back-end RNN, the trained models with the lowest validation categorical cross-entropy (CE) loss will be used for the test set. Categorical CE loss, used for multi-class classification, trains the network to output a probability over the C classes for each image (Géron, 2019). CE loss, which is represented in Equation 2, is combined with the softmax activation function, provided in Equation 3, to create categorical CE loss (Géron, 2019). Equation 4 shows the Categorical CE loss equation. In these equations, t_i is the vector of true classes and s_i is the network's vector of predicted scores for each class in C.

$$CE = -\sum_{i \epsilon C} t_i log(s_i) \tag{2}$$

$$f(s)_i = \frac{e^{s_i}}{\sum_{j \epsilon C} e^{s_j}} \tag{3}$$

$$CategoricalCE = -\sum_{i \epsilon C} t_i log(f(s)_i) \tag{4}$$

The overall Bayesian model results will be compared to the baseline model based on model performance metrics and model capabilities. The performance metrics include test accuracy and out-of-scope sample predictions. The model capabilities includes the model's ability to provide uncertainty information regarding its predictions. The baseline and Bayesian front-end networks will also be compared based on the improvement in accuracy after being equipped with the ability to leave images 'non-classified' when the respective uncertainty thresholds are met. Both networks will form their own modified versions on the test set consisting of the images that they choose to classify, respectively.

IV. Results and Analysis

This chapter summarizes notable results from training the models, testing the models, and evaluating model capabilities.

4.1 Hardware and Software

All model training and evaluation were conducted on a Windows 10 Professional PC with an AMD Ryzen 5 5600X CPU, 32 GB RAM, and Sapphire Nitro+ RX 5700XT, as well as the Python 3.7 packages Tensorflow 2.1.0, Keras 2.3.1, and all necessary dependencies.

4.2 Parameter Tuning Results

The Scikit-learn library includes the tool GridSearchCV, which uses k-fold cross validation to exhaustively consider all parameter combinations from a specified grid to determine the optimal parameter values for a model, with a default value of 3 for k (Pedregosa et al., 2011). K-fold cross validation is the process of splitting a training set into k subsets, training the model k times on *k-1* of the subsets, and evaluating the model on the left out subset, each time selecting a different subset for evaluation (Géron, 2019). GridSearchCV computes the mean model accuracy for each combination of specified parameters (Pedregosa et al., 2011). Although only one grid search could determine the optimal model parameters, breaking the process down into multiple grid searches requires less time.

4.2.1 Front-end Network

The first grid search involves a grid of batch sizes ranging from 30 to 70 and epochs ranging from 10 to 100. As a default, the grid search utilized the Adam optimizer, 50 neurons in the first dense layer, no weight initialization, and a test data set of 20% of the data. Table 2 contains the results of this grid search. Two different sets of parameters achieves the best average model accuracy, 0.969744: the first with a batch size of 30 and 100 epochs, and the second with a batch size of 50 and 100 epochs. These two sets of parameters have model accuracy standard deviations of 0.005003 and 0.004511, respectively. These standard deviation values represent the average difference between that model's accuracy and its mean accuracy, which is an indication of its consistency. For this reason, the values 50 and 100 were selected for the batch size and the epochs, respectively, for both the network training and for the following grid searches.

Table 2. Results of Front-end Network Epochs and Batch Size Grid Search

Epochs	Batch Size	Mean Model Accuracy
10	30	0.9666
	40	0.9620
	50	0.9575
	60	0.9065
	70	0.9631
50	30	0.9672
	40	0.9675
	50	0.9691
	60	0.9672
	70	0.9657
100	30	0.9697
	40	0.9654
	50	**0.9697**
	60	0.9670
	70	0.9693

The second grid search explores the differences in model performance across seven optimizers: SGD, AdaGrad, Adadelta, RMSprop, Adam, Adamax, and Nadam. These optimizers utilize the network gradients at each step to update the network weights in the correct direction and by the correct magnitude. A network gradient measures the change in error respective to the change in network weights at each step. The learning rate dictates the amount by which the network weights are updated in response to the estimated error.

SGD, Stochastic Gradient Descent, calculates the error for a random instance in the training data set and updates all the network weights with the same learning rate at each step (Géron, 2019). AdaGrad, Adaptive Gradient Algorithm, is similar to SGD but updates parameters using different learning rates that are tailored to each parameter at each step based on the history of gradients (Géron, 2019). Adadelta is a variant of AdaGrad that limits the amount of previous gradient history taken into account at each step (Ruder, 2016). RMSprop, Root mean square prop, is a variant of AdaGrad that addresses concerns about AdaGrad's radically diminishing learning rates (Géron, 2019). Adam, Adaptive moment estimation, is a combination of AdaGrad and RMSprop that requires less memory and is more efficient (Géron, 2019). Adamax is a variant of Adam that provides less sensitivity to the choice of hyperparameters (Ruder, 2016). Nadam, Nesterov and Adam optimizer, is an Adam variant that updates gradients one step ahead of when Adam does (Géron, 2019).

Table 3 provides the results of this second grid search. Although all optimizers perform well aside from Adadelta, Adamax proves to have the best mean model accuracy. For this reason, the Adamax optimizer is used for the network training and for the following grid searches.

Table 3. Results of Front-end Network Optimizer Grid Search

Optimizer	Mean Model Accuracy
Adamax	**0.9702**
Adam	0.9666
RMSprop	0.9652
Adagrad	0.9627
Nadam	0.9620
SGD	0.9488
Adadelta	0.6588

The third grid search is concerned with the weight initialization distribution of the network's fully connected layers. The grid search includes eight various initialization distributions: uniform, Glorot uniform, normal, He normal, Glorot normal, Lecun uniform, zero, and He normal. As Table 4 shows, all distributions aside from 'He normal' led to good model performance. Using the uniform distribution to initialize the weights of the fully-connected layers provides the best mean model accuracy.

Table 4. Results of Front-end Network Weight Initialization Grid Search

Initializer	Mean Model Accuracy
Uniform	**0.9702**
Glorot Uniform	0.9691
Normal	0.9679
He Uniform	0.9670
Glorot Normal	0.9666
Lecun Uniform	0.9656
Zero	0.9652
He Normal	0.6724

The purpose of the final grid search for the front-end network is to determine the number of neurons in the first fully-connected layer that would lead to the best model performance. Out of the three options shown in Table 5, the network with 50 neurons in the first fully-connected layer achieves the highest mean model accuracy.

Table 5. Results of Front-end Network Dense Layer Neurons Grid Search

Neurons	Mean Model Accuracy
50	**0.9702**
100	0.9656
500	0.9679

4.2.2 Back-end Network

The back-end network requires a smaller set of grid searches as there are fewer parameters to tune. Additionally, this network will train for 100 epochs to match the front-end network. The first grid search for the back-end network evaluates its change in performance with different batch sizes ranging from 30 to 70. Table 6, which contains the grid search results, shows that a batch size of 50 leads to the best performance.

Table 6. Results of Back-end Network Batch Size Grid Search

Batch Size	Mean Model Accuracy
30	0.0050
40	0.0090
50	**0.0120**
60	0.0080
70	0.0090

The second grid search includes three values for the number of neurons in the LSTM layers of the network ranging from 50 to 300. Table 7 indicates that for this grid, 50 neurons in each LSTM layer leads to the best performance.

Table 7. Results of Back-end Network LSTM Layers Neurons Grid Search

Neurons	Mean Model Accuracy
50	**0.0060**
100	0.0030
300	0.0040

The next grid search includes three values for the number of neurons in the dense layers of the network ranging from 50 to 300. Table 8 shows that the best performance occurred with 50 neurons in the dense layer, as well.

Table 8. Results of Back-end Network Dense Layer Neurons Grid Search

Neurons	Mean Model Accuracy
50	**0.0080**
100	0.0070
300	0.0070

The final grid search for the back-end network includes three different optimizers, RMSprop, Adadelta, and Adam. This is because these three optimizers have been found to lead to better model performance with LSTMs (Ruder, 2016). For this final grid, the optimizer Adam leads to the best mean model accuracy.

Table 9. Results of Back-end Network Optimizer Grid Search

Optimizer	Mean Model Accuracy
Adam	**0.0130**
RMSprop	0.0120
Adadelta	0.0050

4.2.3 Summary

In summary, the front-end network will train for 100 epochs with the Adamax optimizer, utilize a batch size of 50, and contain 50 neurons in the first dense layer initialized with a uniform distribution. The back-end network will train for 100 epochs, as well, using the Adam optimizer, a batch size of 50, and 50 neurons in each LSTM and dense layer, aside from the last dense layer.

4.3 Model Training

Figure 13 shows that for the front-end model, both the training accuracy and test accuracy increase during training and level off towards the end of the 100 epochs. However, the gap between the train accuracy and test accuracy curves is consistently almost a 0.10 difference, indicating that overfitting is a possibility. Figure 14 shows that for the front-end model, while the training loss decreases during training and levels off, the test loss increases over the course of the 100 epochs. The steady increase of the test loss confirms that the model is experiencing overfitting. This means that the model is focusing on the noise of the training data, which improves its performance on the training set but leaves it unable to generalize learned features to the test set (Géron, 2019).

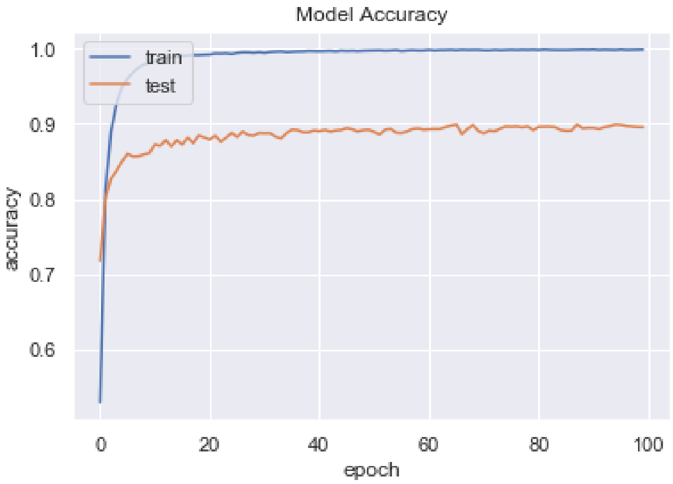

Figure 13. Front-end Model Training and Test Accuracy by Epoch

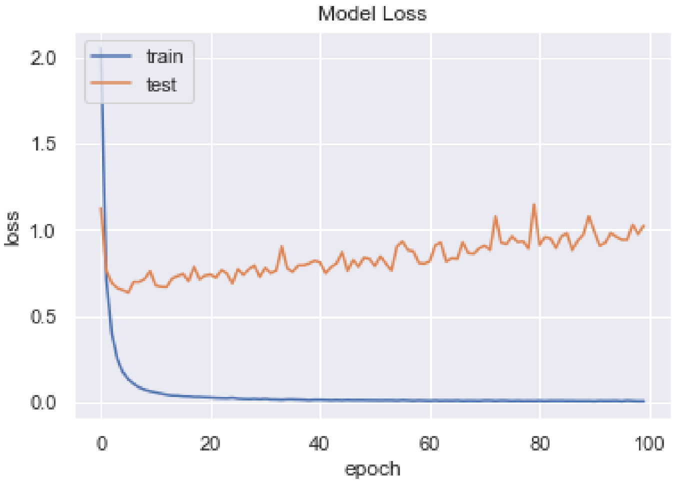

Figure 14. Front-end Model Training and Test Loss by Epoch

Figure 15 shows that for the back-end network, both the training accuracy and test accuracy increase only during the initial steps of training and then level off quickly. The gap between the train accuracy and test accuracy curves is consistently small, indicating that overfitting is unlikely for the back-end network (Géron, 2019). Figure 16 shows that both the training loss and test loss decrease during the initial steps of training and level off quickly. Both Figures 15 and 16 indicate that the back-end network achieves very low performance in its classification tasks.

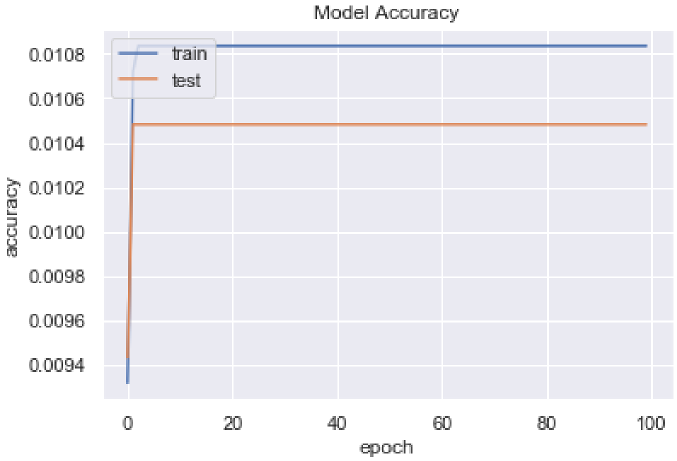

Figure 15. Back-end Model Training and Test Accuracy by Epoch

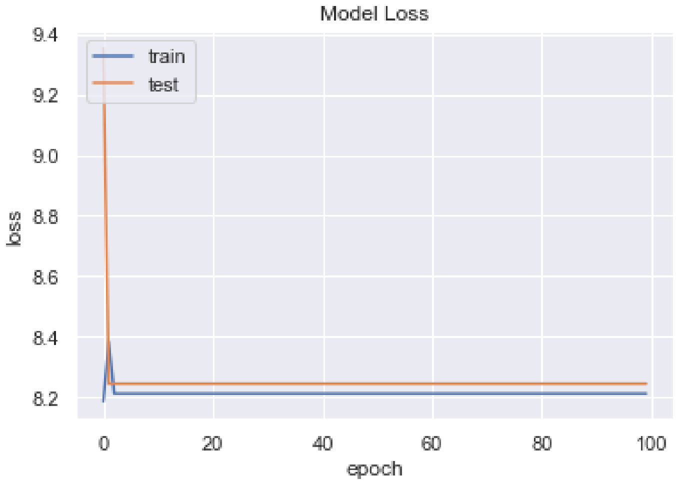

Figure 16. Back-end Model Training and Test Loss by Epoch

4.4 Model Evaluation and Comparison

First, the baseline model and Bayesian model are compared on their performance in classifying the entire test set without the ability to leave images non-classified. Next, both models are evaluated on the measures of uncertainty that they can each provide for their predictions. This includes a sensitivity analysis on the non-classified thresholds for each model. Finally, both models are evaluated on their performance on the subsection of the test set they choose to classify based on the uncertainty thresholds. The test set used for evaluation spans all 101 classes and contains 3,782 videos, which collectively contain 28,890 images.

4.4.1 Model Performance on Whole Test Set

The baseline front-end model achieves 25.2% accuracy on the whole test set. Feeding the baseline front-end network's predictions to the back-end network, the model as a whole achieves 1.3% accuracy. The Bayesian front-end network achieves 20.9% accuracy on the whole test set. With the Bayesian front-end network predictions as input for the back-end network, the Bayesian model as a whole achieves 1.3% accuracy. Both front-end networks achieve an accuracy far below the average training and test accuracy that occurred in the training phase, indicating that the model parameters are over-fit to the training data set. The results of the performances of the back-end network align with the poor back-end training results presented in Figures 15 and 16.

4.4.2 Non-Classified Threshold Sensitivity Analysis

In order to keep the results of the two different front-end networks comparable, the final cutoff for the predicted class probability value, ϕ_1, will be the same for both networks. Note that this value means different things to each of the networks. For the Bayesian front-end network, the cutoff value ϕ_1 represents the the minimum value of certainty in the predicted most likely class in order for an image to be classified; if the cutoff value is not met, then the image is left non-classified. For the baseline front-end network, the cutoff value ϕ_1 represents the percentile of the distribution of correctly classified augmented data from which to form the predicted probability threshold. For example, with a ϕ_1 value of 0.1, each baseline prediction must meet the 10$^{\text{th}}$ percentile of correctly predicted augmented data values for the predicted class in order to be classified rather than non-classified.

This sensitivity analysis explores the effects of changing the cutoff value ϕ_1, which affects both front-end networks, as well as the effects of changing the other two uncertainty thresholds, ϕ_2 and ϕ_3, of the Bayesian front-end network. The analyzed ϕ_1 values range from 0.1 to 0.9.

Baseline Model

Figure 17 depicts the results of the ϕ_1 cutoff value sensitivity analysis regarding the baseline front-end model. As ϕ_1 increases, the maximum threshold for the 101 classes increases quickly and approaches 1, while the mean threshold and minimum threshold both steadily increase. The blue line in Figure 17 represents the percentage of the test set images that the baseline front-end network classifies when given the option to classify or not. This percentage of classified images steadily decreases as ϕ_1 increases. This is because as ϕ_1 grows larger, the thresholds become consistently stricter for all 101 classes, leading the baseline front-end network to leave more images non-classified. The green line in Figure 17 represents the accuracy of the baseline front-end network on the modified test set, which consists of all the test set images that meet the threshold to be classified. Whereas the blue line indicates that the size of this modified test set decreases with an increasing cutoff value ϕ_1, the green line shows that the baseline network accuracy increases with the cutoff value. This is due to the thresholds being more and more selective as ϕ_1 increases, allowing the model to only classify those images of test set about which it is very certain.

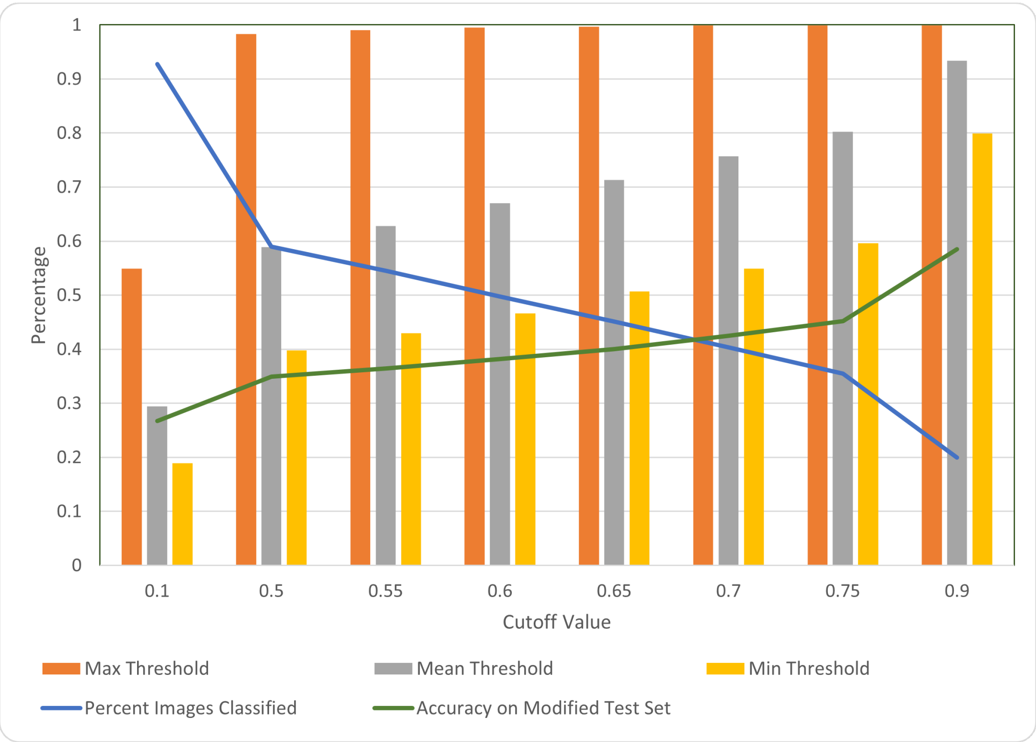

Figure 17. ϕ_1 Cutoff Value Sensitivity Analysis for Baseline network Uncertainty Threshold

For the baseline front-end network to classify at least half of the test set images, which is at least 14,445 images, the ϕ_1 cutoff value must be at most 0.57. The thresholds obtained at a percentile of the correctly predicted augmented data probabilities higher than $\phi_1 = 0.57$ are too strict for the baseline network to classify even half of the test set.

Note that the frequentist thresholds used to create a semblance of certainty for the baseline network do not provide an exact measure of uncertainty for the network ((Ambaum, 2012)). Instead, each threshold provides the minimum probability associated with the cutoff value percentile with which the baseline network most frequently predicts the correct class.

Bayesian Model

The three threshold criteria that renders an image non-classified by the Bayesian front-end network, as mentioned in the previous chapter, are as follows:

1. An image and its video are flagged if the maximum mean predicted class probability is less than a determined cutoff value ϕ_1.

2. An image and its video are flagged if there are two or more mean predicted class probabilities greater than a determined value ϕ_2.

3. An image and its video are flagged if the maximum standard deviation for any of the predicted class probabilities is greater than a determined value ϕ_3.

The images that do not meet any of these three criteria form the modified test set and are classified by the network. Changes in the ϕ_1 cutoff value do not affect the modified test set size or the Bayesian network's accuracy on it while the standard deviation threshold ϕ_3 is less than 0.1. The same is true for changes in ϕ_2: changes in the percent value for this threshold do not affect the modified test set or the network's accuracy while the standard deviation threshold ϕ_3 is less than 0.1. This indicates that the certainty of the Bayesian front-end network in its predictions relies most heavily on the value of the standard deviation threshold, ϕ_3. With ϕ_1 and ϕ_2 held constant, increasing the standard deviation threshold ϕ_3 increases the size of the modified test set and decreases the network's accuracy on the modified test set. Once the standard deviation threshold ϕ_3 exceeds 0.1, increasing the cutoff value ϕ_1 steadily decreases the size of the modified test set and steadily increases the Bayesian network's accuracy on the modified test set. Once the standard deviation threshold ϕ_3 exceeds 0.1, increasing ϕ_2 steadily increases the size of the modified test set and steadily decreases the Bayesian network's accuracy on the modified test set.

Figure 18 shows the influence of the standard deviation threshold value ϕ_3 on the modified test set size and network accuracy while all other threshold values are held constant. As the standard deviation threshold value exceeds 0.3, the influence of this threshold on the modified test set and network accuracy diminishes and levels off.

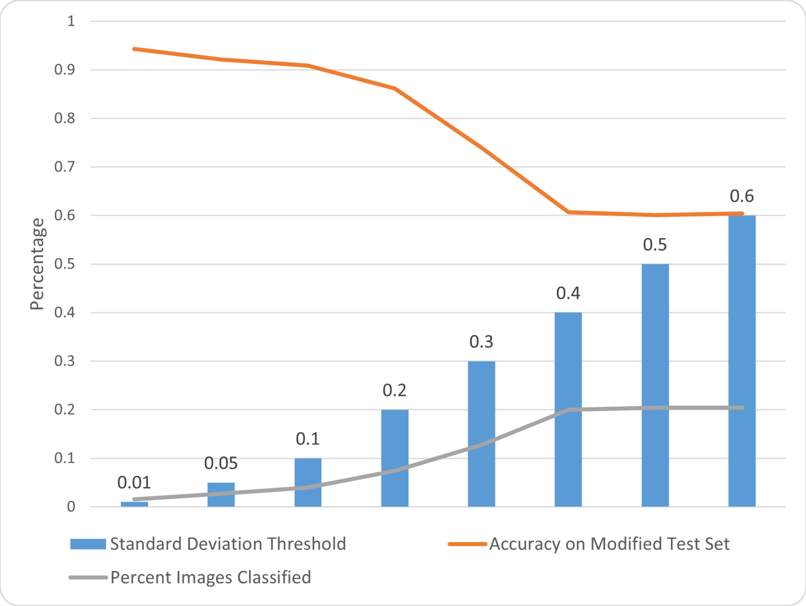

Figure 18. Standard Deviation Value (ϕ_3) Sensitivity Analysis for Bayesian network Uncertainty Threshold

To focus on the influence of changes in the cutoff value ϕ_1, Figure 19 holds the standard deviation threshold ϕ_3 at a constant 0.4. This shows that past the boundaries of the modified test set that are determined by the standard deviation threshold, increasing the cutoff value for the mean predicted class probability reduces the size of the modified test set while increasing the Bayesian network's accuracy on it.

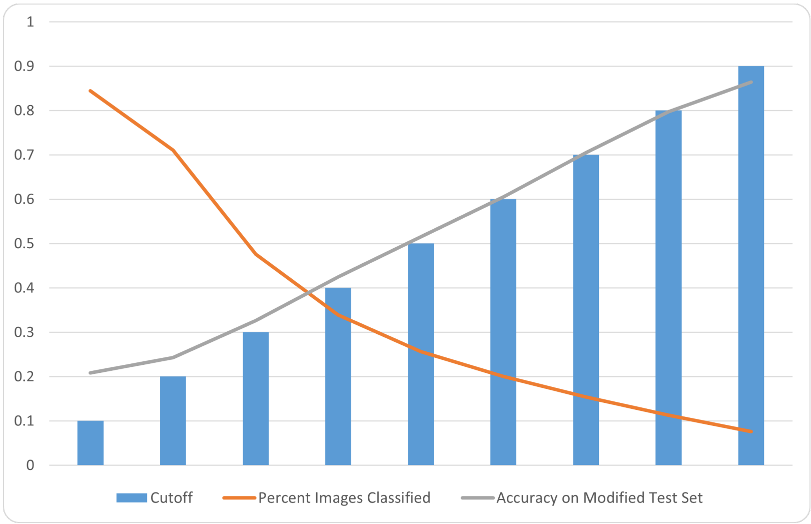

Figure 19. Cutoff Value ϕ_1 Sensitivity Analysis for Bayesian network Uncertainty Threshold

In order for the Bayesian front-end network to classify at least half of the test set images, which means at least 14,445 images, the standard deviation threshold ϕ_3 must be greater than 0.32. At any standard deviation threshold value less than 0.32, the other two threshold values do not allow the network to classify even half of the test set, no matter how lenient. In fact, even with a standard deviation threshold of 0.32, the Bayesian network only classifies half of the test set with a cutoff value ϕ_1 of 0.10 and a value of 0.5 for ϕ_2. With these values, the Bayesian network classifies all images that have maximum mean predicted class probability greater than 0.10, no more than one class with a mean predicted probability greater than 0.50 (which could not occur regardless), and with no standard deviations greater than 0.32 for the predicted class probabilities.

For research purposes, the final thresholds for all three Bayesian uncertainty thresholds and for the baseline uncertainty threshold are chosen to enable the Bayesian front-end network to classify at least 20% of the whole test set, which means at least 5,778 images. This will enable a more thorough analysis of the network's capabilities. To accomplish this, the cutoff value ϕ_1 for both the Bayesian and baseline model is 0.6, the ϕ_2 value for Bayesian uncertainty is 0.25, and the Bayesian standard deviation threshold ϕ_3 value is 0.4. For the baseline network, this means that for an image to be classified, the softmax output for the predicted class must be at least the value of the 60^{th} percentile of that class's correctly predicted augmented data softmax probabilities. For the Bayesian network, this means that the mean predicted class probability must be at least 0.6, no more than one mean class probability can be over 0.25, and no standard deviation of the class probabilities can exceed 0.4.

4.5 Performance on Out of Scope Samples

Out-of-scope samples are data that do not belong to the data set being used to train and test a model. Presenting out-of-scope samples to each of the front-end networks for classification will test their ability to wield their uncertainty thresholds in a new situation ((Sterbak, 2020)). Because out-of-scope samples do not belong to any class in the data set, each front-end network should ideally leave them as non-classified. Additionally, should one or both of the front-end networks classify a large number of out-of-scope samples as one of the 101 classes, it would indicate that the model does not understand the concept of that class. To evaluate the ability of both front-end networks to identify out-of-scope samples, 1,000 randomly-generated images are presented to the networks for classification. Figure 20 provides an example of a randomly generated image used as an out-of-scope sample.

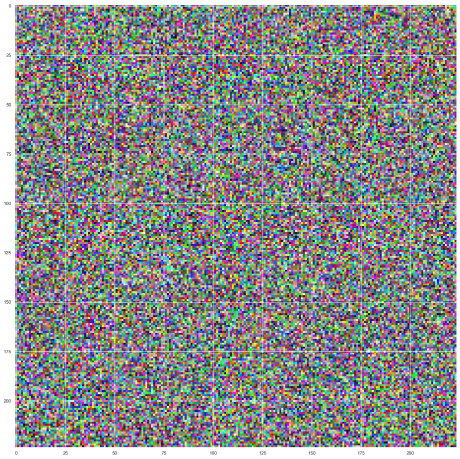

Figure 20. Example of Randomly Generate Image as Out-of-scope Sample

Out of the 1,000 randomly-generated images, the baseline network leaves 959 of the images non-classified, while the Bayesian network leaves 984 images non-classified. In this evaluation, both models are able to identify the vast majority of out-of-scope samples and leave them non-classified. This shows that each model is not confident enough about these 'trick questions' to classify them. However, when forced to classify all 1,000 images, the results are revealing about the model insufficiencies. The baseline network, when forced to classify all images, classifies 88.9% of the images as 'Throw Discus,' 10.7% of the images as 'Hammering,' and 0.4% of the images as 'Swing.' Similarly, when forced to classify all images, the Bayesian network classifies 81.9% of the images as 'Throw Discus' and 18.1% of the images as 'Hammering.' These results indicate that neither front-end model understands the concept of the classes 'Throw Discus' or 'Hammering.' Additionally, the baseline network does not understand the concept of the class 'Swing' either. This could be due to too much variation in the data for each of these classes, meaning that the videos and their frames are too widely varied in terms of camera angle, lighting, movement, etc and the model is unable to learn concrete features that associate with these classes.

4.6 Model Performance on Modified Test Set

Figure 21 shows the ratio of classified to non-classified for both images and videos for the networks with the chosen threshold values. The baseline network classifies significantly more images and videos than the Bayesian network.

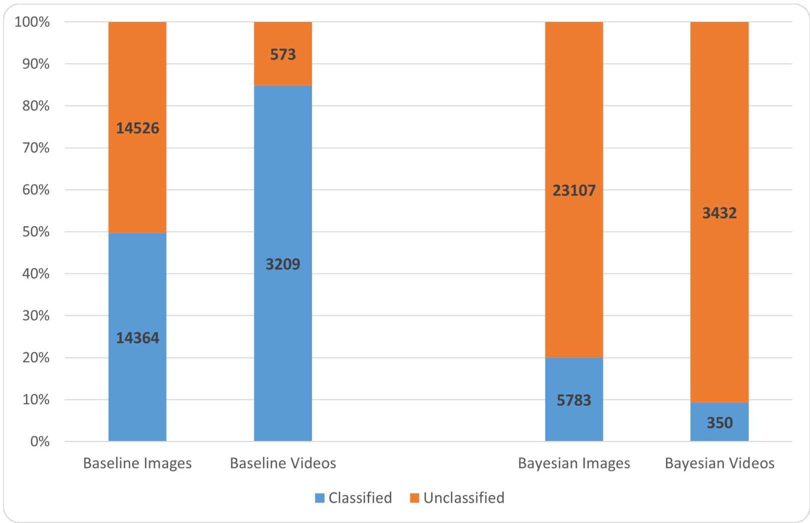

Figure 21. Classified to non-classified ratio for each front-end network

On the modified test set, the baseline front-end network achieves 38.2% accuracy, which is an increase of 13 percentage points. The baseline model as a whole achieves 1.40% accuracy on the modified test set of videos, which is roughly the same as the baseline model performance on the entire test set of videos. On the modified test set, the Bayesian front-end network achieves 61.0% accuracy, which is an increase of 40.1 percentage points. The Bayesian model as a whole achieves 0% accuracy on the modified test set of videos, which is roughly 1.3 percentage points less than the Bayesian model performance on the entire test set of videos. Figure 22 provides a visual of the difference between whole test set performances and modified test set performances. As specified in Figure 22, the baseline front-end and back-end experienced a 52% and 8% increase in accuracy, respectively. The Bayesian front-end experienced a 192% increase in accuracy, while the Bayesian back-end had a 100%

decrease in accuracy.

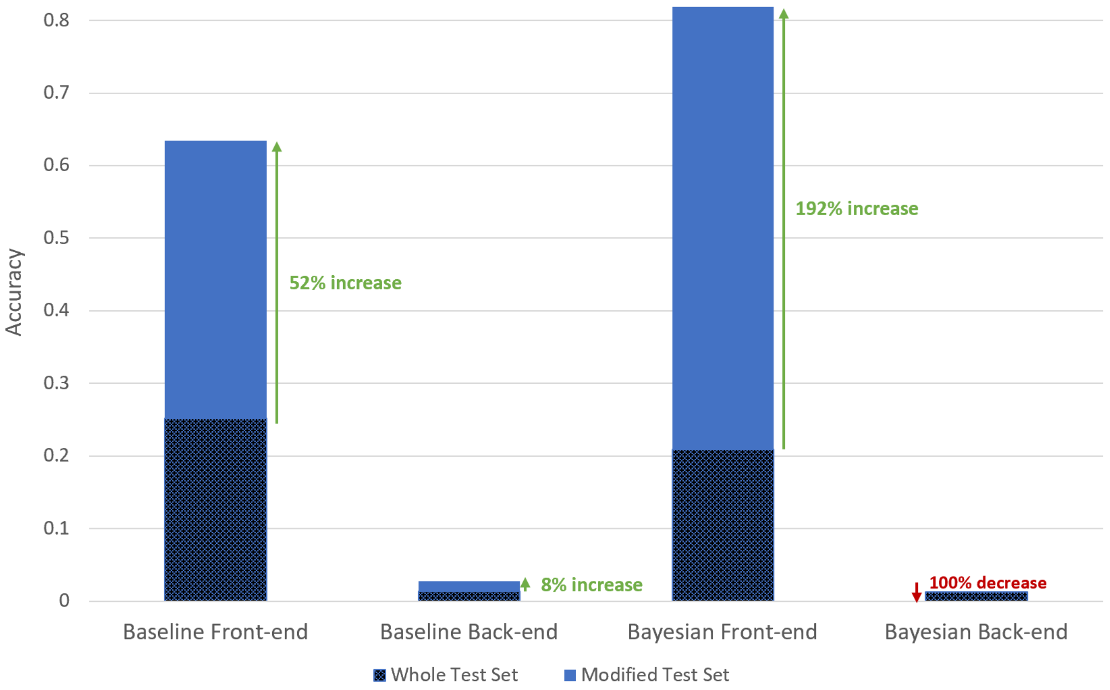

Figure 22. Model performances on whole test set and on modified test set. Here the entire bar represents the accuracy of a model on the modified test set, while the shaded portion of the bar represents the accuracy of a model on the whole test set.

4.7 Incongruity between the Two Front-end Networks

Of the 23,107 images left non-classified by the Bayesian front-end network, 8,610 of them were classified incorrectly by the baseline network. The 8,610 images account for roughly 30% of the whole test set. Figure 23 provides an example of one of these images, a video frame belonging to the class 'Bowling.' Figure 24 contains a visual representation of the distribution of MC predicted probabilities for the image in Figure 23. In Figure 24, each of the 101 classes of the data set are represented by a horizontal bar. For every one of the MCD predictions, the probabilities are plotted on the bars, creating a color gradient that represents a class's predicted probability variation for the image. The bolded, black bars represent the mean of the MCD predicted probabilities.

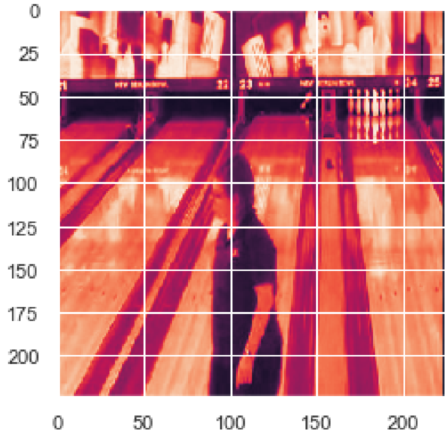

Figure 23. Video frame belonging to the class 'Bowling' that was non-classified by the Bayesian model and incorrectly classified by the Baseline model

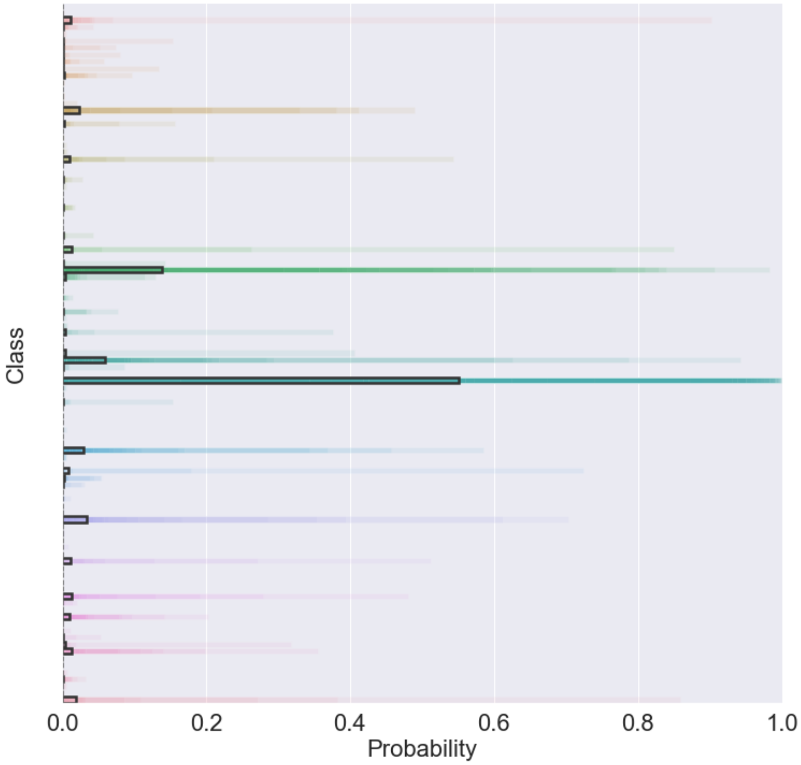

Figure 24. MCD Predictions for all 101 classes for Bowling frame

While Figure 24 provides an overview of the MCD predicted probabilities for every class in the data set, displaying the wide range of predictions and variations.

Figure 25 contains only the classes with the three highest mean MCD predicted probabilities, 'Mopping Floor,' 'Head Massage,' and 'Lunges.' Figure 25 indicates that the Bayesian front-end network most consistently predicted high probabilities for the class 'Mopping Floor' for the 'Bowling' video frame in Figure 23. The color gradients of the 'Head Massage' and 'Lunges' bars indicate that the network MCD predictions vary more widely for these classes for this image. If the Bayesian front-end network were forced to classify this image, it would defer to the highest mean MCD predicted probability, 'Mopping Floor,' which would incorrect. The Bayesian network was right to leave the image non-classified because it would have been a misclassification. The baseline network, on the other hand, chose to classify the 'Bowling' video frame and predicted the class 'Mopping Floor.' This is a prime example of the value of accurate uncertainty measures. In this situation and 8,609 others of the test set, both models landed on the same misclassification, 'Mopping Floor,' but only one of the models was able to use its high uncertainty in the prediction to decide not to classify the image at all.

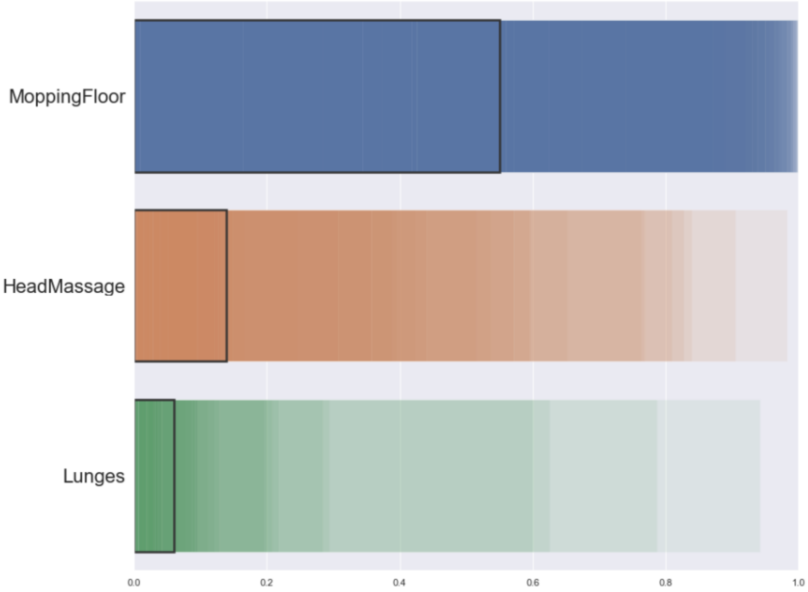

Figure 25. MCD Predictions for top three classes for Bowling frame

V. Conclusions and Recommendations

5.1 Conclusions

This research aims to explores the results of utilizing a blend of neural networks to classify videos and images. This blend consists of a CNN for image classification, an RNN for sequences of images (video) classification, and a BNN to equip the model with the ability to measure its uncertainty in each prediction. While this model's regular performance on a given test set do not outperform its non-Bayesian equivalent, when it is allowed to use uncertainty thresholds to discern which images and videos about which it is confident enough to classify or not, this model increases its accuracy by almost 200%.

5.2 Recommendations

The purpose of this research is to provide a proof of concept and so has been conducted on a small scale. Should the available resources and data be scaled up, the advantages provided by a measure of a model's uncertainty incorporated with each prediction would likely grow, as well. For this reason, expanding the size of the model could prove beneficial.

In future work, the back-end RNN should be left out of the blend of models. Not only does the back-end network perform poorly on the data, the front-end network, Bayesian or not, is sufficient in accounting for the subject of a video. This is due to the fact that frequently, any given frame of a video from the UCF101 data set is representative of the rest of the frames of that video. For this reason, it could be unnecessary to classify a video as a sequence of all of its frames using an RNN. If it is the goal to retain the RNN in the network blend, then a different data set should be sought.

Future work should also test a wider variety of uncertainty thresholds for the Bayesian model in its construction of a modified test set to classify.

Appendix: Code

```
import keras
from keras.models import Model, Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, Input, LSTM
from keras.layers import  MaxPooling2D as MaxPool2D
from tqdm import tqdm
import matplotlib.pyplot as plt
import numpy as np
import csv
import pandas as pd
from keras.preprocessing import image
from sklearn.model_selection import train_test_split
import cv2
from glob import glob
import os
os.environ[’TF_CPP_MIN_LOG_LEVEL’] = ’2’
import math
import seaborn as sns
from sklearn.metrics import accuracy_score
from keras.utils.vis_utils import plot_model
sns.set()
```

(Sterbak (2020))

```
def get_dropout(input_tensor,
                p=0.5,
                mc = False):
    if mc:
        return Dropout(p)(input_tensor, training=True)
    else:
        return Dropout(p)(input_tensor)
def build_frontend(num_classes,
              dropout,
              input_shape = (224, 224, 3),
              mc = False):
    inp = Input(input_shape)
    x = Conv2D(32, (3, 3), input_shape = input_shape,
            activation=’relu’)(inp)
    x = Conv2D(32, (3, 3), activation = ’relu’)(x)
    x = MaxPool2D(pool_size=(3, 3))(x)
    x = Flatten()(x)
    x = get_dropout(x, p=dropout, mc=mc)
    x = Dense(units = 500, activation = 'relu')(x)
    x = get_dropout(x, p = dropout, mc = mc)
    out = Dense(units=num_classes,
        activation="softmax")(x)
    model = Model(inputs=inp, outputs=out)
    model.compile(loss=keras.losses.categorical_crossentropy,
                  optimizer=keras.optimizers.Adamax(),
                  metrics=['accuracy'])
    return model
def get_frontend_data_setup(training_file_name =
        'train_new.csv'):
    train = pd.read_csv(training_file_name)
    num_classes = train['class'].nunique()
    class_labels = train['class'].unique().tolist()
    train_image = []
    input_shape = (224, 224, 3)
    for i in tqdm(range(train.shape[0])):
        img = image.load_img('train_1/'+train['image'][i],
                            target_size = input_shape)
        # converting it to array
        img = image.img_to_array(img)
        # normalizing the pixel value
        img = img/255.0
        # appending the image to the train_image list
        train_image.append(img)
    X = np.array(train_image)
    del train_image
    y = train['class']
    del train
    X_train, X_test, y_train, y_test = train_test_split(X,
                        y,
                        random_state=42,
                        test_size=0.2, stratify = y)
    del X
    y_train = pd.get_dummies(y_train)
    y_test = pd.get_dummies(y_test)
    return num_classes, class_labels, input_shape,
            X_train, X_test, y_train, y_test
def get_class_labels(training_file_name = 'train_new.csv'):
    train = pd.read_csv(training_file_name)
    class_labels = train['class'].unique().tolist()
    return class_labels
def get_backend_data_setup2(file_name = 'trainlist01.txt'):
    f = open(file_name, "r")
    temp = f.read()
    videos = temp.split(’\n’)
    # creating a dataframe having video names
    train = pd.DataFrame()
    train[’video_name’] = videos
    train = train[:-1]
    # get rid of the weird number at the end
    for i in range(train.shape[0]):
        train[’video_name’][i] =
                    train[’video_name’][i].split(’ ’)[0]
    video_names = train[’video_name’]
    return video_names
def get_back_input2(video_names,
                   mc_model,
                   num_classes,
                   n_iter = 500,
                   batch_size = 50,
                   mc_preds = False):
    frames = []
    frame_labels = []
    frame_mc_preds = []
    belongs_to_video = []
    output = []
    output_labels = []
    for i in tqdm(range(video_names.shape[0])):
        if i != 341:
            count = 0
            videoFile = video_names[i]
            cap = cv2.VideoCapture(’./Videos/’+
                        videoFile.split(’ ’)[0])
            frameRate = cap.get(5) #frame rate
            files = glob(’temp/*’)
            for f in files:
                os.remove(f)
            while(cap.isOpened()):
                frameId = cap.get(1) #current frame number
                ret, frame = cap.read()
                if (ret != True):
                    break
                if (frameId % math.floor(frameRate) == 0):
                    filename =’temp/’ + "_frame%d.jpg" %
                            count;count+=1
                    cv2.imwrite(filename, frame)
            cap.release()
            images = glob("temp/*.jpg")
            the_images = []
            for j in range(len(images)):
                img = image.load_img(images[j],
                        target_size=(224,224,3))
                img = image.img_to_array(img)
                img = img/255
                the_images.append(img)
                frames.append(img)
                belongs_to_video.append(i)
                frame_labels.append(videoFile
                        .split(’/’)[1].split(’_’)[1])
            the_images = np.array(the_images)
            print(str(i) + ": " + str(the_images.shape))
            video_frames_predictions = []
            for j in range(n_iter):
                 y_p = mc_model.predict(the_images,
                        batch_size=batch_size)
                 video_frames_predictions.append(y_p)
            mean_video_frames_predictions = np.mean
                    (video_frames_predictions,
                    axis = 0)
            output.append(mean_video_frames_predictions)
            output_labels.append(videoFile
                        .split(’/’)[1].split(’_’)[1])
    frames = np.array(frames)
    frame_labels = pd.DataFrame(frame_labels,
                    columns=[’class’])
    frame_labels = pd.get_dummies(frame_labels)
    output = standardize_matrix_size(output,
                    num_classes)
    if mc_preds:
        frame_mc_preds = get_mc_predictions(
                    mc_model, frames, n_iter)
        return output, output_labels,
                    frames, frame_labels,
                    belongs_to_video, frame_mc_preds
    else:
        return output, output_labels, frames,
                    frame_labels, belongs_to_video
```

(Sterbak (2020))

```
def get_mc_predictions(mc_model,
                       test_set_images,
                       n_iter = 500):
    mc_predictions = []
    for i in tqdm(range(n_iter)):
        y_p = mc_model.predict(test_set_images,
            batch_size=50)
        mc_predictions.append(y_p)
    # make model predictions on the test set 500 times
    return mc_predictions
def get_flag1(p0, cutoff, stdev):
    flag = False
    if ((p0.mean(axis=0).max() < cutoff) or
        (len([i for i in p0.mean(axis=0)
                if i > 0.25]) >= 2) or
        (p0.std(axis=0).max() > stdev)):
        flag = True
    return flag
def get_most_uncertain_images(test_set_images,
            mc_predictions, stdev):
    max_means = []
    max_vars = []
    class_preds = []
    flags = []
    for idx in range(test_set_images.shape[0]):
        px = np.array([p[idx] for
                    p in mc_predictions])
        class_preds.append(px.mean(axis=0).argmax())
        max_means.append(px.mean(axis=0).max())
        max_vars.append(px.std(axis=0)
                [px.mean(axis=0).argmax()])
        flags.append(get_flag1(px, 0.6, stdev))
    test_uncertainties = pd.DataFrame(
                            {’max_means’:max_means,
                            ’max_vars’: max_vars,
                           ’class_pred’: class_preds,
                            ’flagged’: flags})
    top_by_prob = test_uncertainties
                .sort_values(by = [’max_means’])
    top_by_prob = top_by_prob.reset_index()
    top_by_var = test_uncertainties
                    .sort_values(by = [’max_vars’],
                     ascending = False)
    top_by_var = top_by_var.reset_index()
    return top_by_prob, top_by_var
```

(Sterbak (2020))

```
def check_for_model_misunderstanding(mc_model,
                                     num_classes,
                                     class_labels,stdev,
                                     input_shape = (224, 224, 3),
                                     num_random_images = 100):
    posterior_counts = [0 for x in range(num_classes)]
    top_class_probs = np.empty(num_random_images)
    top_class_vars = np.empty(num_random_images)
    num_flags = 0
    for j in tqdm(range(num_random_images)):
        random_img = np.random.random(input_shape)
        random_predictions = []
        for i in range(100):
            y_p = mc_model.predict(np.array([random_img]))
            random_predictions.append(y_p)
        p0 = np.array([p[0] for p in random_predictions])
        index = p0.mean(axis=0).argmax()
        posterior_counts[index] = posterior_counts[index]+1
        top_class_probs[j] = p0.mean(axis=0)[index]
        top_class_vars[j] = p0.std(axis=0)[index]
        if get_flag1(p0, 0.6. stdev):
            num_flags = num_flags+1
    above_zero = []
def baseline_check_for_model_misunderstanding(model,
                                     num_classes,
                                     class_labels,
                                     thresh,
                                     input_shape = (224, 224, 3),
                                     num_random_images = 100):
    posterior_counts = [0 for x in range(num_classes)]
    num_flags = 0
    for j in tqdm(range(num_random_images)):
        random_img = np.random.random(input_shape)
        y_p = model.predict(np.array([random_img]))
        index = y_p.argmax()
        posterior_counts[index] = posterior_counts[index]+1
        if y_p.max() < thresh[int(index)]:
            num_flags = num_flags+1
    print()
    print('Percentage of randomly generated images flagged: ')
    print(str(100* num_flags/num_random_images) + '%')
    print()
    above_zero = []
    for index in range(len(posterior_counts)):
        if posterior_counts[index] > 0:
            above_zero.append(index)
def build_backend(num_classes):
    backend = Sequential()
    backend.add(LSTM(50,
                     return_sequences = True,
                     input_shape = (None, num_classes)))
    backend.add(LSTM(50))
    backend.add(Dense(50,
                      activation = 'relu'))
    backend.add(Dense(num_classes))
    backend.compile(loss=keras.losses.categorical_crossentropy,
                  optimizer='adam',
                  metrics=['categorical_accuracy', 'accuracy'])
    return backend
def standardize_matrix_size(data,
                            num_classes):
    longest_size = 0
    for video in data:
        if video.shape[0] > longest_size:
            longest_size = video.shape[0]
    for video in range(len(data)):
        padded_matrix = np.zeros((longest_size, num_classes))
        padded_matrix[:data[video].shape[0],
                    :data[video].shape[1]] = data[video]
        data[video] = padded_matrix
    return data
```

(ZOU (2019))

```
def plot_prediction_TEST(idx,
                         images,
                         labels,
                         mc_predictions,
                         class_labels,
                         num_classes):
    labels = labels.to_numpy()
    p0 = np.array([p[idx] for p in mc_predictions])
    p0_avg = p0.mean(axis=0)
    fig, axes = plt.subplots(1, 2, figsize=(30,12))
    ###
    df = pd.DataFrame(p0_avg, columns = [’col1’])
    df[’col2’] = np.arange(num_classes)
    df = df.sort_values(by = ’col1’, ascending = False)
    df = df.reset_index(drop = True)
    contenders = df[’col1’][df[’col1’]>0.05].count()
    ###
    # second plot
    sns.barplot(orient = ’h’)
    for dist in p0:
        sns.barplot(y =np.arange(contenders),
                    x = dist[df[’col2’]][0:contenders],
                    alpha = 0.1,
                    ax = axes[1], orient = ’h’)
    axes[1].set_xlim([0,1])
    # third plot
    sns.barplot(y = np.arange(contenders),
                x = p0_avg[df[’col2’]][0:contenders],
                ax = axes[1],
                linewidth=2.5, facecolor=(1, 1, 1, 0),
                edgecolor=".2", orient = ’h’)
    new_labels = []
    for i in range(contenders):
        new_labels.append(class_labels[df[’col2’][i]])
    return p0, p0_avg, df
def get_image_distribution(idx,
                           mc_predictions,
                           num_classes,
                           class_labels):
    p0 = np.array([p[idx] for p in mc_predictions])
    fig, axes = plt.subplots(1, 1, figsize=(12,12))
    for i in range(num_classes):
        sns.distplot(p0[:,i], kde = False,
                hist_kws=dict(alpha=0.7))
    axes.set_xlim = [0,1]
    axes.set(xlabel=’Probability’, ylabel=’Count’)
    plt.legend(class_labels)
def get_image_breakdown(idx,
                        test_set_images,
                        test_set_labels,
                        mc_predictions,
                        class_labels, cutoff, stdev):
```

```
    p0 = np.array([p[idx] for p in mc_predictions])
    test_set_labels = test_set_labels.to_numpy()
    print("posterior mean: {}".format(
                p0.mean(axis=0).argmax()) + ’, ’ + class_labels[p0.mean(axis=0).argma
    print("true label: {}"
        .format(test_set_labels[idx].argmax())+ ’, ’ + class_labels[test_set_labels[i
    print()
```

(ZOU (2019))

```
def plot_prediction(idx,
                    images,
                    labels,
                    mc_predictions,
                    class_labels,
                    num_classes):
    labels = labels.to_numpy()
    p0 = np.array([p[idx] for p in mc_predictions])
    p0_avg = p0.mean(axis=0)
    # ###
    # df = pd.DataFrame(p0_avg, columns = [’col1’])
    # df[’col2’] = np.arange(num_classes)
    # df[’col3’] = p0.std(axis = 0)
    # df = df.sort_values(by = ’col1’, ascending = False)
    # df = df.reset_index(drop = True)
    # contenders = df[’col1’][df[’col3’]>0.01].count()
    # ###
    plt.figure(figsize = (12, 12))
    # second plot
    sns.barplot(orient = ’h’)
    for dist in p0:
        sns.barplot(y =np.arange(num_classes), x = dist,
                    alpha = 0.1, orient = ’h’)
    plt.xlim([0,1])
    #plt.title(’Posterior Samples’)
    # third plot
    plt.ylabel(’Class’, fontsize = 22)
    plt.xlabel(’Probability’, fontsize = 22)
    plt.yticks(ticks = None, labels = None, color = ’w’)
    plt.xticks(fontsize = 22)
    sns.barplot(y = np.arange(num_classes), x = p0_avg,
                linewidth=2.5, facecolor=(1, 1, 1, 0),
                edgecolor=".2", orient = ’h’)
def plot_epoch_accuracy(history):
    plt.plot(history.history[’accuracy’])
    plt.plot(history.history[’val_accuracy’])
    plt.title(’Model Accuracy’)
    plt.ylabel(’accuracy’)
    plt.xlabel(’epoch’)
    plt.legend([’train’, ’test’], loc=’upper left’)
    plt.show()
def plot_epoch_loss(history):
    plt.plot(history.history[’loss’])
    plt.plot(history.history[’val_loss’])
    plt.title(’Model Loss’)
    plt.ylabel(’loss’)
    plt.xlabel(’epoch’)
    plt.legend([’train’, ’test’], loc=’upper left’)
    plt.show()
def model_to_png(model, filename):
    plot_model(model, to_file=filename, show_shapes=True,
                show_layer_names=True)
def get_flagged_images_and_videos(mc_preds, belongs_to_video,
                                    cutoff, stdev):
    flagged_image_indices = []
    flagged_image_booleans = []
    flagged_video_indices = []
    not_flagged_images = []
    not_flagged_videos = []
    for idx in range(mc_preds[0].shape[0]):
        px = np.array([p[idx] for p in mc_preds])
        flagged_image_booleans.append(get_flag1(px,
            cutoff, stdev))
        if get_flag1(px, cutoff, stdev) == True:
            flagged_image_indices.append(idx)
        else:
            not_flagged_images.append(idx)
    for idx in range(len(flagged_image_booleans)):
        if flagged_image_booleans[idx] == True:
            flagged_video_indices
                .append(belongs_to_video[idx])
    flagged_video_indices = list(set(flagged_video_indices))
    for idx in range(len(np.unique(belongs_to_video))):
        if idx not in flagged_video_indices:
            not_flagged_videos.append(idx)
    return flagged_image_indices, flagged_video_indices,
                    not_flagged_images, not_flagged_videos
```

```
def get_baseline_nonflags(frames,
                       thresh,
                       non_mc_model,
                       belongs_to_video):
    baseline_not_flagged_images = []
    baseline_not_flagged_videos = []
    baseline_booleans = []
    for idx in tqdm(range(frames.shape[0])):
        probabilities = non_mc_model.predict(
            frames[idx].reshape((1,) + frames[idx].shape))
        prediction = np.argmax(probabilities, axis=1)
        if np.max(probabilities) < thresh[int(prediction)]:
            baseline_booleans.append(True)
        else:
            baseline_booleans.append(False)
            baseline_not_flagged_images.append(idx)
    for idx in range(len(baseline_booleans)):
        if baseline_booleans[idx] == False:
            baseline_not_flagged_videos
                .append(belongs_to_video[idx])
    baseline_not_flagged_videos = list(set(
        baseline_not_flagged_videos))
    return baseline_not_flagged_images,
            baseline_not_flagged_videos
import matplotlib.pyplot as plt
import numpy as np
from Functionsss import *
class_labels = get_class_labels()
dropout = 0.5
num_classes = 101
input_shape = (224, 224, 3)
frontend = build_frontend(num_classes = num_classes,
                            dropout = dropout,
                            input_shape = input_shape,
                            mc = True)
frontend.load_weights('frontend_weights_tuned.hdf5')
non_mc_model = build_frontend(num_classes = num_classes,
                                dropout = dropout,
                                input_shape = input_shape,
                                mc = False)
non_mc_model.load_weights('frontend_weights_tuned.hdf5')
backend = build_backend(num_classes)
backend.load_weights('backend_weights.hdf5')
```

```
video_names = get_backend_data_setup2(
        file_name = 'testlist01.txt')
output, output_labels, frames, frame_labels,
                belongs_to_video,
                mc_preds = get_back_input2(video_names,
                frontend,  num_classes,
                n_iter = 100, batch_size = 50,
                mc_preds = True) s
output_labels = pd.get_dummies(output_labels)
output = np.array(output)
baseline_preds = non_mc_model.predict(frames,
                batch_size = 50)
belongs_to_video_array = np.asarray(belongs_to_video)
baseline_output = [baseline_preds[belongs_to_video_array==k]
                for k in np.unique(belongs_to_video)]
baseline_output = standardize_matrix_size(
                baseline_output, num_classes)
baseline_output = np.array(baseline_output)
                .reshape(output.shape)
mc_ensemble_pred = np.array(mc_preds).mean(axis=0)
                .argmax(axis=1)
ensemble_acc = accuracy_score(
                pd.DataFrame.to_numpy(frame_labels)
                .argmax(axis=1),
                mc_ensemble_pred)
bayes_eval1 = frontend.evaluate(frames, frame_labels)
base_eval1 = non_mc_model.evaluate(frames, frame_labels)
bayes_back_eval1 = backend.evaluate(output, output_labels)
base_back_eval1 = backend.evaluate(baseline_output,
            output_labels)
from tensorflow.keras.preprocessing.image import
        ImageDataGenerator, array_to_img, img_to_array,
        load_img
from sklearn.metrics import classification_report,
                accuracy_score, confusion_matrix
from PIL import Image
```

(Cerliani (2020))

```
datagen = ImageDataGenerator(
        rotation_range=20,
        width_shift_range=0.1,
        height_shift_range=0.1,
        shear_range=0.1,
        zoom_range=0.1,
        horizontal_flip=True,
        fill_mode=’nearest’)
reps = 10
# create augmented images by class
for c in tqdm(range(num_classes)):
    # make folder for augmented images for that class
    createFolder(’./augmented/’ + str(c) + ’/’)
    # load image and reshape it for datagen
    x = frames[frame_labels.iloc[:,c]==1]
    #x = x.reshape((1,) + x.shape)
    # augment it 100 times
    i = 0
    for batch in datagen.flow(x, batch_size = 1,
                              save_to_dir =
                                ’./augmented/’ + str(c),
                              save_prefix = "",
                              save_format = ’jpeg’):
        i += 1
        if i == reps*x.shape[0]:
            break
diz_prob = {}
diz_prob_correct = {}
diz_prob_mistake = {}
for i in range(num_classes):
    diz_prob[i] = []
    diz_prob_correct[i] = []
    diz_prob_mistake[i] = []
for c in tqdm(range(num_classes)):
    count = 0 # how many images are in the class
    augmented_images = []
    for filename in glob(’./augmented/’+ str(c) + ’/*.jpeg’):
        im = Image.open(filename)
        x = img_to_array(im)
        augmented_images.append(x/255.)
        count += 1
    augmented_images = np.reshape(augmented_images,
                (count, 224, 224, 3))
    true_labels = [c] * count
    pred_prob = non_mc_model.predict(augmented_images)
    pred = np.argmax(pred_prob, axis=1)
    for j, prob in zip(pred, pred_prob):
        diz_prob[c].append(prob[j])
        diz_prob
        if c == j:
            diz_prob_correct[j].append(prob[c])
        else:
        diz_prob_mistake
            diz_prob_mistake[j].append(prob[j])
cutoff = 0.6
thresh = {}
for i in range(len(class_labels)):
    thresh[i] = np.quantile(diz_prob[i], cutoff)
for c in range(num_classes):
    plt.hist(diz_prob_correct[c], alpha=0.3,
        label='correct')
    plt.hist(diz_prob_mistake[c], alpha=0.3,
        label='mistake')
    plt.axvline(thresh[c], color='red',
        linestyle='--')
    plt.legend(); plt.xlabel('probability');
    plt.ylabel('count');
    plt.title(class_labels[c])
    plt.show()
baseline_not_flagged_images,
    baseline_not_flagged_videos =
    get_baseline_nonflags(frames,
    thresh, non_mc_model, belongs_to_video)
discrepancies = []
for i in tqdm(range(len(flagged_images))):
    idx = flagged_images[i]
    if idx in baseline_not_flagged_images:
        base_pred = non_mc_model.predict(
            frames[idx].reshape((1,)+frames[idx].shape))
        if frame_labels.iloc[idx][base_pred.argmax()] == 0:
            # then I'm interested in it
            discrepancies.append(idx)
idx = discrepancies[1088]
plt.imshow(frames[idx][:,:,0])
plot_prediction(idx, frames, frame_labels, mc_preds,
    class_labels, num_classes)
get_image_breakdown(idx, frames, frame_labels,
    mc_preds, class_labels, 0.6, 0.4)
p0, p0_avg, df = plot_prediction_TEST(idx,
    frames, frame_labels, mc_preds, class_labels, num_classes)
non_mc_model.predict(frames[idx]
        .reshape((1,)+ frames[idx].shape)).argmax()
check_for_model_misunderstanding(frontend,
    num_classes, class_labels, num_random_images = 1000)
baseline_check_for_model_misunderstanding(
        non_mc_model, num_classes, class_labels,
        thresh, num_random_images = 1000)
bayes_eval2 = frontend.evaluate(
    frames[not_flagged_images],
    pd.DataFrame(frame_labels).iloc[not_flagged_images])
bayes_back_eval2 = backend.evaluate(
    output[not_flagged_videos],
    pd.DataFrame(output_labels).iloc[not_flagged_videos])
base_eval2 = non_mc_model.evaluate(
    frames[baseline_not_flagged_images],
    pd.DataFrame(
    frame_labels).iloc[baseline_not_flagged_images])
del baseline_not_flagged_videos[-1]
base_back_eval2 = backend.evaluate(
    baseline_output[baseline_not_flagged_videos],
    pd.DataFrame(
        output_labels)
        .iloc[baseline_not_flagged_videos])
```

Bibliography

Ambaum, M. H. P. (2012), 'Frequentist vs bayesian statistics - a non-statisticians view'.

Bihl, T. and Talbert, M. (2020), Analytics for autonomous C4ISR within e-government: a research agenda, *in* '53rd Hawaii International Conference on System Sciences, HICSS 2020, Maui, Hawaii, USA, January 7-10, 2020', ScholarSpace, pp. 1–10.
URL: *http://hdl.handle.net/10125/64012*

Buntine, W. L. and Weigend, A. (1991), 'Bayesian back-propagation', *Complex Syst.* **5**.

Cerliani, M. (2020), 'When your neural net doesn't know: a bayesian approach with keras'. Accessed on 09 Feb 2021.
URL: *https://towardsdatascience.com/when-your-neural-net-doesnt-know-a-bayesian-approach-with-keras-4782c0818624*

Chollet, F. et al. (2015), 'Keras'. Accessed on 09 Feb 2021.
URL: *https://github.com/fchollet/keras*

Denker, J. S. and LeCun, Y. (1990), Transforming neural-net output levels to probability distributions, *in* 'Proceedings of the 3rd International Conference on Neural Information Processing Systems', NIPS'90, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, p. 853–859.

Denker, J., Schwartz, D., Wittner, B., Solla, S., Howard, R., Jackel, L. and Hopfield, J. (1987), 'Large automatic learning, rule extraction, and generalization', *Complex Syst.* **1**.

Fukushima, K. (1980), 'Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position', *Biological Cybernetics* **36**, 193–202.

Gal, Y. (2016), Uncertainty in Deep Learning, PhD thesis, University of Cambridge.

Gal, Y. and Ghahramani, Z. (2016), 'Dropout as a bayesian approximation: Representing model uncertainty in deep learning'.

Géron, A. (2019), *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, O'Reilly Media.
URL: *https://books.google.com/books?id=HnetDwAAQBAJ*

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. (2012), 'Improving neural networks by preventing co-adaptation of feature detectors', *CoRR* **abs/1207.0580**. Accessed on 09 Feb 2021.
URL: *http://arxiv.org/abs/1207.0580*

Hochreiter, S. and Schmidhuber, J. (1997), 'Long short-term memory', *Neural Computation* **9**(8), 1735–1780. Accessed on 09 Feb 2021.

Lecun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998), Gradient-based learning applied to document recognition, *in* 'Proceedings of the IEEE', pp. 2278–2324.

Mcculloch, W. and Pitts, W. (1943), 'A logical calculus of ideas immanent in nervous activity', *Bulletin of Mathematical Biophysics* **5**, 127–147.

Neal, R. M. (1995), Bayesian Learning for Neural Networks, PhD thesis, CAN. AAINN02676.

Neal, R. M. (2012), *Bayesian learning for neural networks*, Vol. 118, Springer Science & Business Media.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. and Duchesnay, E. (2011), 'Scikit-learn: Machine learning in Python', *Journal of Machine Learning Research* **12**, 2825–2830.

Ruder, S. (2016), 'An overview of gradient descent optimization algorithms', *CoRR* **abs/1609.04747**. Accessed on 09 Feb 2021.
URL: *http://arxiv.org/abs/1609.04747*

Shridhar, K., Laumann, F. and Liwicki, M. (2019), 'A comprehensive guide to bayesian convolutional neural network with variational inference'.

Simonyan, K. and Zisserman, A. (2014), 'Very deep convolutional networks for large-scale image recognition'.

Soomro, K., Zamir, A. R. and Shah, M. (2012), 'UCF101: A dataset of 101 human actions classes from videos in the wild', *CoRR* **abs/1212.0402**. Accessed on 09 Feb 2021.
URL: *http://arxiv.org/abs/1212.0402*

Specht, D. F. (1990), 'Probabilistic neural networks', *Neural Networks* **3**(1), 109–118.
URL: *https://www.sciencedirect.com/science/article/pii/089360809090049Q*

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. (2014), 'Dropout: a simple way to prevent neural networks from overfitting.', *Journal of Machine Learning Research* **15**(1), 1929–1958. Accessed on 09 Feb 2021.
URL: *http://www.cs.toronto.edu/ rsalakhu/papers/srivastava14a.pdf*

Sterbak, T. (2020), ‘Model uncertainty in deep learning with monte carlo dropout in keras’. Accessed on 09 Feb 2021.
URL: *https://www.depends-on-the-definition.com/model-uncertainty-in-deep-learning-with-monte-carlo-dropout/load-the-mnist-data*

Tishby, Levin and Solla (1989), Consistent inference of probabilities in layered networks: predictions and generalizations, *in* ‘International 1989 Joint Conference on Neural Networks’, pp. 403–409 vol.2.

Werbos, P. J. (1974), Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences, PhD thesis, Harvard University.

ZOU, L. (2019), ‘Bayesian cnn model on mnist data using tensorflow-probability (compared to cnn)’. Accessed on 09 Feb 2021.
URL: *https://medium.com/python-experiments/bayesian-cnn-model-on-mnist-data-using-tensorflow-probability-compared-to-cnn-82d56a298f45*

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*
25–03–2021

2. REPORT TYPE
Master's Thesis

3. DATES COVERED *(From — To)*
August 2019 — March 2021

4. TITLE AND SUBTITLE
Bayesian Augmentation of Convolutional Neural Network - Long Short Term Memory for Video Classification with Uncertainty Measures

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Swize, Emmie K., 2d Lt, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/ENS)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT-ENS-MS-21-M-186

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
Dr. Trevor Bihl
1864 4th St
Wright-Patterson AFB, OH 45433
trevor.bihl.2@us.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Distribution Statement A: Approval for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Success of Department of Defense (DoD) missions rely heavily on intelligence, surveillance, and reconnaissance (ISR) capabilities, which supply information about the activities and resources of an enemy or adversary. To secure this information, satellites and unmanned aircraft systems collect video data to be classified by either humans or machine learning networks. Traditional automated video classification methods lack measures of uncertainty, meaning the network is unable to identify those cases in which it predictions are made with significant uncertainty. This leads to misclassification, as the traditional network classifies each observation with same amount of certainty, no matter what the observation is. Bayesian neural networks offer a remedy to this issue by leveraging Bayesian inference to construct uncertainty measures for each prediction. Because exact Bayesian inference is typically intractable due to the large number of parameters in a neural network, Bayesian inference is approximated by utilizing dropout in a convolutional neural network.

15. SUBJECT TERMS
machine learning, neural network, Bayesian inference, video classification

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Dr. Lance E. Champagne, AFIT/ENS
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
U	U	U	UU	77	9372553636 x 4646; lance.champagne@afit.edu