VIDEO ANALYSIS BY DEEP LEARNING

by

Mona Ramadan

BS in Electronic Engineering, Sebha University, 2005

MS in Electrical Engineering, University of Pittsburgh, 2010

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2019

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This dissertation was presented

by

Mona Ramadan

It was defended on

November 14, 2018

and approved by

Amro A. El-Jaroudi, PhD, Associate Professor
Department of Electrical and Computer Engineering

Ervin Sejdic, PhD, Associate Professor
Department of Electrical and Computer Engineering

Zhi-Hong Mao, PhD, Professor
Department of Electrical and Computer Engineering

Murat Akcakaya, Assistant Professor
Department of Electrical and Computer Engineering

Patrick Loughlin, PhD, Professor, Department of Bioengineering

Dissertation Director: Amro A. El-Jaroudi, PhD, Associate Professor
Department of Electrical and Computer Engineering

VIDEO ANALYSIS BY DEEP LEARNING

Mona Ramadan, PhD

University of Pittsburgh, 2019

The tasks of automatically classifying the content of videos or predicting the outcome of a series of events occurring in a sequence of frames, while may sound simple, are still very challenging research areas despite the vast improvement in computing hardware and the easy access to large sets of data. In our work, we extend machine learning techniques to comprehend videos by tackling three challenging tasks: video classification on the full-length video level, video classification both on the level of actions performed in certain frames and the full-length video level, and action prediction of upcoming events. Classification on the video level is a classic machine learning problem that has been addressed previously. We address this problem both using a standard deep learning approach, where a deep convolutional neural network (CNN) is trained on video frames then a Long Short Term Memory (LSTM) network is used to aggregate the features learned by the CNN into a single video label. And we introduce a different approach that uses still images of a data set that is independent on the video data set to train a CNN that is later used to either classify a selection of video frames and make a conclusion about the video class assuming that the temporal content of the video is insignificant, or to extract the spatial features of video frames and use those features to train an LSTM to incorporate the temporal content.

Our approach results in a classification accuracy of 73% when classifying test videos based on 20 randomly selected video frames, and an accuracy that ranges between 91% and 94% when using an LSTM on top of the CNN while processing only 10 and 300 video frames, respectively, of the test videos on a subset of the YouTube Sports-1M data set. On the other hand training the CNN on video frames then using an LSTM results in an accuracy that

ranges between 96% and 97% when processing 10 and 300 video frames of the test videos. While our proposed LSTM trained on features extracted by a CNN trained on independent images did not outperform the LSTM trained on video frames-trained CNN features, it is important to note that training the CNN on independent images did result in a comparable classification accuracy, which means that the concept could still be used especially if the training video data set does not provide enough images to train a CNN.

Classification on the actions level and the video class level is not a well-addressed problem. We tackle the challenge by using a hybrid CNN-Hidden Markov Model (HMM) system where a dictionary of actions is constructed from the training data and is used to detect a sequence of video actions then map this actions sequence into a video class for the entire video. Our approach detects the actions in videos of the Actions for Cooking Eggs (ACE) data set with an accuracy of 79% while classifying the videos with a 100% accuracy. Finally, we address the problem of next action prediction by using the same hybrid CNN-HMM system to predict the next performed action when only part of the video is available. Our approach successfully predicts the next first and second performed actions in a video stream with a probability higher than 50% when 60% or more of the video is available for processing, with the prediction accuracy continuing to increase as the system gains access to more video frames.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

1.1 MOTIVATION

The rapid advancement of technology in the last fifty years has given ordinary people access to a tremendous amount of digital images and videos. However, the task of understanding the general content of images and videos, as simple as it sounds, still remains a very challenging problem. Research groups have been participating in actual challenges for large scale image classification. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [1] and the Microsoft Common Objects in Context challenge (COCO) [2] are very common competitions introduced in 2010 and 2014 respectively, that not only proposed large scale image data sets, but also announced contests allowing the research community to compete in solving the image classification, localization, detection and segmentation problems. The ILSVRC in particular has attracted many very sophisticated research teams from all over the world; and techniques published based on competition submissions are forming the state of the art in the image classification field. In 2012, the challenge gained special popularity when the winning team managed to reduce the classification error dramatically by using a deep Convolutional Neural Network (CNN) architecture [3]. From that point on, almost every winning team has involved one form or the other of CNNs.

Although image comprehension using deep networks is very popular and is getting accomplished with higher accuracy over the years, much more work and effort is still being put into the video recognition task and yet the results are not comparable to those published for images. Video comprehension is more broad, from the classification of scenes to object tracking, to action recognition, to action prediction; each one of those tasks is difficult on its own and further requires a different data set to examine. There is no benchmark method for

solving each problem, and there isn't even a benchmark large scale data set for it. Different data sets serving different actions and different video analysis problems have been created over recent years. From kinematics activities to assisted daily living activities to activities in sports and Hollywood movies; many data sets are available and a lot of work has been done on each set [4].

Motivated by the high accuracy results of CNN-based image classification models, and by the availability of large video data sets we study the application of deep CNN architectures to classification and action identification and prediction in videos on two different sets of data, namely, the Sports-1M [5] and the ACE [6] data sets.

1.2 CONVENTIONAL VIDEO CLASSIFICATION

A video is an assortment of a number of images, where each image is called a frame. Each video frame is a digital image that is represented by a matrix of values. A single element in an image matrix is known as a picture element, or pixel . The succession of still images displayed at a sufficiently high rate creates the illusion of motion in a video. A video clip is a short video that is composed of one or more video scenes; where a scene consists of a group of frames taken within a single camera action [7]. Classifying entertainment videos, such as assigning a movie genre to a video clip or a sport's type to sports movies is the most popular domain for classification. Videos are usually recorded such that both image and audio information is captured. Many videos even have text information. The text could be inherently embedded in the images themselves, like a player's name printed on a jersey. Or it could be added to the video, like stamping sports games scores on the screen. Conventional video classification techniques either use text, audio, or image features. However, most video classification methods are based on visual video features, either used alone, or in combination with text or audio features [8].

Many standard machine learning classifiers have been used to classify videos. Classifiers based on support vector machines (SVM), neural networks, Gaussian mixture models (GMM) and hidden Markov models (HMM) are commonly used [8]. However, the HMM classifier

is the most widely used due to its ability to capture the temporal relationship inherently existing in video data [9, 10, 11, 12, 13, 14, 15]. The problem with conventional video classification approaches, however, is not only choosing the classification technique; it is rather more of choosing the features that are fed to the classifier.

The literature is full of many feature bases used over the years to classify videos. In [16], the discrete cosine transform (DCT) of 64×64 grayscale images sampled from the original colored videos is taken. Then principal component analysis (PCA) is used to find the DCT coefficients with the highest variance, and use those as input to the classification model that classifies meeting and presentation videos. Wavelet coefficients are also used to obtain features. In [17], the spatio-temporal dynamic activity is used to classify violent scenes. The dynamic activity is found by first applying a 2-D wavelet transform to all the frames of a video shot to obtain a temporal intensity map, then 1-D wavelet transform is applied to the intensity map to obtain the motion dynamic activity of the shot. Color histograms have also been used as features. In [18] the hue saturation value (HSV) color histogram of 3×3 regions of the I-frames from MPEG videos is taken and used along with other features such as texture, edge direction and camera motion to classify videos content as face, people, indoor, outdoor and cityscape. Motion features are also very common especially in classifying action videos. Since different sports represent different motions, [12] used motion features along with color features to classify sports videos into four types: ice hockey, basketball, football, and soccer.

A combination of many other features, such as, texture, shot length, shot transition, pixel brightness and saturation, pixel luminance, camera movement and face frame ratio [19, 20, 21, 22] has also been used; all in an attempt to feed the classification algorithm the best possible engineered features so that it results in the best possible classification performance. Feature engineering has been the main focus of the research community, until recent years when deep learning was introduced [23]. Deep learning techniques on image recognition problems not only showed better performance, but also spared researchers the burden of carefully crafting feature detectors.

The deep learning revolution that transformed the artificial intelligence field has inspired considerable research in the domain of video classification [24, 25, 26, 27, 28, 29]. We,

inspired by the promise of deep learning; high performance with minimal feature engineering, also investigate multiple deep learning techniques and different network architectures in an attempt to tackle the video comprehension problem.

1.3 DIRECTION AND GOALS

Our prime target is to develop a deep learning based architecture that effectively classifies, detects actions and predicts actions of next frames in video signals. We build a dictionary of motions for different video categories, then classify those videos based on the built dictionary. To tackle our problem, we address the following elements:

- Set the classification problem by identifying the video classes and selecting the training and testing data sets.
- Develop a video classification process based on an image classification system.
- Develop a system based on a deep network architecture that identifies and detects video actions then classify unprocessed videos based on the extracted motion features.
- Establish a system that uses a sequence of detected video actions to predict the next performed actions.

1.4 DISSERTATION ORGANIZATION

The remainder of this dissertation is organized as follows. Chapter 2 provides an introduction to deep learning and explains convolutional neural networks in details. Chapter 3 demonstrates previous work on video classification using deep learning then presents our work on video classification using convolutional neural networks. Chapter 4 introduces long short term memory networks and presents video classification results using them. Chapter 5 introduce a hybrid system that addresses the problem of video classification on the video and frame levels by combining a deep network architecture and a hidden Markov model. And Chapter 6 uses the hybrid deep network and HMM system to tackle the next action prediction problem. Finally Chapter 7 summarizes all the research results and suggests a framework for future research.

2.0 BACKGROUND

2.1 MACHINE LEARNING

Machine learning is a rapidly growing computer science field in which computer programs and algorithms are constructed to improve and excel by experience and training; a very similar concept to the human learning process. Machine learning problems roughly fall into one of two categories depending on the nature of the available training data. The first category is the supervised learning problem (which is more common), where the training data is correctly labeled. The computer is thus guided to learn by good experience from labeled data and is expected to give a correct prediction of the class of some unseen data. The second category is the unsupervised learning problem, where the training data comes unlabeled. The computer then trains to learn the hidden patterns in the training data completely on its own.

Classification is a common supervised learning problem where the machine learning algorithm automatically learns to predict the class of an unseen example based on the labels it learns from past observations. An appropriate machine learning algorithm and a classification rule should be chosen according to the nature of the classification problem and the available training data. There exists a large number of supervised classification techniques. Some are based on artificial intelligence like the logic-based and perceptron-based techniques. And some are based on statistics like Bayesian networks [30]. In the field of image classification, both types of techniques have been used. Images could be classified by a wide range of methods like decision trees, support vector machines, fuzzy measures and artificial neural networks [31]. However, artificial neural networks have been proven to perform very well

when classifying image data [31]. Therefore, through the whole of our work, we focus for the most part on neural network based classification techniques.

2.2 NEURAL NETWORKS

An artificial neural network, or simply a neural netwrok, is a computing system that is inspired by the way the human brain processes information. Like in the human nervous system, the neural network is built by a large number of densely interconnected processing elements called neurons. Where each neuron simply takes a number of real-valued inputs and produces a single real-valued output. The simplest possible neural network is constructed of a single neuron as shown in Figure 1. The neuron takes as input $x_1, x_2, ..., x_m$ and a bias term b_k. Each input is associated with a weight $w_{k1}, w_{k2}, ..., w_{km}$. A linear combination of weighted input signals is then passed through the activation function $\varphi : \Re \to \Re$ which produces the network output,

$$y_k = \varphi(\sum_{j=1}^{m} x_j w_{kj} + b_k) \tag{2.1}$$

The standard activation function is the sigmoid function [32], defined as,

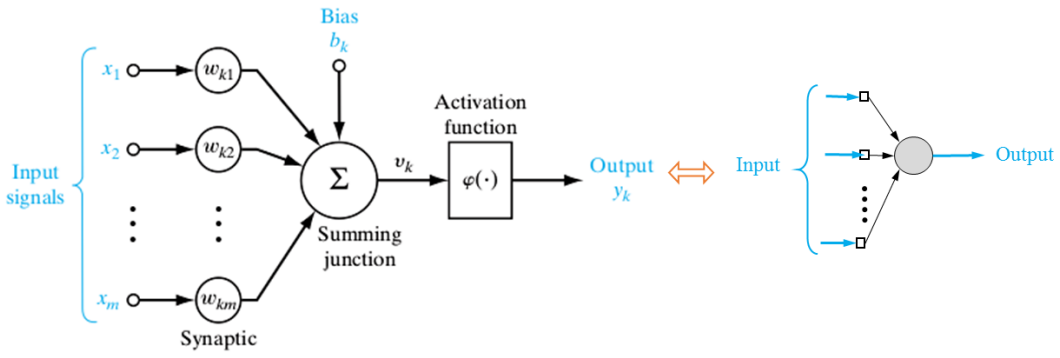


Figure 1: Neuron model

$$\varphi(v_k) = \sigma(v_k) = \frac{1}{1 + \exp(-v_k)} \tag{2.2}$$

The sigmoid function has the nice property that it is not only differentiable, but its derivative is also a product of sigmoid functions.

$$\sigma(v_k)' = \sigma(v_k)\,[1 - \sigma(v_k)] \tag{2.3}$$

However, other functions such as the hyperbolic tangent function and the rectified linear function are also commonly used [33].

A multi-layer neural network is composed by connecting together multiple simple neurons with multiple summing junctions, as shown in Figure 2. The network shown in Figure 2 is called a feedforward neural network where the information moves forward in one direction from the input nodes through the hidden layer to the output nodes.

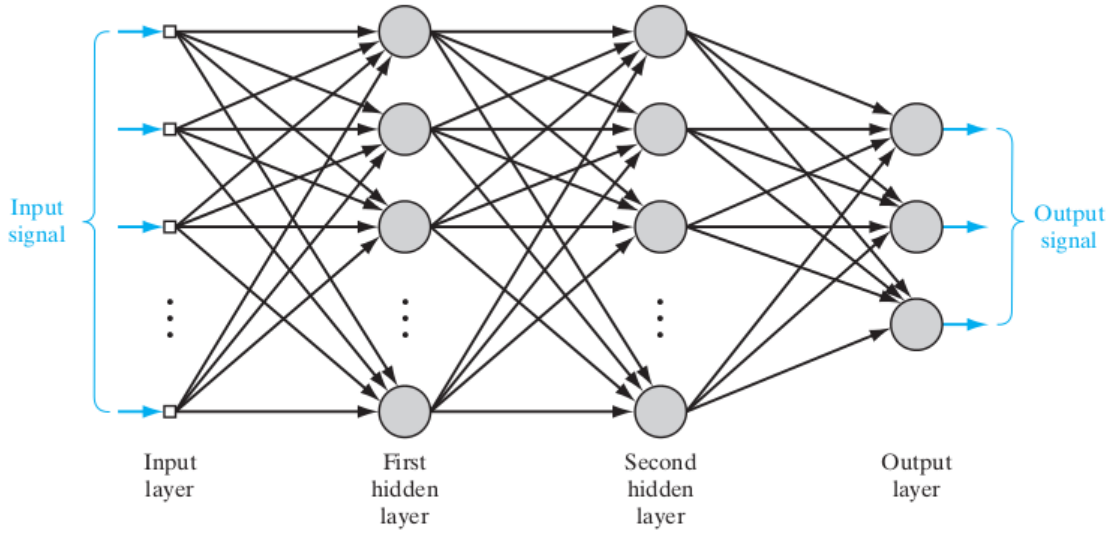


Figure 2: Multi-layer feedforward neural network

The goal of the neural network is to find (learn) a set of weights $\mathbf{W}$ that best align the network's output $\mathbf{y}$ to a desired output $\mathbf{d}$ given an input $\mathbf{x}$, where the bold face is used to indicate a vector representing the network's multiple inputs and outputs. Much like humans, neural networks learn by example. The network is trained using a set of N training examples

$T = \{(\mathbf{x}^{(1)}, \mathbf{d}^{(1)}), ..., (\mathbf{x}^{(N)}, \mathbf{d}^{(N)})\}$. The desired output $\mathbf{d}$ is a $C \times 1$ vector with one element that equals 1 at the index of class c that example $\mathbf{x}$ belongs to, and is 0 everywhere else. The weights are found by minimizing a cost function. There are many ways to define the cost, or error, function. However, the most mathematically convenient way to define it is by half the squared difference between the network's output and the desired output summed over all training example for all C classes,

$$E = \frac{1}{2} \sum_{n=1}^{N} \sum_{k=1}^{C} (y_k^n - d_k^n)^2 = \frac{1}{2} \sum_{n=1}^{N} \|\mathbf{y}^n - \mathbf{d}^n\|_2^2 \tag{2.4}$$

Training a neural network is performed by running two passes. First during the forward pass, the output of every neuron is found by simply applying Equation 2.1. Then during the backward pass, all the weights of all the layers in the network are updated so that the error function is minimized. The backpropagation algorithm seeks to find a local minimum of the error function [34, 35]. At the beginning of training, the network is initialized with randomly chosen weights, then the gradient of the error function is used to correct the initial weights. The backpropagation algorithm computes the gradient of the cost function recursively. To explain backpropagation, let's assume a very simple two-layered single-input single-output neural network with a sigmoid activation function as shown in Figure 3. For simplicity, we assume there are no biases in the network. The backporpagation problem can then be stated as, find w_1, w_2 that minimize the error E, where $E = \frac{1}{2}\|y - d\|^2$

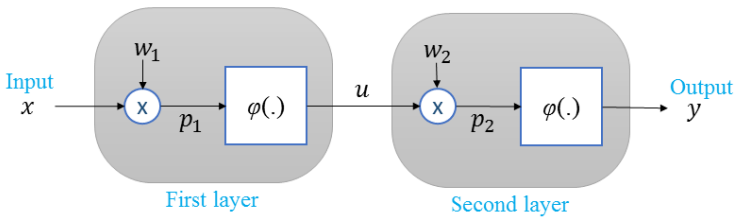


Figure 3: Simple two layer neural network

Gradient descent is used to solve the optimization problem. However, to apply gradient descent we need to find the derivatives $\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}$ which can be found using the chain rule as,

$$\frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial y}\,\frac{\partial y}{\partial w_2} = \frac{\partial E}{\partial y}\,\frac{\partial y}{\partial p_2}\,\frac{\partial p_2}{\partial w_2} \tag{2.5}$$

where , $y = \sigma(p_2)$, $p_2 = uw_2$ and $\sigma(.)$ is the sigmoid function defined as in Equation 2.2. Thus

$$\frac{\partial E}{\partial w_2} = (y - d)\; y(1 - y)\; u \tag{2.6}$$

Similarly, for the first hidden layer,

$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial y}\,\frac{\partial y}{\partial w_1} = \underbrace{\frac{\partial E}{\partial y}\quad\frac{\partial y}{\partial p_2}}_{\text{previously calculated}}\quad\frac{\partial p_2}{\partial w_1} = \frac{\partial E}{\partial y}\,\frac{\partial y}{\partial p_2}\quad\frac{\partial p_2}{\partial u}\,\frac{\partial u}{\partial p_1}\,\frac{\partial p_1}{\partial w_1} \tag{2.7}$$

where $u = \sigma(p_1)$ and $p_1 = xw_1$, thus

$$\frac{\partial E}{\partial w_1} = \underbrace{(y - d)\; y(1 - y)}_{\text{previously calculated}}\; w_2\; u(1 - u)\; x \tag{2.8}$$

The name backproagation comes from the property of using previously calculated terms to calculate derivatives in back layers. If a neural network has 100 layers, the derivative at the input layer will depend only on terms within its vicinity and terms that have been already calculated.

This result can be generalized to any C classes L layers feedforward neural network. At any layer l (the input and output layers are at $l = 1, l = L$ respectively), the output of the layer is found by a forward propagation pass as,

$$\mathbf{x}^l = f\left(\mathbf{u}^l\right), \quad \text{where} \quad \mathbf{u}^l = \mathbf{W}^l\mathbf{x}^{l-1} \tag{2.9}$$

where $\mathbf{W}^l$ is the network weights matrix at layer l and $f(.)$ is the activation function. The loss function E is defined as

$$E(\mathbf{W}) = \frac{1}{2}\sum_{n=1}^{N}\|\mathbf{y}^n - \mathbf{d}^n\|_2^2 = \sum_{n=1}^{N} E_n(\mathbf{W}) \tag{2.10}$$

where $\mathbf{y} = \mathbf{x}^L$, and the error is optimized over the weights and E_n is the error resulting from the n-th training data point.

Stochastic gradient descent is used to find $\mathbf{W}$ that minimizes the cost function $E(\mathbf{W})$. The weights are updated one training example at a time, thus the update rule is,

$$\mathbf{W}^{(\tau+1)} = \mathbf{W}^{(\tau)} + \Delta\mathbf{W}^{(\tau)} \text{ , where } \Delta\mathbf{W}^{(\tau)} = -\eta\nabla E_n(\mathbf{W}^{(\tau)}) \tag{2.11}$$

where $\eta > 0$ is the learning rate and τ is the iteration step. The problem now is to find the derivatives of the error with respect to weights.

The derivative of the error with respect to weights is given by,

$$\nabla E_n(\mathbf{W}) = \frac{\partial E_n}{\partial \mathbf{W}} = \frac{\partial E_n}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{W}} \tag{2.12}$$

where

$$\frac{\partial \mathbf{u}^l}{\partial \mathbf{W}^l} = \frac{\partial \mathbf{W}^l \mathbf{x}^{l-1}}{\partial \mathbf{W}^l} = \left(\mathbf{x}^{l-1}\right)^T \tag{2.13}$$

The derivative of the error with respect to a layer's total input is commonly referred to as delta (or sensitivity) [36], where

$$\boldsymbol{\delta} = \frac{\partial E_n}{\partial \mathbf{u}} \tag{2.14}$$

The sensitivity for the output layer will therefore be defined as,

$$\boldsymbol{\delta}^L = \frac{\partial E_n}{\partial \mathbf{x}^L}\frac{\partial \mathbf{x}^L}{\partial \mathbf{u}^L} = \frac{\partial E_n}{\partial \mathbf{y}^n}\frac{\partial \mathbf{x}^L}{\partial \mathbf{u}^L} = \frac{\partial}{\partial \mathbf{y}^n}\left(\frac{1}{2}\|\mathbf{y}^n - \mathbf{d}^n\|_2^2\right)\frac{\partial f(\mathbf{u}^L)}{\partial \mathbf{u}^L} = (\mathbf{y}^n - \mathbf{d}^n) \circ f'(\mathbf{u}^L) \tag{2.15}$$

where $\circ$ denotes element-wise multiplication. For any other layer l, the sensitivities are backpropagated using the recurrence relation,

$$\boldsymbol{\delta}^l = \left(\mathbf{W}^{l+1}\right)^T \boldsymbol{\delta}^{l+1} \circ f'(\mathbf{u}^l) \tag{2.16}$$

Thus, the delta rule for updating a weight assigned to a given neuron is just a copy of the input to that neuron scaled by the neuron's delta,

$$\frac{\partial E_n}{\partial \mathbf{W}^l} = \mathbf{x}^{l-1}\left(\boldsymbol{\delta}^l\right)^T \tag{2.17}$$

The backpropagation algorithm is thus applied by first finding $\boldsymbol{\delta}^L$ as in Equation 2.15 for the output layer. Then for each following hidden layer, we find $\boldsymbol{\delta}^l$ as in equation 2.16. Finally the weights get updated as in Equation 2.11, where the error gradient is defined as in Equation 2.17. The biases are also updated in a similar manner. It is worth noting that the more

hidden layers the network has, propagating the errors backward will result in the gradient getting smaller and smaller. This is called the vanishing gradients problem, and is one major drawback for training deep neural networks.

After training, the performance of the neural network is measured on a set of unseen examples called the testing set. The objective of training a neural network is for it to learn a set of weights that minimizes the error between the network's output and a desired output averaged over all the training examples in hope that when the network sees a new example, its output is also not far away from the desired one. In other words, a well trained network is a network that minimizes the error not only on the training set, but also on the testing set. When the training error is driven to be very small, while the testing error is large; the network is said to be overfitting on the training set. An overfitting network is a network that does not learn well, but rather memorizes all the training examples so that when presented with a new situation, it fails to generalize. Early stopping is essential to prevent overfitting. Stopping training early could be achieved by monitoring the training and testing (or validation) errors and stopping once the validation error stops improving; Figure 4 illustrates the training and validation errors for overfitting networks. For as much as it is important to avoid overfitting, it is equally important not to underfit. We need to find a balance between the two. Although underfitting is easier to spot and avert, preventing overfitting is a challenge. The most trivial way to avoid overfitting is to train on a sufficiently large training set, however, it is not always possible to increase the training set. Another way to reduce the effect of overfitting is to train multiple neural networks with different settings, then average their predictions. Although this is an expensive approach (both in the training and testing phases), it helps eliminate overfitting since different networks will overfit differently; therefore averaging their results, hopefully, cancels the effect of overfitting [37].

There are many other more refined techniques to prevent a network from overfitting and to improve its overall performance. Some techniques suggest the use of alternative or modified cost functions, some suggest the use of other activation functions instead of the sigmoid non-linearity and some have to do with initializing and updating the weights [38]. The mathematically convenient least squares cost function could be modified by adding $\ell 1$

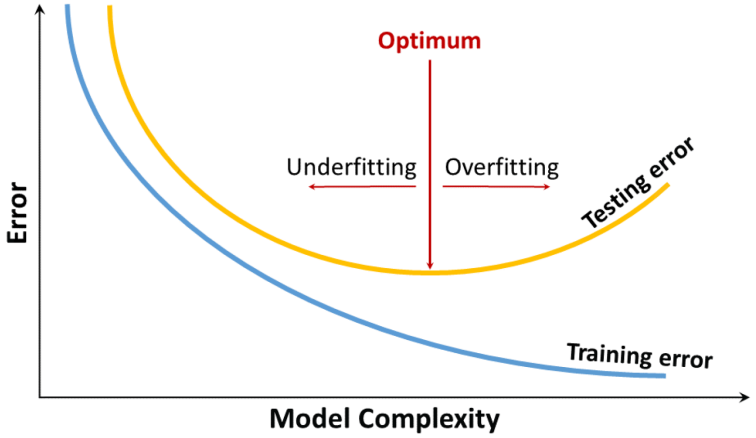


Figure 4: Illustration of overfitting and optimal early stopping

or $\ell 2$ regularization terms. This is also known as weight decay, where a penalty term for weight magnitudes is added to the cost function to drive the gradient descent to search for smaller magnitude weights. The cost function is thus modified as

$$E(\mathbf{W}) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}^n - \mathbf{d}^n\|_2^2 + \lambda \|\mathbf{W}\|_p^p \tag{2.18}$$

where p is 1 or 2 for $\ell 1$ or $\ell 2$ regularization respectively, and the matrix norm is defined as $\|\mathbf{W}\|_p^p = \sum_i \sum_j |w_{ij}|^p$. Other forms of cost functions have proven to perform better and cause the gradient descent to converge faster [39, 40]. Cross entropy for instance is the standard cost function in most recent neural network applications, it is defined as,

$$E = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{C} d_k^n \log y_k^n \tag{2.19}$$

Dropout is another form of regularization that unlike $\ell 1$ and $\ell 2$ regularization, does not modify the cost function but rather modifies the network structure. The basic idea of dropout is to randomly (and temporarily) cancel $p * 100\%$ of the hidden neurons in the network while keeping the input and output neurons untouched [41, 42]. During training, on each presentation of each training example, every hidden neuron is randomly turned off with a probability of p. The neurons that are dropped out do not contribute to the forward

pass or backpropagation, which prevents any neuron in one layer from relying too strongly on a single neuron in the previous layer and thus reduces overfitting. During testing, all the dropout neurons are switched on again and the full network structure is evaluated on the test data. However, the outgoing weights of all the hidden neurons are scaled down by a factor of p to compensate for the fact that $1/p$ as many of them are active. Dropout thus could be seen as a less expensive version of model combination. The probability the a hidden neuron is dropout p is ideally chosen by validation, however, $p = 0.5$ has shown to be a great choice for many applications [42].

The other direction to improve neural networks performance is to modify the way weights are initialized and updated during backpropagation. Althought weights are generally initialized randomly to small values driven from a Gaussian distribution with zero mean and a variance inversely proportional to the number of network inputs [38, 43], different applications may call for different initialization schemes. Nevertheless, the way weights are updated could improve the learning process and the network's performance. The momentum technique [44, 43] offers a small modification to the weights update rule in gradient descent that accelerates learning. In this case a weighted average of the current and previous gradients is used to update the weights. The update rule therefore is modified as,

$$\Delta \mathbf{W}^{(\tau)} = -\eta \nabla E_n(\mathbf{W}^{(\tau)}) + \mu \Delta \mathbf{W}^{(\tau-1)} \tag{2.20}$$

where $0 \leq \mu \leq 1$ is the momentum coefficient. Adding a momentum term to the update rule causes the step size to gradually increase in search regions where the gradient is not changing, which results in speeding the convergence.

Another key point worth clarifying is the choice of all the network's hyper-parameters. Parameters like the learning rate η, the momentum coefficient μ, the dropout percentage p, the number of layers in the network, the regularization coefficient λ, even the decision of whether to use regularization and what type of regularizers to use can be thought of as hyper-parameters. In machine learning, hyper-parameters are chosen through the concept of validation. Validation is applied to a machine learning model in order to assess how well the model generalizes to to an independent data set called the validation set. The hyperparameters are tuned by training different models using different hyperparameters,

then choosing the parameters that yield the best performance on both the training and validation sets. Moreover, validation is another way to prevent overfitting.

2.3 DEEP LEARNING

Neural networks with different number of layers can produce different decision boundaries. A single layered neural network for instant has a hyperplane decision boundary, while networks with three layers can generate arbitrary decision regions. As a matter of fact the deeper the network architecture gets, the more expressive it becomes [43]. On the other hand, training deeper networks using backpropagation becomes inefficient due to the issue of vanishing gradients. Nonetheless, there is an increasing demand for a machine learning tool that is capable of handling complex patterns especially with the growing availability of bigger data. Deep learning is a machine learning tool that promises to learn high level representations and model very complex functions for larger data sets. Furthermore, deep learning vows to learn those very complex patterns on its own from raw data; eliminating the challenging step of feature engineering. It relies on the training process to discover the most useful patterns across the data.

Like conventional machine learning, deep learning techniques could be roughly categorized into supervised and unsupervised (or generative) models. Convolutional neural networks and deep belief networks are the most common models for supervised and unsupervised deep learning algorithms respectively. Since we are concerned with the supervised classification problem, we focus our attention on convolutional neural networks.

2.4 CONVOLUTIONAL NEURAL NETWORKS

A convolutional neural network (CNN) is one type of feedforward neural networks. Unlike the traditional feedforward neural networks, CNNs are not necessarily fully connected which leads to many desired properties. Although traditional neural networks work well for image

recognition, the full connectivity of all their neurons prevents the network from scaling well to higher resolution images. For example the MNIST hand written digits images are 28×28 pixels, which means that a single fully connected neuron in the first hidden layer would have 784 weights, if the images are 100×100 with multiple hidden layers the number of weights that need to be optimized grows immensely. Furthermore, the fully connected architecture treats every pixel of the image exactly the same without taking into consideration the spatial structure of the image. CNN's architecture allows us to take advantage of the spatial structure of images resulting in better inference, faster training and reducing the risk of overfitting.

CNNs are designed to handle multi-dimensional data like colored or gray scaled images. As a matter of fact, CNNs have recently been the leading approach for image classification, segmentation and recognition by the computer vision community [23]. CNNs combine three architectural ideas: local receptive fields, shared weights and pooling [45, 33]. The following subsections explain these concepts in details.

2.4.1 Local receptive fields

In the traditional fully-connected feedforward neural network shown in Figure 2, the inputs are expressed as a vertical line of neurons where every input neuron is connected to every single neuron in the following layer. This fully connected scheme results in two major problems, increasing the model complexity and not scaling well for higher dimensional images. For instance, if we consider the MNIST data set [46], which is a large database of handwritten digits. Each image in the data set is a 28×28 matrix whose values correspond to the 28×28 pixel intensities of the grayscaled image used as input. A single fully connected neuron in the first layer of a traditional feedforward neural network would learn $28 * 28 = 784$ weights. On the other hand, if the input is a colored image of size 256×256, then each neuron needs to learn $256 * 256 * 3 = 196,608$ weights just in the first layer, which leads to an enormous number of parameters that recklessly increases the model complexity resulting in expensive training and overfitting. In contrast, for CNNs the input is represented by a square of neurons, for instance a 28×28 matrix for MNIST data where input pixels are

connected to a layer of hidden neurons, however, not every input pixel will be connected to every hidden neuron. Instead, only small localized regions of the input image get connected to the hidden neurons, as shown in Figure 5, where instead of connecting all 28×28 input neurons only a 7×7 region is connected to one of the hidden neurons. That connected region is called the local receptive field for the hidden neuron and can be thought of as a window that looks on a segment of the input.

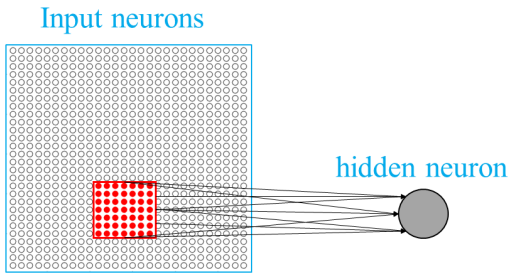


Figure 5: Local receptive field

Each connection in the local receptive field learns a weight and the hidden neuron learns an additional bias. Then the window is slid across the entire input image with an appropriate overlap (stride length) so that each local receptive field is connected to a different hidden neuron [47], as shown in Figure 6. The motivation is that natural images are stationary, meaning that the statistics of one part of the image are the same as any other part [48]. With local receptive fields, neurons can extract elementary visual features such as oriented edges, corners or end-points. Those elementary features are then combined in higher layers to learn higher order features.

2.4.2 Shared weights and biases

Each input neuron within a local receptive field learns a weight and every hidden neuron learns a bias. These weights and bias however are shared between all local receptive fields and their corresponding hidden neurons within a hidden layer. This suggests that the features that we learn at one part of the image can also be applied to other parts of the image, and

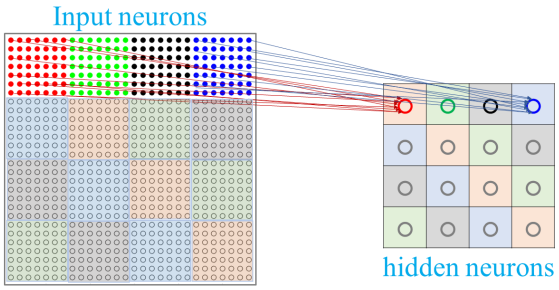


Figure 6: Input neurons connected to the first hidden layer. The local receptive field is slid vertically and horizontally by a stride length of 7.

we can use the same features at all locations. Sharing weights and biases is a very nice property of CNNs, it greatly reduces the number of parameters involved in a network which leads to faster training. The map from the input layer to the hidden layer (the output of the hidden neuron) is called a feature map, and the weights and bias that define the feature map are called shared weights and shared bias respectively. In other words, the input image gets scanned by a single neuron that has a local receptive field, then the states of this neuron get stored at corresponding locations in the feature map. This operation is equivalent to a convolution with a small size kernel followed by an activation function. And hence, the layer is named the convolution layer, the shared weights and bias are named the kernel or filter of the layer and the convolution filter size is the receptive field. A convolutional layer is usually composed of several feature maps with different weights and biases so that different feature maps learn different types of features as shown in Figure 7. Although Figure 7 illustrates the use of 3 feature maps, CNNs usually use many more features maps. For example the early LeNet-5 [47] used 6 feature maps associated with 5×5 local receptive fields to classify the MNIST digits.

2.4.3 Pooling

After learning a set of features using convolution, the features are used for classification. Of course using all the the extracted features is ideal, but it could be computationally

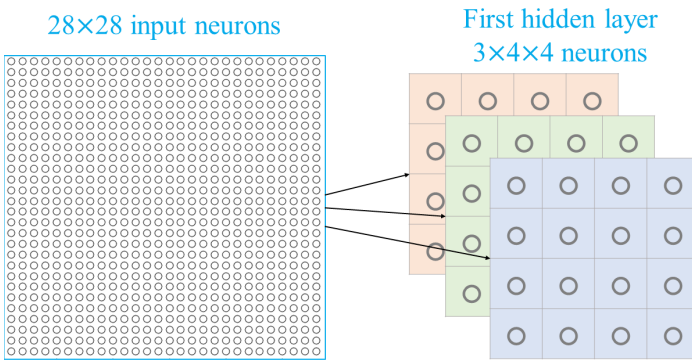


Figure 7: A hidden layer with 3 feature maps, each is defined by a set of 7×7 shared weights and one shared bias. This layer can find 3 different features across the entire image.

challenging. In deep learning architectures, pooling is used after convolution to merge all similar features within one space into one. The pooling layers simplify the information in the output of the convolutional layer in a form of non-linear downsampling. Pooling partitions the output of convolution into a set of non-overlapping blocks (although the blocks could overlap) and outputs a summary of the block. There are many ways to summarize a block, such as taking the average, $\ell 2$ norm, maximum, or a linear combination of the neurons within the block. The most common way for pooling is max-pooling, which takes the maximum of each neuron within the pooled unit. Max-pooling is applied to every feature map separately, as illustrated in Figure 8. Pooling not only reduces the number of learned features, it also helps the learned features to be more invariant to small translations and distortions.

After several convolutional and pooling layers, high level learning takes place using some fully connected layers. A group of fully connected layers is basically just a traditional feedforward neural network that takes as input the features learned and summarized by a set of convolution and pooling layers. Figure 9 shows a complete structure of a CNN, where convolution and pooling layers are followed by a fully connected layer. Every layer within the CNN topology (shown in Figure 10) receive and process data differently. The different processes could be broken down as follows:

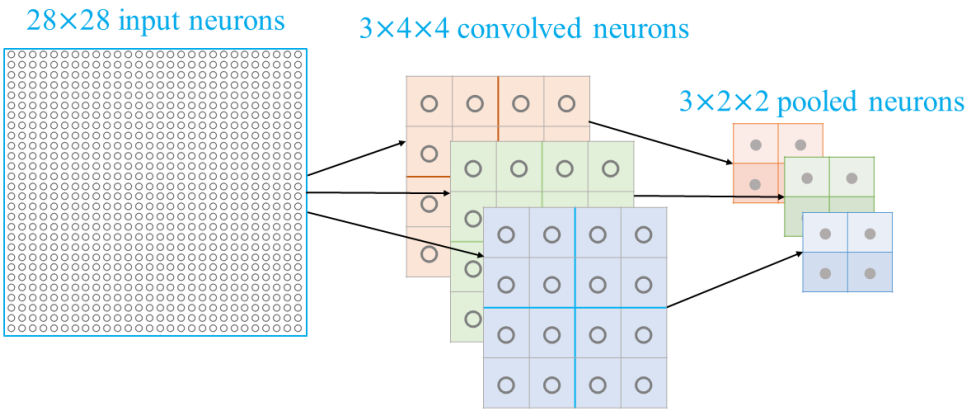


Figure 8: Pooling layer. Every 2×2 block of the hidden neuron in each feature map is summarized (pooled) to a single neuron.

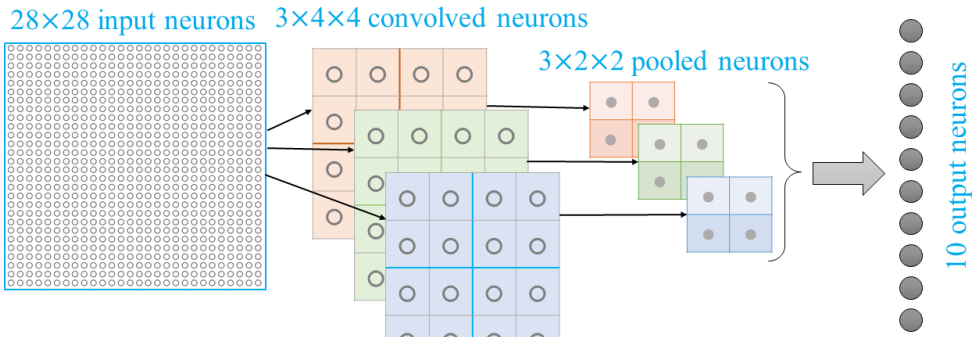


Figure 9: A complete CNN structure. Every single neuron of the last pooled layer is connected to every one of the 10 output neurons that correspond to the 10 possible values of the MNIST digits.

- **Input layer:** an $M \times N \times C$ image matrix, where M is the width, N is the height and C is the number of channels of the image matrix.

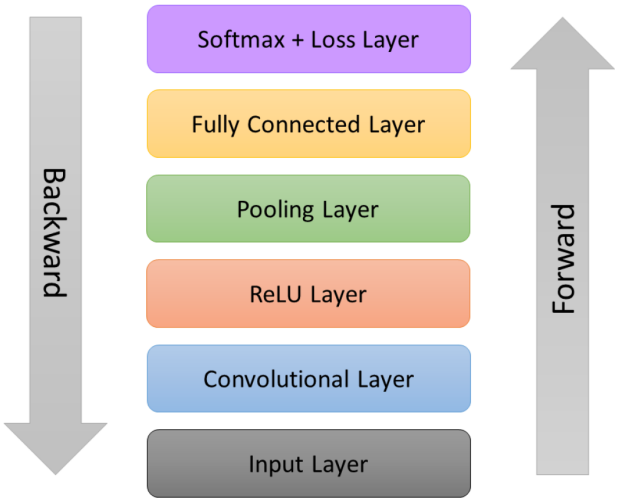


Figure 10: CNN topology with the main building blocks

- **Convolutional layer:** has K filters of size $F \times F$ and uses stride S and zero padding P. It takes an input of size $W_1 \times H_1 \times D_1$, performs the convolution process and outputs a volume of size $W_2 \times H_2 \times D_2$, where [49]

$$\begin{aligned} W_2 &= \left\lfloor \frac{W_1 - F + 2P}{S} \right\rfloor + 1 \\ H_2 &= \left\lfloor \frac{H_1 - F + 2P}{S} \right\rfloor + 1 \\ D_2 &= K \end{aligned} \tag{2.21}$$

  The layer learns $F \times F \times D_1$ weights plus a single bias per filter, with a total parameters of $K \times (F \times F \times D_1 + 1)$.

- **ReLU Layer:** performs a non-linear function on each element of its input. The input and output of the layer are of equal sizes and the layer does not perform any learning. Although the sigmoid function is the traditional non-linear activation function, the rectifier is the most popular non-linearity used in deep learning. The activation function for the rectifier is defined as

$$f(x) = \max(0, x) \tag{2.22}$$

An activation unit that uses a rectifier function is called rectified linear unit (ReLU) [50]. Using ReLU has been shown to result in faster learning [3] due to its non-saturating nature. In gradient descent learning, the weights are updated proportionally to the gradient of the activation function. If sigmoid is used as an activation, the gradient will be very close to 0 (vanishes) for higher activations. Consequently, the weights will stop changing, which results in slowing down the learning or converging to bad local minimum. On the other hand, for ReLU, the derivative will be 1 for any activation higher than 0.

- **Pooling Layer:** has a pooling window of size $F \times F$ and stride S. It performs pooling on an input volume of size $W_1 \times H_1 \times D_1$, and outputs a volume of size $W_2 \times H_2 \times D_2$, where [49] :

$$\begin{aligned} W_2 &= \left\lfloor \frac{W_1 - F}{S} \right\rfloor + 1 \\ H_2 &= \left\lfloor \frac{H_1 - F}{S} \right\rfloor + 1 \\ D_2 &= D_1 \end{aligned} \tag{2.23}$$

  The layer does not learn any new parameters.

- **Fully Connected Layer:** is equivalent to any hidden layer in traditional neural networks. Nonetheless, it is possible to represent fully connected layers as convolutional layers by using a number of filters that equals the desired number of neurons in the fully connected layer. For example, if a fully connected layer takes an input of size $(W_1 = 5) \times (H_1 = 5) \times (D_1 = 512)$ and is supposed to output a layer with 4096 neurons fully connected to each neuron of the input, then the layer could be converted to a convolutional layer with $F = 5, K = 4096, S = 1, P = 0$. A Fully connected layer learns a total of $K \times (F \times F \times D_1 + 1)$ parameters.

- **Softmax and loss layer:** The last layer of a CNN applies softmax followed by a loss function which is usually the cross entropy function defined as in equation 2.19. The softmax ensures the network is assigning normalized probabilities to different classes. A softmax function takes as input a vector $\mathbf{z}$ of size C, where C is the number of classes and outputs a vector of size C that is defined as

$$f(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{C} e^{z_k}}, \quad \text{for} \quad j = 1, ..., C \tag{2.24}$$

2.5 LEARNING IN CNN LAYERS

Although the architecture of the CNN is different than that of the traditional neural network, the overall picture is similar. In both cases the network is constructed by multiple simple units whose behaviors are determined by their weights and biases. And the goal in both architectures is to use the training data to train the network's weights and biases so that the error between the network output and the desired output is minimal. The forward and backward propagation will differ depending on which layer is propagated through.

2.5.1 Convolutional Layer

The convolutional layer is the key building unit in the CNN architecture. Furthermore, it's the only unit that learns new parameters (besides the fully connected unit). At a convolution layer, the previous layer's feature maps are convolved with learnable weights and put through the activation function along with the convolution filter's bias to form the output feature map. Every convolutional layer is followed by a ReLU (and possibly a pooling unit) and the output of layer l is fed as input to layer $l+1$. Let $w^l_{i,j}$, $u^l_{i,j}$ be a weight and output of a convolutional layer at layer l that is passed through an activation unit before it goes into the next layer. The forward pass through the convolutional layer outputs $u^l_{i,j}$ which is defined as,

$$u^l_{i,j} = w^l_{i,j} * x^l_{i,j} + b^l = \sum_{i'} \sum_{j'} w^l_{i',j'}\, x^l_{i-i',j-j'} + b^l \text{ , where } \quad x^l_{i,j} = f(u^{l-1}_{i,j}) \tag{2.25}$$

A convolutional layer l learns a weight matrix $\mathbf{w}^l$ and a bias $\boldsymbol{b}^l$, where learning takes place through backpropagation using gradient descent. To derive the update rule for the weights, we find the derivative of error E at layer l with respect to the filter $\mathbf{w}^l$ which is obtained by the chain rule as [36]:

$$\frac{\partial E}{\partial w^l_{i,j}} = \frac{\partial E}{\partial u^l_{i,j}} \frac{\partial u^l_{i,j}}{\partial w^l_{i,j}} = \sum_{i'} \sum_{j'} \frac{\partial E}{\partial u^l_{i',j'}} \frac{\partial u^l_{i',j'}}{\partial w^l_{i,j}} \tag{2.26}$$

but

$$\begin{aligned}\frac{\partial u^l_{i',j'}}{\partial w^l_{i,j}} &= \frac{\partial}{\partial w^l_{i,j}}\left[\sum_{i''}\sum_{j''} w^l_{i'',j''}\, x^l_{i'-i'',j'-j''} + b^l\right] \\ &= \frac{\partial}{\partial w^l_{i,j}}\left[\sum_{i''}\sum_{j''} w^l_{i'',j''}\, f\left(u^{l-1}_{i'-j'',i'-j''}\right) + b^l\right] \end{aligned} \tag{2.27}$$

However, taking the derivative with respect to $w^l_{i,j}$ will result in non-zero terms only when $i = i''$ and $j = j''$. Therefore the derivative becomes,

$$\begin{aligned}\frac{\partial u^l_{i',j'}}{\partial w^l_{i,j}} &= \frac{\partial}{\partial w^l_{i,j}}\left[w^l_{i,j}\, f\left(u^{l-1}_{i'-i,j'-j}\right) + b^l\right] \\ &= f\left(u^{l-1}_{i'-i,j'-j}\right) \end{aligned} \tag{2.28}$$

Thus,

$$\begin{aligned}\frac{\partial E}{\partial w^l_{i,j}} &= \sum_{i'}\sum_{j'} \frac{\partial E}{\partial u^l_{i',j'}} \frac{\partial u^l_{i',j'}}{\partial w^l_{i,j}} \\ &= \sum_{i'}\sum_{j'} \delta^l_{i',j'} \frac{\partial u^l_{i',j'}}{\partial w^l_{i,j}} \\ &= \sum_{i'}\sum_{j'} \delta^l_{i',j'} f\left(u^{l-1}_{i'-i,j'-j}\right) \\ &= \delta^l_{i,j} * f\left(u^{l-1}_{-i,-j}\right) \\ &= \delta^l_{i,j} * f\left(\mathrm{rot}_{180^\circ}\left[u^{l-1}_{i,j}\right]\right) \end{aligned} \tag{2.29}$$

Where $\mathrm{rot}_{180^\circ}[v]$ is the 180° rotation operation of a vector v, and delta is obtained as

$$\begin{aligned}\frac{\partial E}{\partial u^l_{i,j}} &= \sum_{i'}\sum_{j'} \frac{\partial E}{\partial u^{l+1}_{i',j'}} \frac{\partial u^{l+1}_{i',j'}}{\partial u^l_{i,j}} \\ &= \sum_{i'}\sum_{j'} \delta^{l+1}_{i',j'} \frac{\partial u^{l+1}_{i',j'}}{\partial u^l_{i,j}} \\ &= \sum_{i'}\sum_{j'} \delta^{l+1}_{i',j'} \frac{\partial}{\partial u^l_{i,j}}\left[\sum_{i''}\sum_{j''} w^{l+1}_{i'',j''}\, f\left(u^l_{i'-i'',j'-j''}\right) + b^{l+1}\right] \end{aligned} \tag{2.30}$$

However, taking the derivative with respect to $u^l_{i,j}$ will result in non-zero terms only when $i' - i'' = i$ and $j' - j'' = j$. Therefore the derivative becomes,

$$\begin{aligned}
\frac{\partial E}{\partial u^l_{i,j}} &= \sum_{i'}\sum_{j'} \delta^{l+1}_{i',j'} \frac{\partial}{\partial u^l_{i,j}} \left[w^{l+1}_{i'-i,j'-j} \, f\left(u^l_{i,j}\right) + b^{l+1} \right] \\
&= \sum_{i'}\sum_{j'} \delta^{l+1}_{i',j'} \, w^{l+1}_{i'-i,j'-j} \frac{\partial f\left(u^l_{i,j}\right)}{\partial u^l_{i,j}} \\
&= \sum_{i'}\sum_{j'} \delta^{l+1}_{i',j'} \; w^{l+1}_{i'-i,j'-j} \, f'\left(u^l_{i,j}\right) \\
&= \left[\delta^{l+1}_{i,j} * w^{l+1}_{-i,-j}\right] \; f'\left(u^l_{i,j}\right) \\
\delta^l_{i,j} &= \left[\delta^{l+1}_{i,j} * \mathrm{rot}_{180^\circ}\left(w^{l+1}_{i,j}\right)\right] \circ f'\left(u^l_{i,j}\right)
\end{aligned} \tag{2.31}$$

where $\circ$ implies element-wise multiplication. Equations 2.29,2.31 together define the gradient of the error function with respect to the weights. The weights are then updated as usual using gradient descent [36].

2.5.2 Pooling Layer

Propagation through the max-pooling layers is much simpler since they do not do any learning themselves. They simply take some $k \times k$ region of their $N \times N$ input and output the maximum of that region. Each maximum is stored in one unit of the $\frac{N}{k} \times \frac{N}{k}$ output block. Let x and $g(x)$ be the input and output respectively of a max pooling layer. The derivative is thus given by,

$$\begin{aligned}
g(x) &= \max(x) \\
\frac{\partial g}{\partial x_i} &= \begin{cases} 1 & \text{if} \quad x_i = \max(x) \\ 0 & \text{otherwise} \end{cases}
\end{aligned} \tag{2.32}$$

In other words, only the unit which was the maximum during the forward pass receives the error during backpropagation. For this reason, the error propagated from max pooling layers is sparse.

2.5.3 Fully Connected Layer

Propagation through the fully connected layers is the same as propagation through traditional neural network layers, which was discussed in Section 2.2. However, since a fully connected layer can be thought of as a convolutional layer with the filter size being equal to the layer's input size, then one could apply the same forward and backward pass equations for convolutional layers (Equations 2.25 - 2.31) on fully connected ones.

2.5.4 Softmax and Loss Layer

The forward pass of the softmax and loss layer is straight forward. The output of the last hidden layer u^L is passed to the softmax and log loss functions defined as in Equations 2.24 and 2.19 respectively. During backpropagation, no parameters are learned and only the derivatives are needed. The derivative of the softmax function is:

$$\frac{\partial y_i}{\partial z_j} = \frac{\partial}{\partial z_j}\left(\frac{e^{z_i}}{\sum_{k=1}^{C} e^{z_k}}\right) = \frac{\frac{\partial e^{z_i}}{\partial z_j}\sum_{k=1}^{C} e^{z_k} - e^{z_i}\frac{\partial \sum_{k=1}^{C} e^{z_k}}{\partial z_j}}{\left[\sum_{k=1}^{C} e^{z_k}\right]^2} \quad \text{, where} \quad i, j = 1, ..., C \tag{2.33}$$

which has two cases,

$$\begin{aligned}
\text{if } i = j: \quad \frac{\partial y_i}{\partial z_j} &= \frac{e^{z_i}\sum_{k=1}^{C} e^{z_k} - e^{z_i}e^{z_j}}{\left[\sum_{k=1}^{C} e^{z_k}\right]^2} = \frac{e^{z_i}}{\sum_{k=1}^{C} e^{z_k}} - \frac{e^{z_i}}{\sum_{k=1}^{C} e^{z_k}}\frac{e^{z_j}}{\sum_{k=1}^{C} e^{z_k}} = y_i\left(1 - y_j\right) \\
\text{if } i \neq j: \quad \frac{\partial y_i}{\partial z_j} &= \frac{0 - e^{z_i}e^{z_j}}{\left[\sum_{k=1}^{C} e^{z_k}\right]^2} = -\frac{e^{z_i}}{\sum_{k=1}^{C} e^{z_k}}\frac{e^{z_j}}{\sum_{k=1}^{C} e^{z_k}} = -y_i\, y_j
\end{aligned} \tag{2.34}$$

Therefore, the derivative of the cross entropy loss with respect to the softmax input is,

$$\begin{aligned}
\frac{\partial E}{\partial z_i} &= \frac{\partial}{\partial z_i}\left(-\sum_{k=1}^{C} d_k \log y_k\right) \\
&= -\sum_{k=1}^{C} \frac{d_k}{y_k}\frac{\partial y_k}{\partial z_i} \\
&= \begin{cases} -d_i(1-y_i) & , \quad k = i \\ \sum_{k=1}^{C} d_k\, y_i & , \quad k \neq i \end{cases} \\
&= -d_i + \left(d_i\, y_i + \sum_{k\neq i}^{C} d_k\, y_i\right) \\
&= -d_i + y_i \underbrace{\sum_{k=1}^{C} d_k}_{1} = -d_i + y_i
\end{aligned} \tag{2.35}$$

3.0 CONVOLUTIONAL NEURAL NETWORKS FOR VIDEO CLASSIFICATION

In this chapter, we address the problem of video classification using CNNs. Our problem is simply stated as: given a set of labeled training videos, where each video is assigned a single label for the entire length of the video, find a CNN architecture to classify a set of unlabeled test videos. We approach this problem by training a CNN on an independent data set of labeled images that are related to the video labels, then use that CNN to classify a random selection of video frames. The video is assigned a label by averaging the classification scores of the randomly selected video frames.

3.1 CNN FOR IMAGE CLASSIFICATION

CNNs have been heavily used in recent years in the field of image recognition. As a matter of fact, they have given state of the art results on many tasks like image classification and object detection. Most commonly used CNN architectures are inspired by the AlexNet [3] introduced in 2012 as the winning entry of the classification challenge in the ILSVRC contest. The AlexNet was a deep CNN composed of five convolutional and three fully connected layers that takes as input a colored image of size 224×224 and generates 1000 class label predictions. Figure 11 shows the layers of AlexNet architecture. Even though the performance of AlexNet was overcome by other more sophisticated networks in following years, it remains the standard CNN. The building blocks of the network are the five convolutional layers that have different filter configurations as shown in Figure 11. All convolutional and fully connected layers are followed by ReLu activation units, and the activation of first two

convolutional layers are followed by local response normalization units that aid the generalization of the network. As for the pooling layers, they all perform max-pooling; and although not very common, the pooling windows overlap with a stride of 2.

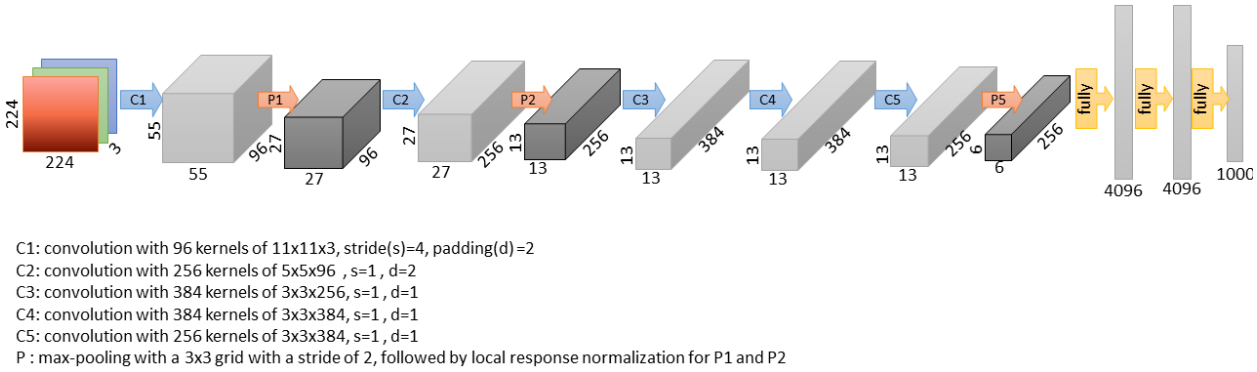


Figure 11: Architecture illustration of the AlexNet. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 290,400 - 186,624 - 64,896 - 64,896 - 43,264 - 4096 4096 1000.

AlexNet is known to generalize well, that is because it uses a lot of techniques to reduce overfitting. Image augmentation is used to increase the size of the training set and thus reduces overfitting by applying label-preserving transformations, where every image in the training set is replaced by ten transformed versions of its original form. This is achieved by randomly extracting five 224×224 patches from the original 256×256 scaled images, along with their horizontal reflections. The predictions made on each of the ten patches is then averaged which results in a decision that is much more robust to image translation. The second form of image augmentation is to alter the RGB pixels intensities of the images. This is done by performing PCA on the RGB pixel values throughout the training set, then adding multiples of the principle components to the original image which results in a prediction that is more robust to changes in image pixel intensities and color illumination. The dropout technique [42] is also used to reduce overfitting by applying an efficient version of model combination, that effectively prevents a unit in one layer from relying too strongly on a single unit in the previous layer.

To produce a class prediction using a trained AlexNet, every test image is resized to 256×256 and five crops (the four corners and center) of size 224×224 are taken from it. Then those five crops along with their horizontal reflections are presented independently to the trained network to produce a label prediction. The ten predictions are then averaged to output one predicted label for the test image. AlexNet CNN achieved a top-1 error rate of 37.5% and a top-5 error rate of 17.0% on the ILSVRC-2010, where the top-5 error rate is the fraction of test images for which the correct label is not among the five labels considered most probable by the model. CNN architectures continued to result in winning submissions in the competition for following years, as in ILSVRC-2013 with a top-5 error rate of 11% [51] where the network is densely applied at each spatial location and at multiple image scales to boost the performance of the CNN. And in ILSVRC-2014 with a top-5 classification error of 7% [52] where instead of the dense application of the network at each possible location, the CNN is applied to images that are randomly cropped from a range of randomly scaled images, which results in an architecture that generalizes better to unseen data.

3.2 CNN FOR VIDEO CLASSIFICATION

A video is a sequence of moving still-images, the most naive approach to process a video is to treat every single frame of it independently. Although in video classification the network has access not only to a single image but also to its temporal variations, some research suggested that taking the local motion into consideration does not seem to have a strong impact on improving the performance of sports video classification [5]. Therefore, as a starting point, we adopt the single frame classification model where independent single frames are used to classify the full length video.

The Sports-1M video data set [5] was used to study CNNs performance. The data consists of 1 million automatically labeled YouTube videos from 487 different classes of sports. However, since the Sports-1M data set labels are automatically generated and therefore noisy, we train our network on still images from the ImageNet database then test the trained network on video frames. A small subset of the Sports-1M data is used to test our models

where only eight ball sports labels: soccer, basketball, baseball, football, rugby, volleyball, ice-hockey and golf are investigated. The Sports-1M data set released in 2014 contained a total of 15,147 videos with labels belonging to the eight classes of interest. Because the released data set is composed of YouTube urls, some videos are removed or no longer available for all users. We were able to download a total of 12,307 videos to use for testing. Nonetheless, a good portion of those 12,307 videos wasn't used for testing due to the incorrect labels associated with them. To generate accurate prediction accuracies, the testing data was cleaned out by manually removing all mis-labeled videos from the data. Using all the available videos for testing is advantageous since it results in a more stable classification accuracy than the case when the available data set is split into training and testing. The training data on the other hand is composed of game-scenes, ball-scenes and other-scenes; where other-scenes have labels like sports equipment, gadgets, players or arena. The training images are downloaded from the ImageNet website [53]. Table 1 breaks down the training and testing labels and Figure 12 shows samples of the training data for every class.

Table 1: Training and testing data description

		# Training Images			**# Testing Videos**	
	Class	ball	game	other	initial download	correct labels
1	Baseball	974	1,851	742	1,405	973
2	Basketball	1,124	1,838	1256	1,411	772
3	Football	68	1,769	1230	1,649	924
4	Golf	1,181	1,680	1395	1,032	633
5	Ice hockey	1,275	1,875	1424	1,498	940
6	Rugby	1,508	1,732	0	1,502	750
7	Soccer	1,345	1,395	1402	1,896	1,292
8	Volleyball	1,283	1,735	1312	1,914	1,238
	Total	31,394			12,307	7,522

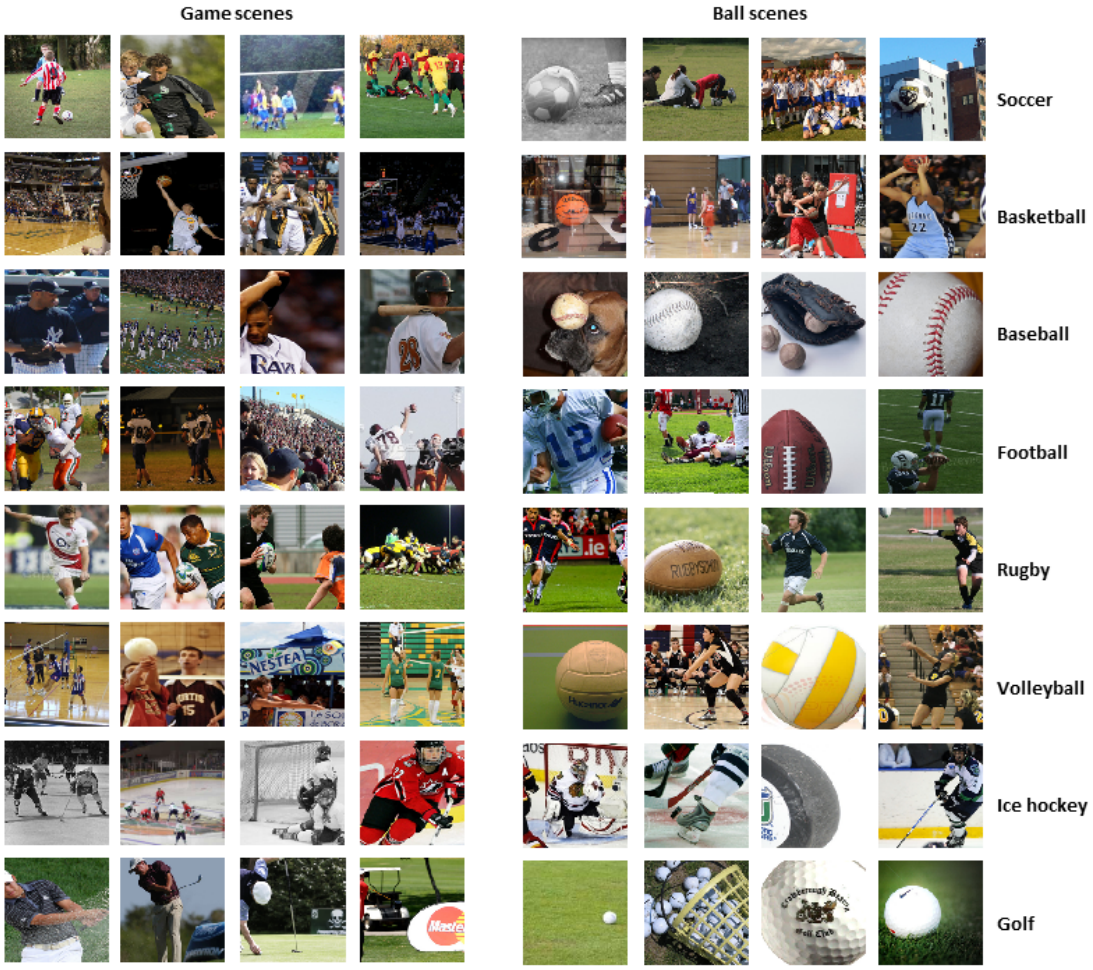


Figure 12: Four randomly selected samples of each training data class

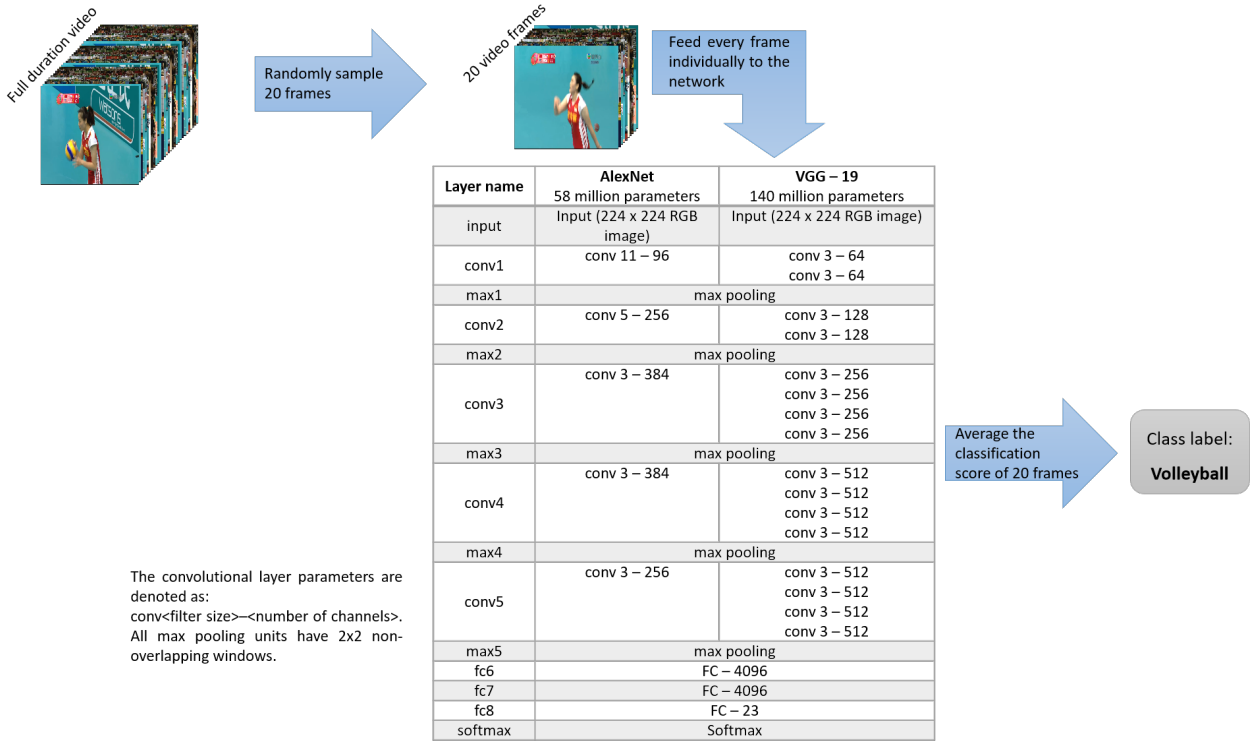
3.3 TRAINING A CNN ON IMAGES AND TESTING IT ON VIDEOS

The AlexNet and the VGG-19 [52] CNNs are used to classify the sports videos. Figure 13 illustrates the structure of the classification networks. VGG-19 is a very deep CNN, developed by the Visual Geometry Group (VGG) at the University of Oxford, similar to the AlexNet, however, the network is deeper and able of extracting finer features. MatConvNet package was used to construct the networks [54]. The networks are trained on 31,394 training images belonging to twenty-three sub-classes.

In the case of AlexNet, all training images are preprocessed by resizing each one to 256×256. Then five different crops of size 224×224 along with their horizontal flips are taken to be used as the training data. This image augmentation scheme increases the size of the training data by a factor of ten and helps the network to reduce overfitting. The mean image of all the training images is calculated and subtracted from every image. The trained network is then tested on the testing set which is composed of 7,522 videos from eight classes. To test a video, each testing frame is resized to 256×256 then five crops (4 corners + center) of size 224×224 are taken and presented along with their horizontal flips individually to the network after subtracting the mean image calculated on the training set from every test image. The average of the ten versions of each frame is calculated to assign a label to a frame.

On the other hand, for the VGG-19 case, the RGB mean value of the training data is found by averaging over the red, green and blue pixel values of each training image independently, where the resulting average RGB value is a 3×1 vector. During training, each training image is rescaled randomly so that its shortest side is of length S where $256 \leq S \leq 512$, then a random 224×224 crop is obtained from the rescaled image and is presented to the network's input after subtracting the RGB mean value. Each image is further flipped horizontally with a probability of 50%. The scale S is drawn uniformly at random for every training image in every iteration. This results in presenting different crops of the image to the network at different epochs, which can be seen as a form of data augmentation and is expected to help with generalization. In contrast to training, no crops are taken from the testing frames. Each testing frame is rescaled so that its shortest side's length is $Q = 384$ and the full scaled image is presented to the network's input. Providing the network with inputs of different sizes results in a class score map with variable spatial resolution and a number of channels equal to the number of classes. This score map is then averaged (by applying an average pooling unit to the output of the network) to obtain a $1 \times$ (number of classes) prediction score. Each testing frame is presented to the network along with its horizontal reflection. The prediction scores of the image and reflection are averaged to produce a final score for the frame.

To test a video, 20 video frames are randomly selected using a Gaussian distribution with $\mu = 0.5 * \text{video-duration}, \sigma = 0.1 * \text{video-duration}$. This is done to draw more frames from the center of the video rather than the beginning and end, as YouTube videos intro and ending usually do not represent the content of the videos. A Video is classified by taking the mean of all 20 predictions scores. The network assigns to each frame one of twenty-three component labels. To generate a video class prediction (one of eight video classes), the prediction scores of ball-scenes, other-scenes and game-scenes are added for every class then the class with the highest summed score is assigned to the video.



Layer name	AlexNet 58 million parameters	VGG – 19 140 million parameters
input	Input (224 x 224 RGB image)	Input (224 x 224 RGB image)
conv1	conv 11 – 96	conv 3 – 64 conv 3 – 64
max1	max pooling	
conv2	conv 5 – 256	conv 3 – 128 conv 3 – 128
max2	max pooling	
conv3	conv 3 – 384	conv 3 – 256 conv 3 – 256 conv 3 – 256 conv 3 – 256
max3	max pooling	
conv4	conv 3 – 384	conv 3 – 512 conv 3 – 512 conv 3 – 512 conv 3 – 512
max4	max pooling	
conv5	conv 3 – 256	conv 3 – 512 conv 3 – 512 conv 3 – 512 conv 3 – 512
max5	max pooling	
fc6	FC – 4096	
fc7	FC – 4096	
fc8	FC – 23	
softmax	Softmax	

The convolutional layer parameters are denoted as:
conv<filter size>–<number of channels>. All max pooling units have 2x2 non-overlapping windows.

Figure 13: CNN for video classification network architecture

3.4 RESULTS AND DISCUSSION

The networks in Figure 13 are used to classify the test videos. Test videos are processed online and fed to the network. The top-1 and top-2 (instead of top-5 since there are only 8 classes) accuracies are reported in Table 2. Figure 14 shows the confusion matrix for classifying the test videos based on 20 randomly selected frames using the VGG-19 architecture. The green diagonal cells show the number and percentage of correct classifications. For example 788 videos are correctly classified as football, which corresponds to 10.5% of all 7,522 videos. While red off-diagonal cells show the number and percentage of incorrect classifications. For example, 28 football videos are incorrectly classified as baseball, which corresponds to 0.4% of all 7,522 videos. The gray horizontal cells at the bottom indicate the percentage of correct/incorrect classification for each true video class. For example, 85.3% of all football videos were correctly classified while 14.7% are incorrectly classified. Similarly, the gray vertical cells at the right indicate the percentage of correct/incorrect classification of every predicted class. For example, 70.7% of the videos predicted as football are correctly classified while 29.3% are incorrectly classified. The bottom most diagonal purple cell shows the overall network performance, where 73.2% of all video class predictions are correct and 26.8% are incorrect.

The confusion matrix plotted in Figure 14 shows that the proposed network makes a lot of mistakes classifying soccer videos. Only 14.4% of all soccer videos are correctly classified as soccer. This is due to having most the training images labeled soccer for professional or amateur soccer played on conventional outdoor soccer fields, where most of the videos labeled soccer are taken during training in indoor gymnastic halls; which results in mis-classifying soccer videos as volleyball or basketball videos. This is one drawback of having independent sources for training and testing data. The other major source of errors comes from mis-classifying rugby videos as football, which is expected to remain challenging even with a more sophisticated architecture.

Table 2 shows that the VGG-19 architecture performs better than the traditional AlexNet network. It also shows that increasing the number of tested frames per video increases the classification accuracy. However, the accuracy seems to saturate when the number of tested

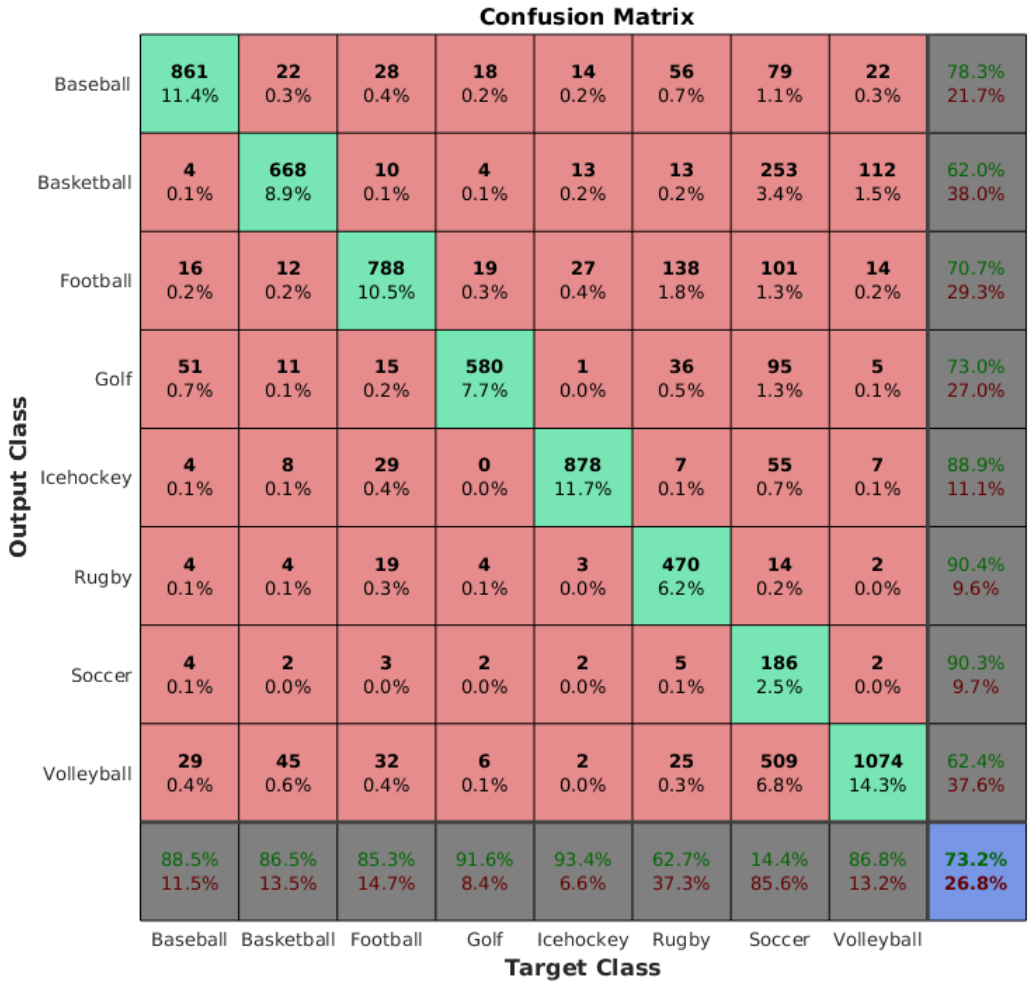


Figure 14: VGG19 classification confusion matrix - classifying videos based on 20 randomly selected frames

frames is increased to 100. Those results are comparable to the results reported in [5] where class predictions are made by randomly sampling 20 video-clips rather than 20 random frames. Predictions made on 20 video-clips achieved a top-1 classification accuracy of 60.9% [5] compared to our accuracy of 73.2%.

Table 2: Classification accuracy on test videos

	Accuracy %			
	AlexNet		**VGG-19**	
# Frames	**Top - 1**	**Top - 2**	**Top - 1**	**Top - 2**
1	48.1	64.2	60.8	73.2
2	53.9	68.8	67.5	77.8
10	60.3	73.8	72.5	81.3
20	61.6	74.5	73.2	81.7
100	62.2	74.8	73.3	82.1

4.0 RECURRENT NEURAL NETWORKS

To this point, we have naively treated videos as a series of independent still images and applied the image trained network on a number of randomly chosen frames of the test videos to output a single video label. In this chapter, we investigate how to incorporate the temporal information videos provide. We could state our problem as: given a set of training videos where each video is assigned a single label, how to classify a set of unlabeled test videos while incorporating both spatial and temporal features. To do so, we make use of recurrent neural networks, in particular long short term memory networks, that are known for their ability to process sequences of data. We use a CNN to learn and detect individual video frames spatial features, then those learned features are fed as a sequence to a recurrent neural network that will learn the temporal features of the video.

4.1 RECURRENT NEURAL NETWORKS

Recurrent neural networks (RNNs) are artificial neural networks with feedback loops between units. Unlike traditional neural networks, RNNs take into consideration not only the current input example they see, but also their own output one step back in time; which is the neural network equivalent of an AR (1) system. In other words, they perform the same task for every element of a sequence; and hence the name recurrent. This very property is what gives RNNs the ability to learn sequential or time-varying patterns [55]. RNNs, like other artificial neural network architectures, are inspired by the way humans think. Humans do not start thinking from scratch every time they face a challenge, instead they try to make sense of previously gained knowledge alongside with current knowledge. The feedback loops

that are the unique feature of RNNs allow for information learned in one step of the network to be used in the next step. Another way to understand RNNs is to think about them as networks that have memory where what has been learned in the previous step is stored in memory and is used in the next task.

The architecture of RNNs ranges from fully interconnected networks to partially inter-connected networks, where in the fully connected mode, every single unit is inter-connected to itself and to every other unit in the network [55]. Although RNNs may sound different than traditional neural networks, they are actually just very deep feed-forward networks in which all the layers share the same weights, as shown in Figure 15.

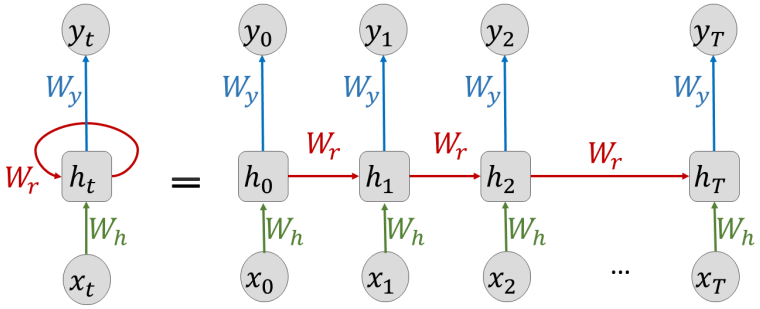


Figure 15: An unfolded recurrent neural network

At all times t, the network weights are shared but the outputs and losses are different. To obtain the gradient update, all the losses should be combined. The forward-pass equations of the network shown in Figure 15 could be written as,

$$h_t = f(W_h x_t + W_r h_{t-1} + b_h) \tag{4.1}$$

$$y_t = g(W_y h_t + b_y) \tag{4.2}$$

where $f(.)$ and $g(.)$ are activation functions, W_r is the recurrent weight, W_h, b_h, W_y, b_y are the input and output weight matrices and biases respectively. The combined cost function can thus be given as,

$$\frac{\partial E}{\partial W_r} = \sum_{t=0}^{T} \frac{\partial E_t}{\partial W_r} \tag{4.3}$$

The major drawback of RNNs is the problem of vanishing/exploding gradients [34] [23]. One of the most successful models of RNNs that overcomes this issue is the long short term memory architecture [56].

4.2 LONG SHORT TERM MEMORY MODELS

Conventional neural networks make the assumption that different inputs are completely independent on each-other. Although RNNs are capable of dealing with short term dependencies, their long term memory is pretty limited. For example a language model can easily predict the next word in the sentence "I am sailing my". Just the word "sailing" could be enough for the model to know that the next word is probably "boat". However, this is not always the case. If the model is trying to predict the next word in the sentence "I was raised in Japan, I speak fluent". The model will be able to figure out that the output is probably a language, however it needs a longer memory (up till the word "Japan") to be able to make a good prediction. This type of longer dependency (memory) is dealt with in long short term memory networks (LSTMs). The basic idea behind LSTMs is that they have a memory cell C that has a recurrent weight $W_r = 1$ to avoid the scaling effect. At the same time the network is enhanced by several units called gating units that allow memory cells to perform three tasks: flush, add to, and get from the memory. The gates performing those tasks are shown in Figure 16 and are called, forget, input and output gates, respectively. The idea of a gate is that it is a way to control the flow of information through the network. A gate takes as input the input of the memory cell at the current time x_t along with the output of the cell at the previous time step h_t, applies an affine transformation (using a learned weight matrix and a bias vector), then passes it through a sigmoid function to squash the gate's output values between 0 and 1 for each number in the cell state C_{t-1}. An output value of zero means that no information is passed through the gate, while a value of 1 means that everything is let through the gate. This control output is then used to produce the final

state of the memory cell C_t. The output of the forget, input and output gates (f_t, i_t and o_t respectively) is given by: [57]

$$f_t = \sigma\left(W_f[h_{t-1}, x_t] + b_f\right) \tag{4.4}$$

$$i_t = \sigma\left(W_i[h_{t-1}, x_t] + b_i\right) \tag{4.5}$$

$$o_t = \sigma\left(W_o[h_{t-1}, x_t] + b_o\right) \tag{4.6}$$

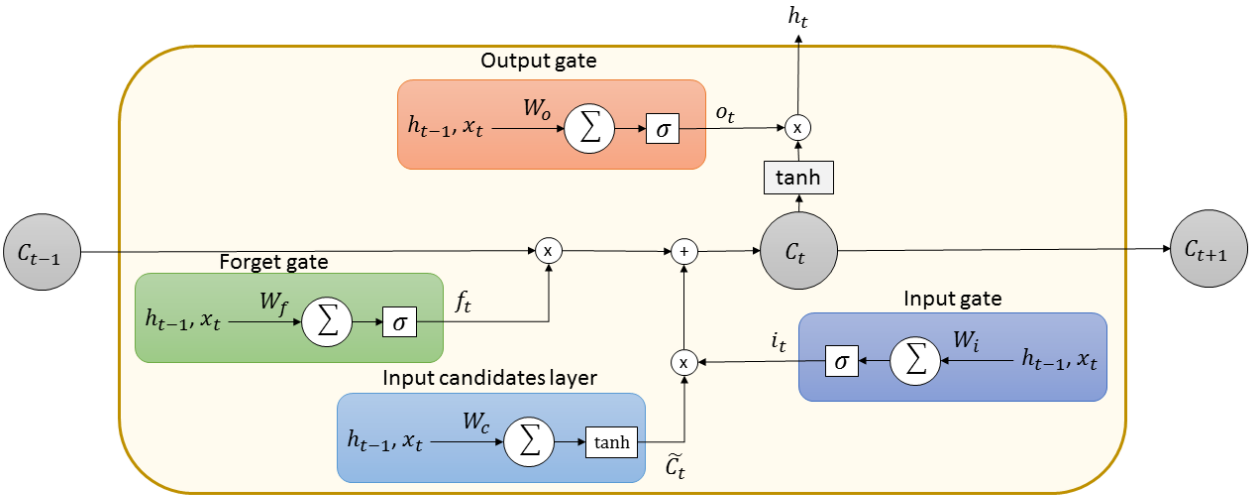


Figure 16: A standard LSTM memory cell with gating units

The first stage of the process an LSTM cell performs is to decide which information from the previous cell to forget and which information to pass to the next cell, this is done in the forget gate. The 0/1 output of the forget gate is multiplied with the previous time step's state to control which information is flushed from the memory and which is passed to the next state. In the next stage, the input gate decides which information is going to be updated; the updates themselves however are created in the input candidate layer. The output of the input candidate layer $\widetilde{C}_t$ are multiplied with the output of the input gate to generate a state update. This update is added to the output of the forget stage to form the updated state at the current time step. The final stage is to decide what the cell outputs. This is achieved using an output gate which produces a 0/1 control vector that is multiplied

by the cell state after passing it through a tanh activation function. The cell updated state C_t and the cell output h_t are thus given by,

$$C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t \tag{4.7}$$

$$h_t = o_t * \tanh(C_t) \tag{4.8}$$

where the $*$ operation is an element wise multiplication, and

$$\widetilde{C}_t = \tanh\left(W_c[h_{t-1}, x_t] + b_c\right) \tag{4.9}$$

The LSTM structure depicted in Figure 16 is only one of many different proposed architectures. Although there are several variations of LSTM structures, they all almost perform similarly [58].

4.3 BI-DIRECTIONAL LSTMS

Bidirectional LSTMs are simply bidirectional RNNs with LSTM recurrent units. The basic idea behind bidirectional RNNs (BRNNs) is to create two layers of the same network by mirroring the recurrent units of the original network and putting them side by side. This creates two layers of recurrent units, where the original input sequence is fed to the first (forward) layer and a reversed copy of the input sequence is fed to the second (backward) layer, then the output of the two layers is combined. The reason this is done is to allow the network to have access to information from both the past (in the forward layer) and the future (in the backward layer). In other words, all the available input information is to be used in order to form a better understanding of the sequence context.

Bidirectional RNNs were introduced in 1997 [59] where it was first suggested to split the state neurons of a regular RNN in a part that is responsible for the positive time direction (forward states) and a part for the negative time direction (backward states) as shown in Figure 17. It is worth noting that if the backward states are taken out of the structure, the resulting network is just the unidirectional RNN shown in Figure 15. Bidirectional LSTMs (BLSTMs) result from replacing the recurrent units (the green blocks in Figure 17) with

LSTM units (like the one shown in Figure 16). Bidirectional LSTMs are very commonly and successfully used in natural language processing applications [59, 60, 61]. In the next sections, we use a unidirectional LSTM and a bidirectional LSTM to classify videos from the Sports1M and the UCF101 data sets.

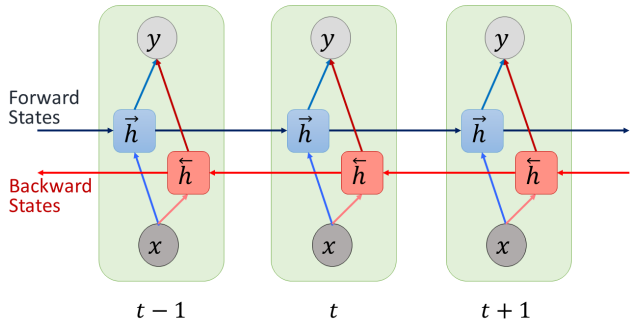


Figure 17: General structure of a bidirectional recurrent neural network

4.4 LSTMS FOR VIDEO CLASSIFICATION

The great ability of CNNs to learn detailed image features has encouraged a lot of research to study the use of CNNs to capture the spatial content of video frames then use some fusion methods to incorporate the temporal content of videos and generate a decision on the video class [5, 24, 62, 63]. However, these approaches capture the short term motions. LSTMs are ideal for capturing long term dependencies and thus have been succefully used for video classification achieving some of the best classification results reported in many video datsets [26, 28, 64, 65]. We aim to use LSTMs in a similar but slightly different manner and compare our results to reported results on the UCF101 data set and eight categories of the Sports-1M data set.

We continue to use the same small subset of the Sports-1M data set as in Chapter 3 (the subset contains 8 categories of ball sports) and we compare our results. Unlike previous work on video classification with LSTMs, we train a CNN to learn frame features from an

independent image data set (which contains 23 classes) as explained in Chapter 3. The trained CNN is then used to extract the spatial features from video frames. Since the variation between consecutive video frames is small, not all video frames are used during training. A number of video frames is extracted from each video to form a small video clip, and the LSTM network is trained on the frames of the short video clip. Unlike what we explored in Chapter 3, where a random selection of video frames is used for testing and classifying a video, the LSTM is tested on video clips of equally spaced frames.

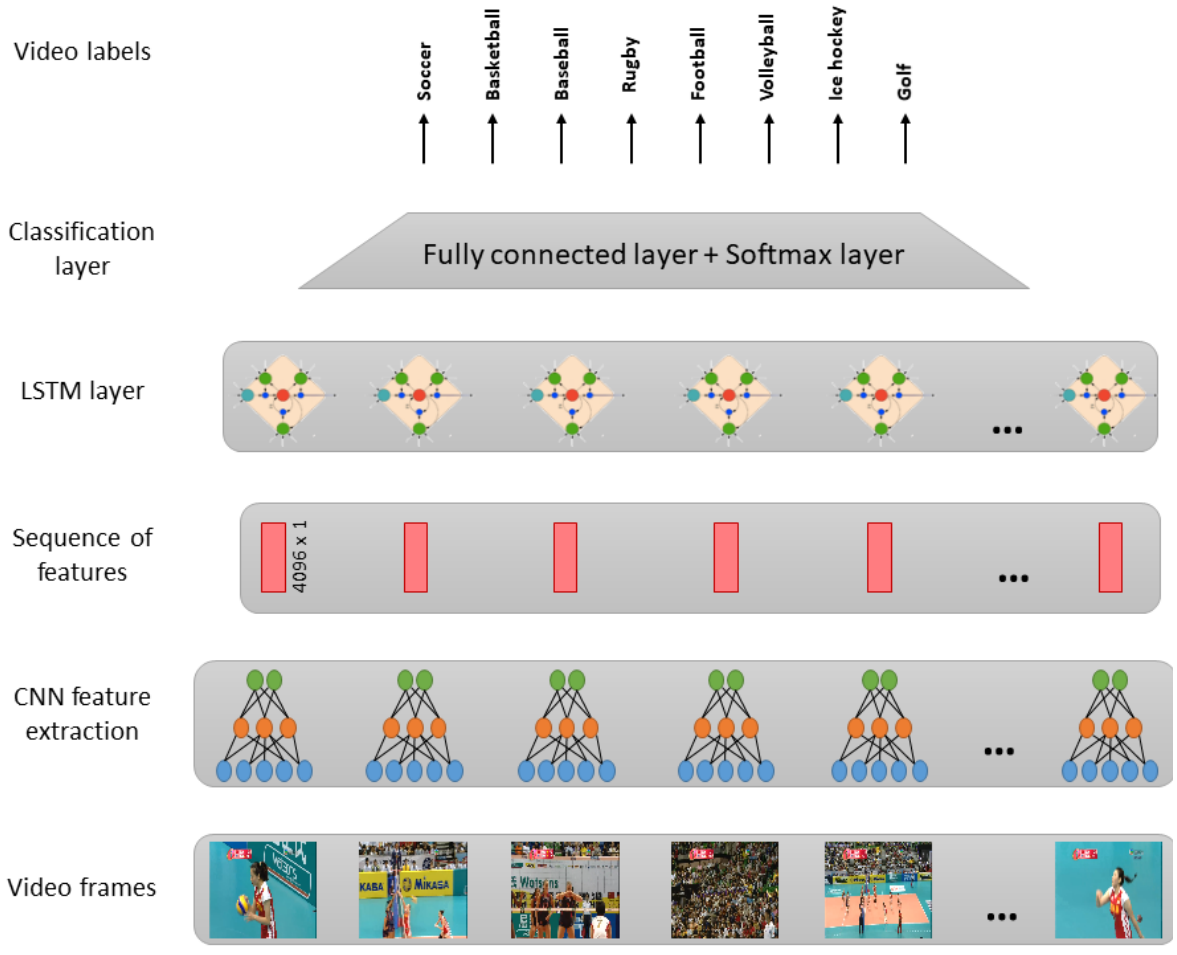


Figure 18: Proposed LSTM network structure

Since an LSTM is to be trained on video clips, the subset of the Sports-1M data set needs to be split into training, validation and testing sets. We split the data set by assigning 50% of the videos to the training set, 10% to a validation set and 40% to a test set. For each set, 300 equally spaced video frames are extracted from each video and the trained CNN is used to extract the features of the 300 images. The last fully connected layer before the final one (the fc7 layer in Figure 13) is used for feature extraction. The CNN extracts a features vector of size 4096×1 of each video frame, then these features are stacked in cells of length 300; where each cell contains the features of the 300 frames short video clip. The cells that contain the sequential features are finally used to train the LSTM networks, the structure of the LSTM network is shown in Figure 18. We use two shallow LSTM configurations, one with a single LSTM layer and another with two LSTM layers; where each LSTM layer has LSTM units with 100 memory cells. We investigate the two configurations once with unidirectional LSTM layers and once with bidirectional LSTM layers.

We further test our network on the UCF101 data set [66] to compare our results to previous work on a complete data set. The UCF101 is an action recognition data set of YouTube human action videos. It has 101 categories and a total of 13,320 videos. The data set is stable since it is available in .avi video format instead of YouTube URLs like in the Sports-1M data set. Three training/testing splits are provided with the data set, we report our classification performance on the first split. UCF101 videos are shorter than the Sports-1M videos, therefore, we extract 200 equally spaced frames of each video instead of 300 frames. A CNN (VGG-19) is trained on the 200 video frames, then the trained network is used to extract video features. The sequential features are then used to train LSTM networks to assign a label to each test video.

MATLAB is used to train the LSTM networks with the following training options. An initial learning rate of 1×10^{-3} is used with the adaptive moment estimation (Adam) optimizer [67] with a gradient decay factor of 0.9, a gradient threshold of 1 and an ℓ_2 regularization factor of 1×10^{-4}. The network is trained on minibatches of length 100 for a maximum of 100 epochs where training is stopped if the validation loss does not improve for 10 consecutive epochs.

4.5 RESULTS AND DISCUSSION

We compare our results using two approaches. In the first approach, a CNN is trained on ImageNet images from 23 classes related to the 8 sports classes of the Sports-1M subset. And in the second approach, the CNN is trained on video frames belonging to 8 classes of the Sports-1M subset. The trained CNNs from both approaches are then used to extract sequential video features of 300 video frames and those sequential features are used to train different LSTM network structures as illustrated in Figure 18 of the previous section. At the testing stage, videos clips of different number of frames are used to classify the videos, the classification accuracy results are summarized in Table 3.

Table 3: LSTM classification accuracy on eight labels of the Sports-1M videos

Number of tested video frames	Acuuracy %				
	one LSTM layer		two LSTM layers		
	LSTM	BLSTM	LSTM	BLSTM	
10	89.6	91.8	89.1	90.0	CNN trained on ImageNet
20	91.7	93.0	90.7	91.6	
100	93.9	95.2	93.7	93.3	
150	94.1	95.0	93.1	93.7	
300	94.5	95.2	93.2	93.4	
10	95.9	95.8	94.6	94.1	CNN trained on video frames
20	96.6	96.8	95.4	94.5	
100	96.6	97.2	95.9	94.9	
150	96.8	97.3	95.8	95.3	
300	96.8	97.3	95.6	94.8	

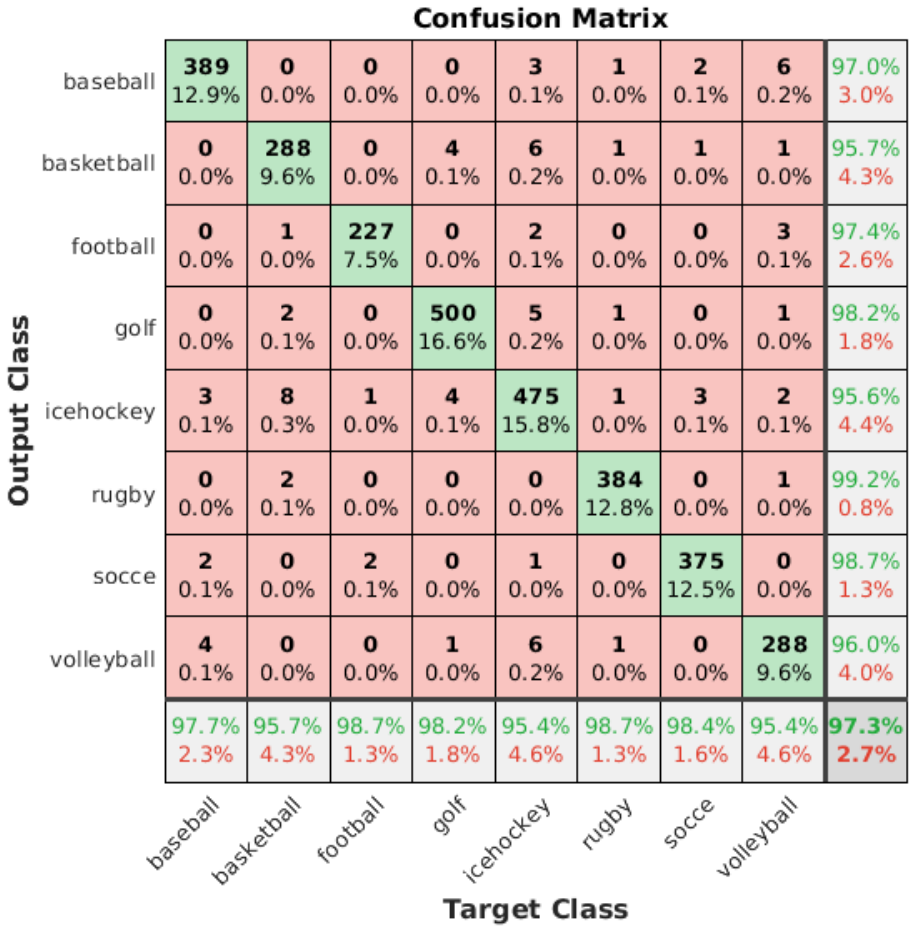


Figure 19: LSTM classification confusion matrix - classifying 300 frames of test videos

The results in Table 3 show classification accuracies in the 90% range even when processing only 10 video frames. The table also shows that a shallow single LSTM layer network performs better than the two LSTM layers, which may be due to overfitting on the training data as the network gets deeper. The results also show that both the unidirectional and bidirectional LSTM structures perform comparably, however, the bidirectional LSTM is less sensitive to reducing the number of processed test video frames. In other words, we can process fewer test frames and achieve a high classification accuracy when using a bidirectional LSTM. Those results outperform the results achieved when treating video frames as inde-

pendent images (the results are shown in Table 2) where a the maximum achieved accuracy was only 73.3% when processing 100 independent test video frames.

The accuracy percentages reported in Table 3 also show that LSTM networks trained on features extracted by a CNN trained on training video frames outperform the LSTM networks trained on features extracted by training a CNN on independent images. It makes sense that the LSTM network would perform better if trained on features extracted from the same data set it is trained on, however training on features extracted from an independent source is beneficial if the video data set does not provide enough frames to train a CNN. We confirm this hypothesis by assuming that we have a small training data set that is composed of only one-tenth of the original training videos. We train a CNN on video frames of this small training set, then use the trained CNN to extract the features of the training set and use those features to train a bidirectional single layered LSTM. Using this bidirectional LSTM on the testing set (the same testings set used in the experiments in Table 3) results in a classification accuracy of 86% when testing 300 frames of each test video. On the other hand, when using the CNN trained on an independent set of training images to extract the features of video frames of the small training set and then training a bidirectional LSTM on these features results in a classification accuracy of 90% when testing 300 frames of each test video.

Figure 19 shows the confusion matrix of single layer bidirectional LSTM trained on video frames extracted CNN features when processing 300 frames of the test videos. The plot shows that the accuracy and precision of every class of the test video is higher than 90%, which confirms the LSTM structure's ability to capture the temporal content of videos and result in high classification performance.

In order to compare our results to previous results on complete data sets, we test our network on the UCF101 data set. In this case we only train the CNN on 200 video frames of each training video in the data set and use the trained CNN to extract sequential features of the videos. Then use different LSTM structures to classify the videos. Table 4 summarizes the obtained classification acuuracies.

Table 4: LSTM classification accuracy on the UCF 101 data set

Number of tested video Frames	Acuuracy %			
	one LSTM layer		two LSTM layers	
	LSTM	BLSTM	LSTM	BLSTM
10	71.0	73.1	62.2	62.6
20	71.7	73.9	63.9	65.1
100	72.0	73.9	63.6	65.0
200	71.7	73.9	64.0	65.2

The results in Table 4 also show that a single layer LSTM structure results in a higher classification results than a two-layer LSTM. It could also be seen that the performance of the network is not sensitive to the number of tested frames, which is a very desirable result that means that the network could be trained on a large number of video frames and on the testing stage as few as 10 frames need to be processed to classify the entire video with a high classification accuracy. The table also shows that the bidirectional LSTM network outperforms the unidirectional LSTM resulting in a classification accuracy of 73.9%. These results are comparable to previous results, [26] and [64], published on the UCF101 data where two layered unidirectional LSTM networks are used to classify the test data. A classification accuracy of 71.1% was achieved [26] when using an LSTM network trained on CNN features extracted from the RGB information of video frames. However, this accuracy is increased to 82.7% when the classification scores of the LSTM trained on CNN features extracted by RGB information are averaged with the classification scores of another LSTM trained on CNN features extracted from the optical flow information of video frames. The state-of-the-art classification accuracy on the UCF101 data set is 91.3% which was achieved by [64], where the classification scores of two LSTM networks (trained on features extracted by a CNN trained on RGB information and a CNN trained on optical flow information) are combined using a neural network instead of simply averaging as in [26].

5.0 A GRAPH DECODING APPROACH FOR VIDEO CLASSIFICATION

In this chapter, we approach a finer problem where we aim to assign two levels of labels to each video frame of the test video; one label for the action performed in the video frame and another label for the overall video class. The problems could be stated as: given a training data set composed of a number of training videos, where each video is labeled with an overall class label and each frame of each video is labeled with an action label. And given a testing data set composed of a number of unlabeled videos:

- Assign an action label to each frame of each test video
- Assign an overall class label to each test video

We attempt to solve those problems by a graph decoding approach where a Hidden Markov Model (HMM) is constructed on top of a deep neural network. We train a deep CNN and an LSTM to classify video frames and use the features learned by the deep networks as observations for the HMM. However, to be able to do so, we need a data set that provides two levels of labeling to each video frame. The Sports-1M video data set [5] used in the previous chapters, although very rich in the number of videos and the number of classes, is assigned only one label for the entire video. In other words, all the frames in the video have the same label, which is the sport's class. For the purpose of action detection and video classification, we searched for a data set where frames representing different actions have different labels. The Actions for Cooking Eggs (ACE) data set [6] was released as the Kitchen Scene Context based Gesture Recognition (ACE) challange in ICPR 2012. Although not very popular, the data set provides videos of different classes where every frame of the video is assigned a different label based on the action that the frame represents and the entire video is assigned a class label that reflects the sequence of performed actions.

5.1 ACE DATA PROCESSING AND TRAINING

The classification of human activities in general and assisted daily living activities(ADL) in particular has drawn the attention of multiple research groups over the last two decades due to its impact on the quality of life of humans [68]. The construction of ADL data sets and the research conducted on the data has further influenced multiple applications in the domain of health care, sports science, movie production of animated avatars, surveillance and many more applications [69]. Kitchen cooking actions is one of the ADL multiple actions categories. Cooking is very dynamic and involves a rich variety of actions, change of original scenes, and different scenarios; which makes it very interesting for action classification. Multiple data sets have been created and studied for human kitchen activities, most of which provide segmented videos of actions; like the the ADL65 [70], the URADL [71] and the KUSK [72] data sets. On the other hand the CMU-MMAC data set [73] provides multiple cameras videos of full length cooking videos of different food menus and provides a subset of the data with frame annotation, where not only the action is labeled, but also the menu cooked is. A similar data set is the ACE data set [6] that provides a more consistent and refined set of actions that accompanies cooking menus that are very similar which makes it more challenging and interesting.

We conduct our research on the ACE data set. The data set consists of videos of seven different actors cooking one of five egg menus while performing one of eight possible cooking actions. When the actor is not performing one of the allowed actions, the frames are assigned a "no-action" label; Table 5 shows a description of the data. The training data consists of five videos per menu, while the testing data has two videos per menu. The major issue with this data set is that it is not large enough for deep learning applications; however, the way the frames are labeled is unique and rather interesting to explore. To use the data for training a CNN, the videos are processed by segmenting every training video into sub videos, each representing one of nine actions (eight cooking actions plus one no-action). Data augmentation is then used to increase the size of the training data. Video frames are augmented by making different versions of the same action video. This is achieved by extracting every n^{th} frame of the original video into a new video then applying random

rotations to the frames (with angles that range from -15 to 15°), then applying random horizontal flips (about the y-axis) to the rotated frames.

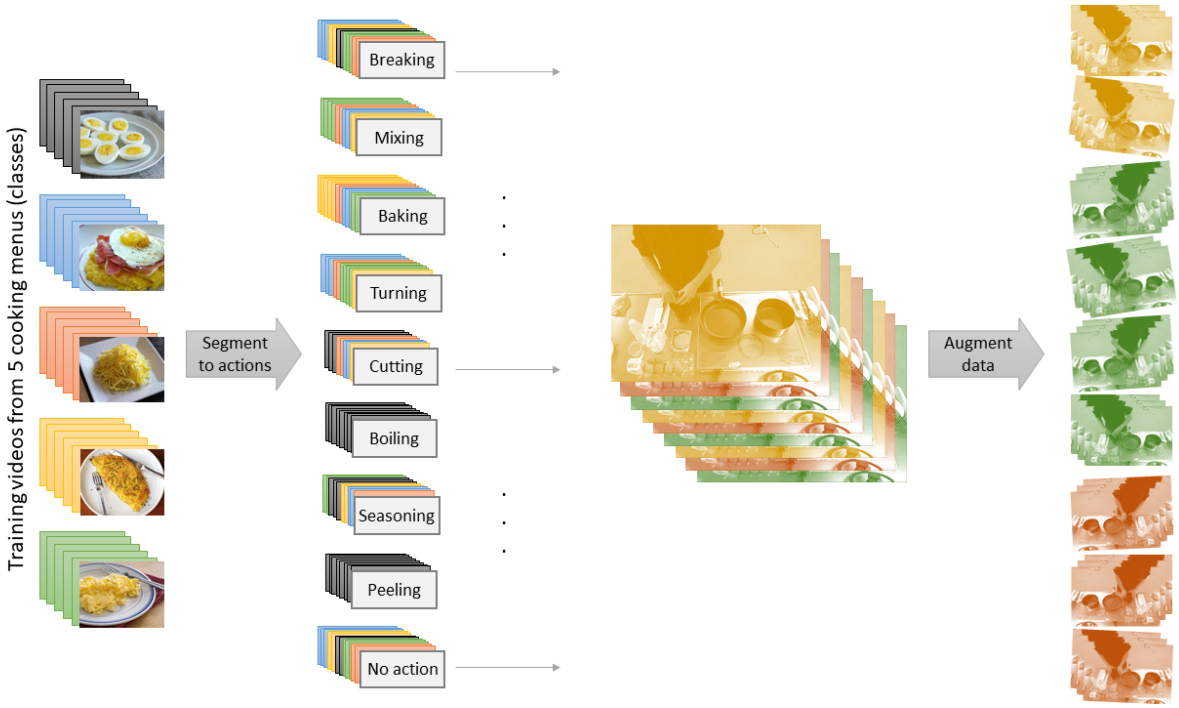


Figure 20: Pre-pocessing the ACE data

Table 5 below, shows the number of videos for each action in the raw training data set. It can be seen from the table that the number of video frames is imbalanced over the different actions. Therefore, data augmentation is applied in a way to have a better balanced number of videos for every action in order to make the data more suitable for training a CNN. To do so, in the data augmentation stage, actions with fewer than 10,000 frames in total are doubled in number by flipping all the frames horizontally. The description of the data after augmentation is shown in Table 5.

Table 5: Training action videos' description

		Before Augmentation		After Augmentation	
Action	Description	# Videos	#Frames	# Videos	# Frames
A	breaking	24	4,797	214	10,700
B	mixing	40	18,613	394	19,700
C	baking	47	43,218	888	44,400
D	turning	9	7,577	310	15,500
E	cutting	20	17,629	361	18,050
F	boiling	8	11,158	226	11,300
G	seasoning	15	5,419	234	11,700
H	peeling	5	10,99	222	11,100
I	no action	128	52,621	1115	55,750

After the data augmentation stage, the segmented video clips of actions are randomly split to a training and validation sets, where the validation set is composed of a randomly chosen 5% of video clips for every action label. The training data is then used to train a CNN to learn frames features. MATLAB's deep learning package is used for training, where the training is initialized by the VGG-19 network (shown in Figure 13) pretrained on the ImageNet data set [74]. During training all training frames from all cooking actions are shuffled once every epoch and are presented independently to the network. At the feature extraction phase, the order of the frames is maintained, the 1-D features (outputs of fc6 and fc7 layers) of each video action in the training data are extracted and those features are then stacked in cell arrays where each cell contains m sequences of n features and is of size $n \times m$, where m is the sequence length and n is the number of features. For our case, all action video clips are composed of 50 frames thus $m = 50$, and $n = 4096$, since we use the output of fc6 and fc7 layers as the learned features. The stacked sequence of features is used to train a shallow LSTM network with one LSTM layer followed by one fully-connected layer and a

classification layer. The LSTM layer is chosen to output the an action label of every frame of the action video sequence as it's making the classification decision at the last frame.

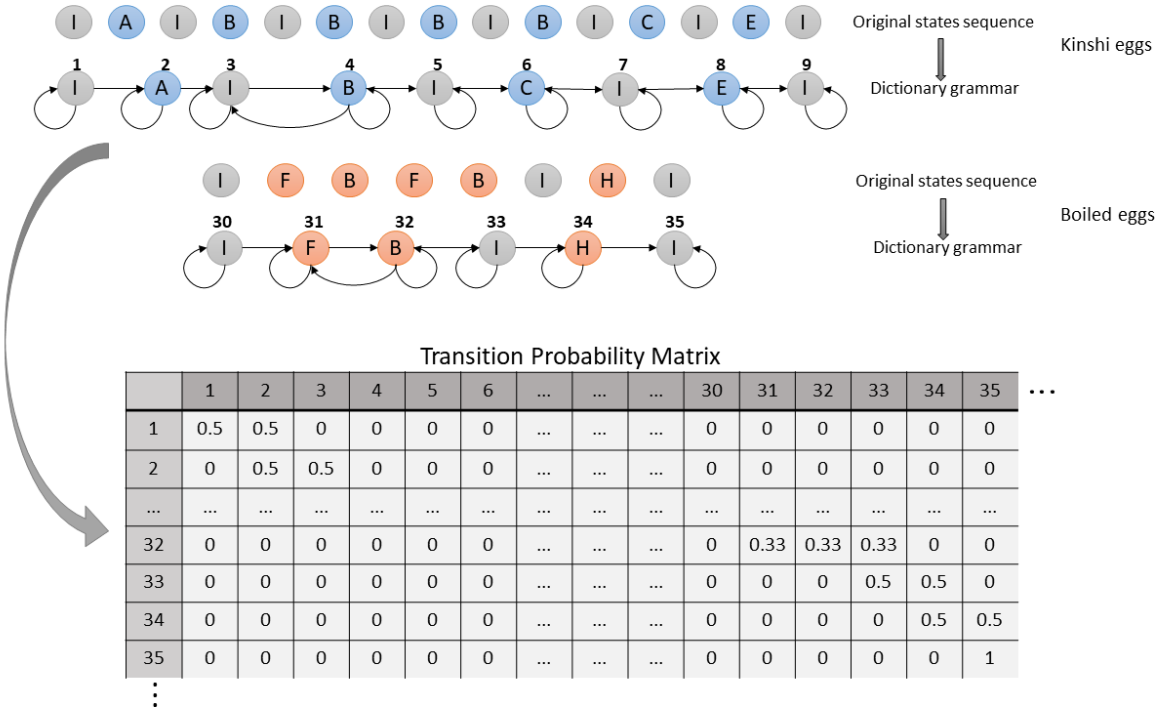
The CNN is trained to learn to classify independent video frames into one of nine actions, while the LSTM is trained to classify a video sequence into one action class. The input to the LSTM network is video frame sequence from one action and not a full video with multiple actions. To choose the number of memory cells for the LSTM layer, the training action videos are used for training LSTMs with different number of memory cells. Then a validation fold is performed where the full training videos (videos with multiple cooking actions) are used for validation since the network has never seen full videos during training.

650 memory cells were chosen for training an LSTM on fc7 CNN features, while 1000 memory cells were used for training on fc6 CNN features. Both the CNN and LSTM networks were trained with a stochastic gradient descent with momentum optimizer, with a momentum of 0.9 and weight decay of 5×10^{-4} on mini-batches of 50 frames (equal to the training videos length). The initial learning rate was set to 1×10^{-4} (because we initialized training from a pretrained network), the learning rate is scheduled to drop by a factor of 0.05 every epoch when training the CNN and every 10 epochs when training the LSTM.

5.2 BUILDING MENUS ACTION DICTIONARY

The CNN and LSTM networks are trained to classify a video frame or a sequence of frames into one class of cooking actions, we will call this stage the action detection stage. However, the question now is how to aggregate the sequence of detected actions into one menu class decision? To a machine, a video is a sequence of unidentified frames, similar to a sequence of unidentified words. In Automatic Speech Recognition (ASR), grammars are used to make a decision on weather a sequence of unidentified words is valid for the language the grammar comes from. Similarly in natural language processing, the problem of sentiment analysis can be solved by the lexicon based approach, where a dictionary is prepared to store polarity values of lexicons; then the polarity scores of each word of the text is added to get an overall polarity score that is used to classify the tone of the text as positive, negative or neutral [75].

We aim to approach the menu classification problem in a similar way, where a dictionary of cooking actions is prepared, then every sequence of actions is mapped to a certain menu.



	1	2	3	4	5	6	...	...	...	30	31	32	33	34	35	...
1	0.5	0.5	0	0	0	0	...	...	...	0	0	0	0	0	0	
2	0	0.5	0.5	0	0	0	...	...	...	0	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
32	0	0	0	0	0	0	...	...	...	0	0.33	0.33	0.33	0	0	
33	0	0	0	0	0	0	...	...	...	0	0	0	0.5	0.5	0	
34	0	0	0	0	0	0	...	...	...	0	0	0	0	0.5	0.5	
35	0	0	0	0	0	0	...	...	...	0	0	0	0	0	1	

Figure 21: Illustrative menu grammars and the corresponding transition matrix

The full duration training videos are used to build a menu dictionary. For every menu, the action sequences are determined for all five actor cooks, where identical cooking action sequences are merged in the dictionary (the full dictionary of grammars is shown in Figure 22). Every action is then assigned a unique numerical state, which makes every numerical state sequence also unique and points to a certain cooking menu. To put it differently, a dictionary of cooking menus is composed of multiple action grammars per menu, where each grammar is unique and is composed of a sequence of unique states. Finally a transition probability matrix of all permitted state transitions is constructed by marking all allowed transitions in the training data then giving all the transitions an equal probability. An illustration of a dictionary and its corresponding transition matrix is shown in Figure 21, the alphabetic actions are described in Table 5. Figure 21 shows one grammar example

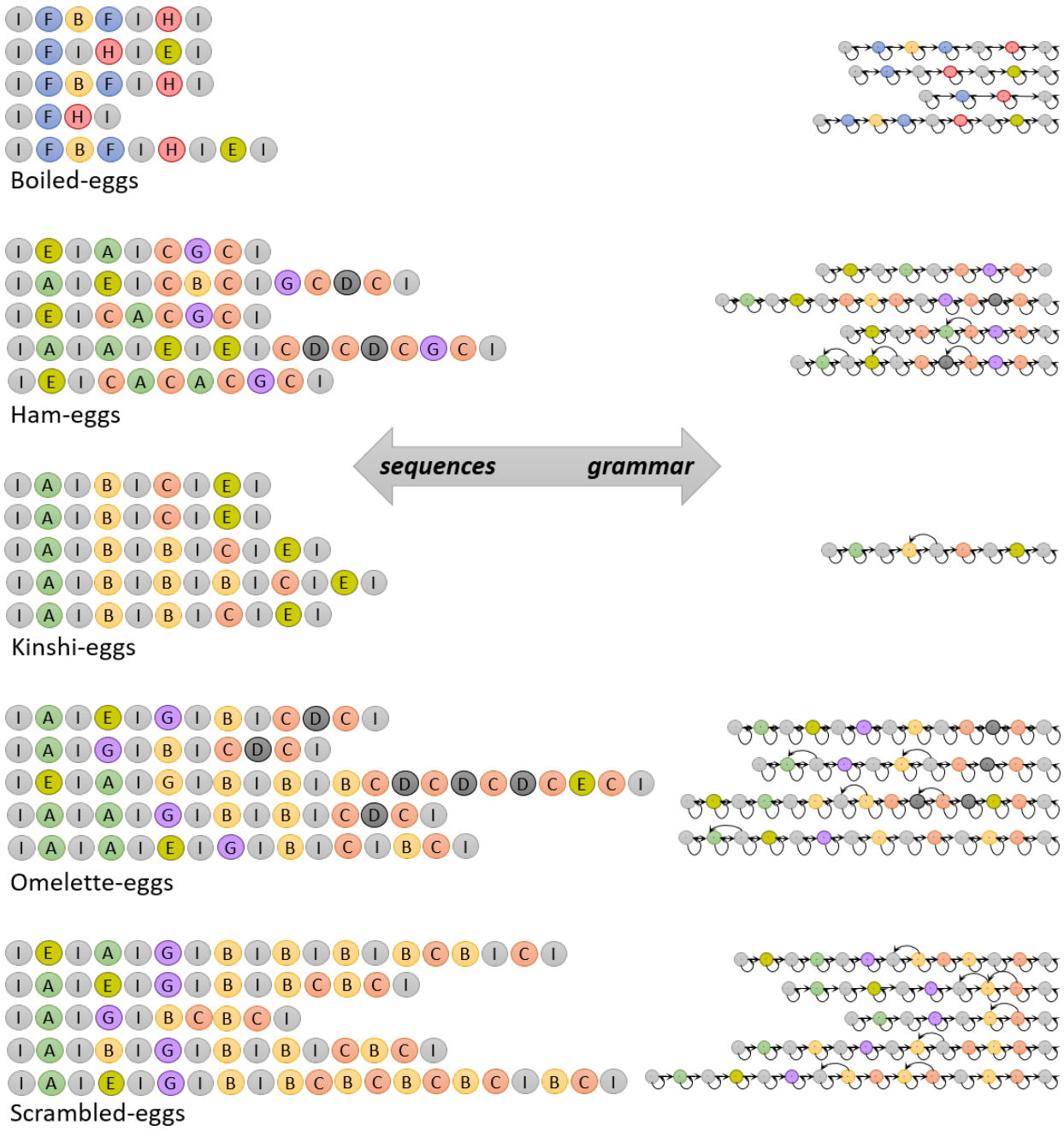


Figure 22: Full menu dictionary construction from training video sequences

for the kinshi-eggs menu and another example for the boiled-eggs menu, the plots show how a sequence of letter states is converted to a graphical grammar that is assigned one of

five menu labels. Every letter state in the grammar is mapped to a unique numeric state (numeric states 1, 3, 5, 7, 9, 30, 33 and 35 all represent the same letter state I; and states 4 and 32 represent state B). The transition probability matrix translates the dictionary of graphical grammars into a matrix, where every element in the matrix represents the probability of transitioning from the numeric state of the row the element falls in to the numeric state of its column. It's worth mentioning that every state is allowed to transition to its own and that all transitions are assigned equal probabilities no matter how many times a transition occurs in the training data. For example, for the boiled-eggs grammar illustrated in Figure 21, numeric state 32 is allowed to transition to either state 33, 31 or itself and thus elements $\{32, 31\}$, $\{32, 32\}$, $\{32, 33\}$ in the transition matrix are assigned an equal transition probability of $\frac{1}{3}$; while element $\{35, 35\}$ is assigned a transition probability of 1 since state 35 is only allowed to transition to itself based on the graphical grammar.

5.3 VITERBI DECODING AND HIDDEN MARKOV MODELS

The transition probability matrix constructed in the previous section along with the CNN and LSTM learned features are used for frames action detection and for menu classification. The problem can be restated as: given the action classification scores (learned features) and a model (a transition probability matrix) concluded from a training data, how to find the best cooking states sequence that is the underlying source of the sequence of the observed data? This is a typical HMM problem called the decoding problem. The HMM is a sequence model, or a sequence classifier, that works on assigning a label to each unit in a sequence.

5.3.1 Hidden Markov Models

The Hidden Markov Model (HMM) is a probabilistic sequence model that takes as input a sequence of units (in language processing, for example, these units could be words, letters, or sentences) and computes a probability distribution over possible sequences of labels to choose the best label sequence that results in the sequence of input units [76]. This sequence

of input units are referred to as the observations, where the sequence of labels is called the hidden states.

A classic HMM has two fundamental components, the hidden states and the observations. The states and observations sets are defined as [76]:

- The hidden states set, S, of N possible states, where $S = \{S_1, S_2, ..., S_N\}$ with q_t being the current hidden state at time t
- The observations set, V, of M possible observation symbols, $V = \{v_1, v_2, ..., v_M\}$, where the HMM model generates a sequence O of T observations, $O = \{O_1, O_2, ..., O_T\}$, with each O_t drawn from the observation set V.

And a set of probabilities that relates the hidden states to the observations, those probabilities are defined as [76]:

- The initial state probability, which is vector of N elements each describes the probability of the first hidden state being equal to the i^{th} hidden state S_i. The initial state probability vector is defined as,

$$\pi_i = P[q_1 = S_i], \qquad 1 \leq i \leq N \qquad \text{s.t} \qquad \sum_{i=1}^{N} \pi_i = 1 \tag{5.1}$$

- The state transition probability matrix $A_{N\times N}$, which characterizes the probability of transitioning from one state S_i to the other S_j,

$$a_{ij} = P[q_{t+1} = S_j \backslash q_t = S_i], \qquad 1 \leq i, j \leq N \qquad \text{s.t} \qquad \sum_{j=1}^{N} a_{ij} = 1 \quad \forall i \tag{5.2}$$

- The emission probability matrix $B_{M\times N}$ that describes the probability of an observation $O_t \in V$ being generated from a state S_i. Elements of the emission probability matrix are defined as,

$$b_i(O_t) = P[O_t = v_j \backslash q_t = S_i], \qquad 1 \leq i \leq N, \quad 1 \leq j \leq M \tag{5.3}$$

  In other words, b_{ij} represents the probability of observing v_j from state S_i.

A first-order hidden Markov model make two simplifying assumptions [76]. The first, is that the probability of a particular state depends only on the previous state, which is also called the Markov assumption,

$$P(q_i \backslash q_i, ..., q_{i-1}) = P(q_i \backslash q_{i-1}) \tag{5.4}$$

And the second assumption is that the probability of an output observation O_t only depends on the state that produced the observation q_i and not on any other state or any other observation, which is also called the independence assumption,

$$P(O_t \backslash q_1, ..., q_i, ..., q_T, O_1, ..., O_t, ..., O_T) = P(O_t \backslash q_i) \tag{5.5}$$

To illustrate the HMM and all its components, let's take a simple example where our model has three unique numerical states $S = \{1, 2, 3\}$ and two possible observations $V = \{\text{Yes}, \text{No}\}$. Let's assume that it is equally likely start in any of the three states. Also assume that state 1 can only transition to state 2, state 2 can only transition to state 3 and state 3 can transition to either sate 1 or to itself. Finally, assume that only a "Yes" could be observed from hidden state 1, only a "No" could be observed from hidden state 2, and either a "Yes" or a "No" are equally possible to be observed from hidden state 3. This example is illustrated in Figure 23 where the HMM and all its elements are defined. The figure also illustrates the probability matrices relating the hidden states set and the observation set as constructed based on equations 5.1, 5.2, 5.3 above.

Once an HMM is completely defined, $\lambda = (A, B)$, then three fundamental problems emerge and could be solved [77],

- Problem 1 (Likelihood): Given an HMM $\lambda = (A, B, \pi)$ and an observation sequence $O = \{O_1, O_2, ..., O_T\}$, how to determine the likelihood $P(O \backslash \lambda)$ (the probability of the observation sequence given the model) ?
- Problem 2 (Decoding): Given an observation sequence $O = \{O_1, O_2, ..., O_T\}$ and an HMM model $\lambda = (A, B, \pi)$, how to discover the best hidden state sequence $Q = \{q_1, q_2, ..., q_T\}$ which best explains the observations?

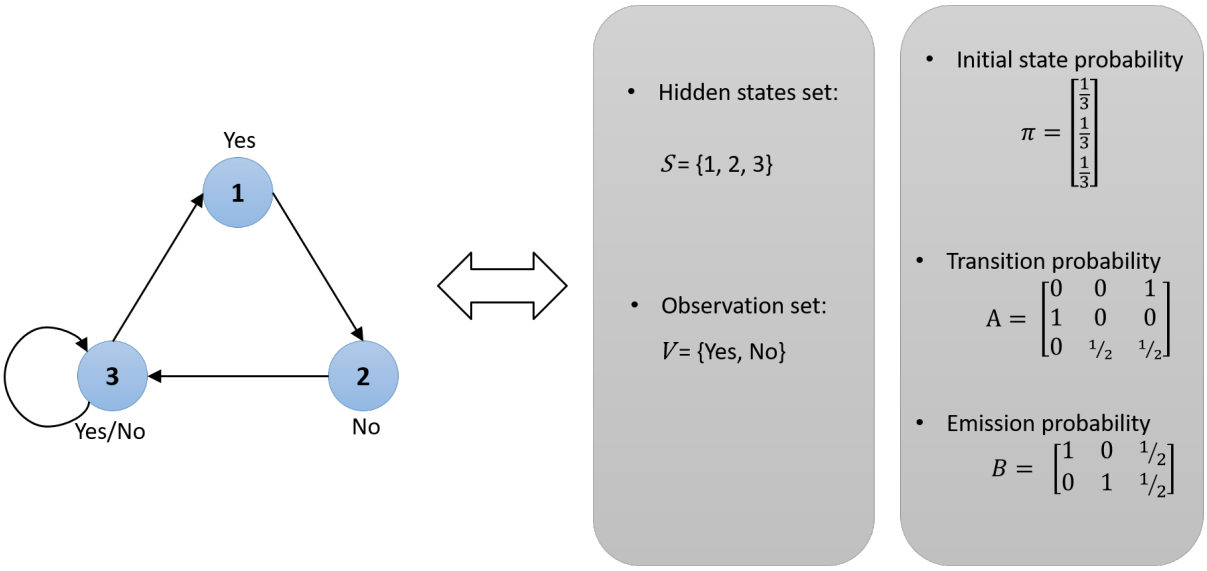


Figure 23: Hidden Markov Model and components illustrative example

- Problem 3 (Learning): Given an observation sequence $O = \{O_1, O_2, ..., O_T\}$ and the set of states $Q = \{q_1, q_2, ..., q_T\}$, how to learn the HMM parameters $\lambda = (A, B, \pi)$ to maximize $P(O \backslash \lambda)$?

We are particularly interested in the decoding problem, which is solved by the Viterbi algorithm.

5.3.2 The Viterbi Algorithm

The decoding problem stated as: given an HMM model $(\lambda = A, B, \pi)$ and a sequence of observations $O = O_1 O_2 ... O_T$, find the single best states sequence $Q = q_1 q_2 ... q_T$ that was responsible for generating the observations. An example of the decoding problem based on our simple illustrative example in Figure 23 would be: given an HMM model $\lambda = (A, B, \pi)$ and a set of observations over time $O = \{\text{Yes, No, No, No, Yes}\}$, find the best sequence of states that produced the observations. The solution to the decoding problem is given by the Viterbi algorithm [76], which is based on dynamic programming methods. The Viterbi

algorithm finds the single best sequence of hidden states, called the Viterbi path, that results in the sequence of observations.

The basic idea behind the Viterbi algorithm is to process the observation sequence left to right, filling out a trellis. Each cell of the trellis, $\delta_t(i)$, represents the probability that the HMM is in state S_i after seeing the first t observations and passing through the most probable state sequence $q_0 q_1 ... q_{t-1}$. The value of each cell $\delta_t(i)$ is computed by recursively taking the most probable path that could lead us to this cell [77],

$$\delta_t(i) = \max_{q_1, q_2, ..., q_{t-1}} P[q_1 q_2 ... q_{t-1} q_t, O_1 O_2 ... O_t, q_t = S_i \backslash \lambda] \tag{5.6}$$

The most probable path is represented by taking the maximum over all possible previous state sequences ($\max_{q_1, q_2, ..., q_{t-1}}$). To put it differently, $\delta_t(i)$ is the probability of the most probable state sequence that has S_i as its final state and that is responsible for the first t observations. Given that we had already computed the probability of being in every state at time t, the Viterbi probability at time $t+1$ is computed by taking the most probable of the extensions of the paths that lead to the current cell. Thus, the value of $\delta_{t+1}(j)$ should maximize the probability of the best path at the previous time incident t that ended in state S_j and maximizes the probability of transitioning to state S_i from state S_j and observing $O_{t+1} = v_j$,

$$\delta_{t+1}(j) = \max_i [\delta_t(i)\, a_{ij}]\; b_j(O_{t+1}) \tag{5.7}$$

The Viterbi path can be retrieved by keeping track of the states that were responsible of maximizing the score in Equation 5.7 (these are the states that led us to the current state). We store the back pointers in the array $\psi_t(j)$. At the end after processing the final observation at time T, we back-trace the best path to the beginning. The steps to apply the Viterbi algorithm can be summarized as follows [77]:

1. Initialize:

$$\begin{aligned} \delta_1(i) &= \pi_i\, b_i(O_1) \qquad 1 \le i \le N \\ \psi_1(i) &= 0 \end{aligned} \tag{5.8}$$

2. Recurs:

$$\begin{aligned} \delta_t(j) &= \max_{1\le i\le N}[\delta_{t-1}(i)\, a_{ij}]\; b_j(O_t), \quad 2 \le t \le T \\ & \qquad\qquad\qquad\qquad\qquad\quad 1 \le j \le N \end{aligned} \tag{5.9}$$

$$\begin{aligned} \psi_t(j) &= \operatorname*{argmax}_{1\le i\le N}[\delta_{t-1}(i)\, a_{ij}], \qquad 2 \le t \le T \\ & \qquad\qquad\qquad\qquad\qquad\quad 1 \le j \le N \end{aligned} \tag{5.10}$$

3. Terminate:

$$\begin{aligned} P^* &= \max_{1\le i\le N}[\delta_T(i)] \\ q_T^* &= \operatorname*{argmax}_{1\le i\le N}[\delta_T(i)] \end{aligned} \tag{5.11}$$

4. Backtrace:

$$q_t^* = \psi_{t+1}(q_{t+1}^*), \qquad t = T-1, T-2, ..., 1 \tag{5.12}$$

where q^* is the Viterbi path (best state sequence) and P^* is the probability of the path (highest probability amongst all other paths).

5.4 VITERBI DECODING FOR ACTION DETECTION AND MENU CLASSIFICATION

After building a dictionary of graphical grammars for cooking actions for each menu as discussed in section 5.2, the constructed graphs are translated into a single transition probability matrix A. An initial state probability vector π could be concluded from the training data by counting. The states set S is the set of N numerical states that could be mapped to nine possible actions (eight cooking actions plus one no-action state). What is left to completely define an HMM model as described in Section 5.3 is to define the observations set V and the emission probability matrix B. In our case, we are interested in applying Viterbi decoding to find the best sequence of states q^* (cooking actions) that results in the sequence of observations (a classification score array for each processed video frame). We do so by assuming that the observation panel for our model is composed of a set of N LEDs

that light up with a different probability based on the model's hidden state. This means that every hidden state (cooking action) is directly linked to an observation LED that lights up with the probability of correctly classifying the current video frame (at time t) to the hidden state by the deep network. Therefore, those probabilities are the deep network classification scores achieved from the CNN or the LSTM networks. In other words, the probability of the i^{th} LED to light up at time t ($P[O_t = 1 \backslash q_t = S_i]$) is the deep network's classification score of the t^{th} video frame to the i^{th} action. In this case, the emission probability matrix B could be defined as the $N \times T$ matrix that contains the deep network classification scores for each video frame. Note that the deep network classification scores for a video frame is a 1×9 vector that needs to be mapped to its corresponding $1 \times N$ vector, where N is the total number of all unique numerical states. Figure 24 shows an example of the mapping based on the kinshi-eggs illustrative grammar shown in Figure 21. Now that our HMM model is

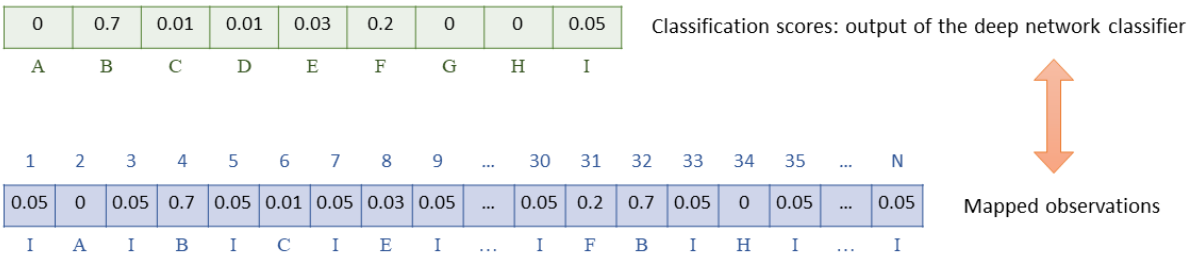


Figure 24: Mapping the original 9 cooking states classification scores into N numerical states observations

completely defined, we proceed to applying Viterbi decoding as described in the previous section. It should be noted that in the termination step (Equation 5.11), we must force the decoder to find the state that maximizes the probability score $\delta_T(i)$ within the states that are defined as end states in the dictionary to insure that the returned Viterbi path starts and ends in agreement with dictionary paths.

The classification scores produced by the CNN and the LSTM networks already classify (detect action) a video frame into one of nine actions. However, we pass those scores on to

the Viterbi decoder in an attempt to increase the action detection accuracy by correcting some classification mistakes. The input to the Viterbi decoder is a set of classification scores per video frame that reflects the action detection (the detected action is the one with the highest classification score); while the output of the Viterbi decoder is a sequence of detected state (one state per video frame). This sequence of detected states is governed by the transitional probability matrix A and thus follows the grammar rules set in the menu dictionary. Therefore, even if the deep network classifies a frame action as "mixing", for example, while the previous detected action was "peeling", since the grammar does not allow for action-mixing to come after action-peeling, the valid actions sequence in the Viterbi decoder will correct the detected action to another permitted one based on the dictionary and the probabilities output by the deep network.

Once the deep network detected actions are modified so that the detected actions sequence is in agreement with the grammars in the dictionary, the system assigns every actions sequence to a cooking menu. This is done fairly easily, since all the numeric states in the grammar are unique, then the menu classification could be easily found by matching the detected actions sequence to its corresponding menu in the dictionary. Figure 25 below illustrates the menu classification process.

5.5 RESULTS AND DISCUSSION

The ACE, despite being a finely labeled and accurate data set, has not been very popular due to its small size; however, considerable work has been previously reported on the data set [78, 79, 80, 81, 82, 83, 84, 85, 86]. We compare our results only to five previous approaches [82, 83, 84, 85, 86] selected to reflect the most recent work on the data set and the diversity in the methods and the performance of the work.

The winning team of the ICPR 2012 challenge [82] used a scene context based approach where SVM is used to detect a set of objects in the kitchen scene then semantic relationships between different objects are used to detect various cooking motions; this approach achieves an action detection accuracy of 72%.

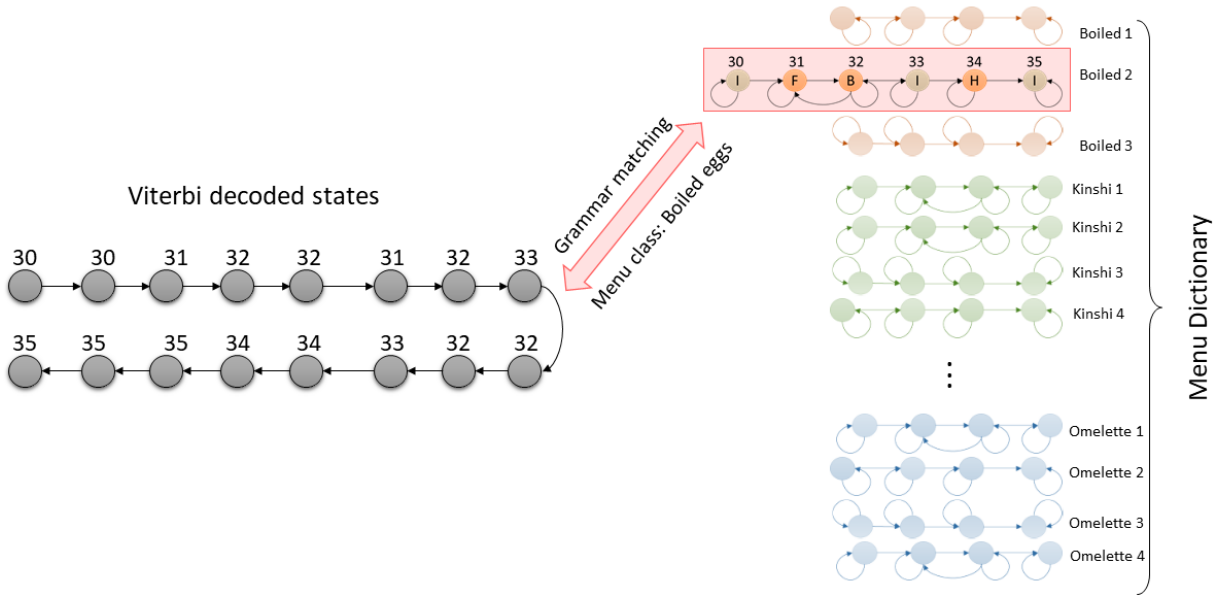


Figure 25: Menu classification from detected states

Another approach we compare our work to is work published on 2017 [83] where deep neural networks, namely CNNs, are used to learn features form both the RGB and the optical flow information the ACE data set provides. Then various methods are used to fuse the features and finally classify every video frame into a certain action; this process results in a top accuracy of 73%.

Another work in 2017 [84] achieves an action detection accuracy of 76.4%, which is the highest reported accuracy to date. The detection problem is tackled by employing a spatio-temporal representation for both the appearance of the cook and the surrounding kitchen utensils. An SVM is used on features extracted by the histogram of oriented gradient variation which can represent not only appearance and temporal change of independent objects but locations of these objects.

Another recent work presented in 2017 [85] achieved a maximum precision of 89% for the "seasoning" action and an overall action detection accuracy of 70%. The approach proposes a hybrid system that uses CNN for image feature extraction followed by a symbolic behavior recognition (SBR) step for plan recognition. This hybrid approach, although does not achieve

the highest detection accuracy, is the only previous work that attempts to examine the menu classification task besides action detection; the hybrid system however outputs a set of hypothesis menus and not a single menu.

Finally we compare our results to another interesting work presented also in 2017 [86]. This approach unlike all the previous work performed on the data set, performs action recognition and video summarization simultaneously. In other words, the recognition is performed on a subset of the video frames that best describes the entire video. A latent structural SVM is used on features extracted by the histogram of oriented gradients and the histogram of optical flow to classify video actions. An overall action detection accuracy on segmented test videos of 77.9% is achieved when only processing the 10 frames that best describes each action, compared to an accuracy of 54.7% they achieve when testing all video clips' frames. This method, however, could not be directly compared to our work or other previous work only because the task of video segmentation is not performed, and the accuracy is reported on already segmented action video clips. However, we still compare it to our results on the effect of the number of processed frames on the overall action detection accuracy.

Since different approaches tackle different problems and report different sets of results, multiple tables are used to compare the results. The comparisons between our proposed system and previous approaches are summarized in Table 7 and Table 8. The graphical approach explained in the previous sections was applied on video features extracted by deep networks trained on the ACE data to investigate the accuracy of the menu classification, action detection and next action prediction tasks. We examine those tasks on features learned from a deep CNN, an LSTM trained on the last fully-connected CNN layer (fc7) and an LSTM trained on CNN's fc6 features. We examine multiple factors while performing the machine learning tasks, namely, the effect of the grammar used in the Viterbi decoder, the effect of the number of processed frames, the effect of state delays and the effect of the length of the available frames on the system performance. Figure 26 illustrates the CNN-HMM proposed system.

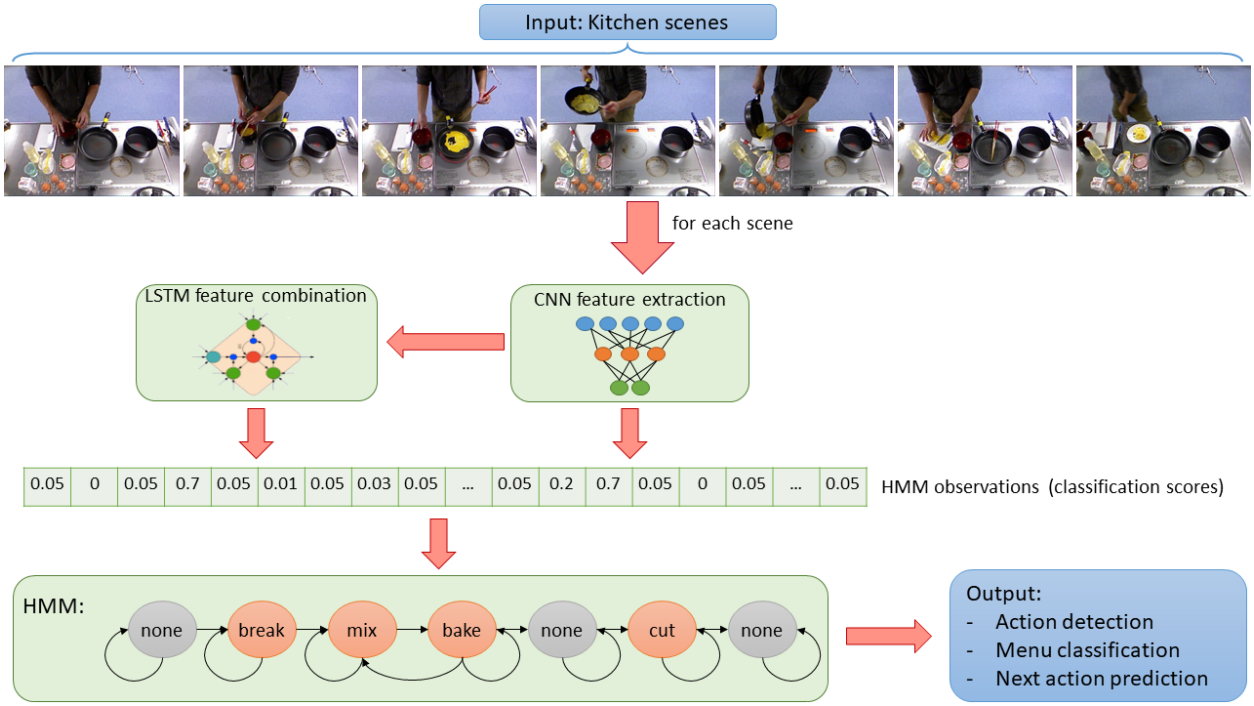


Figure 26: Proposed system (CNN/LSTM followed by HMM) illustration

5.5.1 The effect of using menu dictionaries on detected actions

The first factor we consider is the dictionary construction. Initially, the grammar was built using the training data only (five training examples per each of five menu classes). Every frame of every video in the testing data set was processed and the action detection and menu classification accuracies were recorded. The menu classification accuracy was at most 80%, which means that two out of the ten testing videos were incorrectly classified. Upon investigating the results for all features and categories, it was noticed that the two videos associated with the Kinshi-egg menu were never correctly classified and that was because the two testing videos followed a grammar that was never present in the training data's dictionary, as shown in Figure 27. However, that action (turning) is a common action when preparing several egg menus and is expected to be present in the Kinshi menu grammar as well as the omelets and scrambled eggs menus. For that reason, we explore the effect

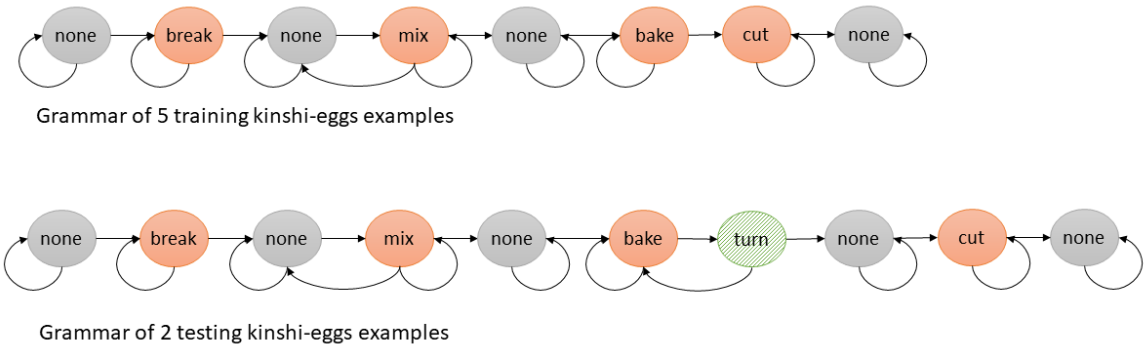


Figure 27: Absent cooking action in the Kinshi-eggs training grammar

of inserting common sense actions in the grammar. We investigate two types of grammar insertions: testing data sneak-peek insertion, where we sneak a peek at the grammar of the testing data and add any grammar absent from the training videos into the grammar. And neutral state insertion, where we insert a neutral state (no-action state) between action states to give the system the option to go for a no-action state or bypasses it anytime it sees an unidentified action, as sketched in Figure 28.

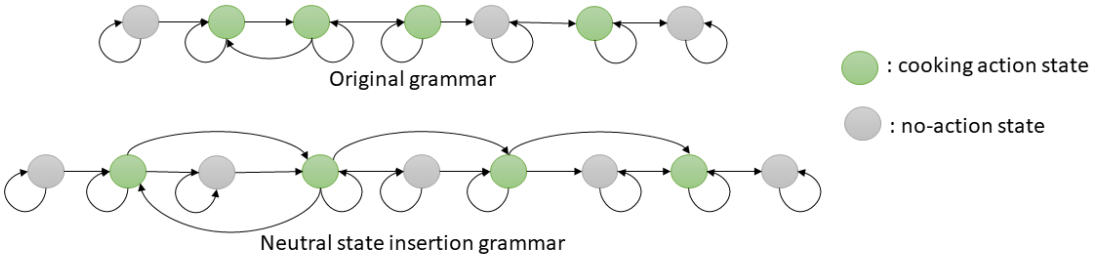


Figure 28: Neutral states grammar insertion illustration

The action detection and menu classification accuracy performance results for different grammars is summarized in Table 6. It can be seen from the table that sneaking a peek at the testing data grammar (which could be thought of as inserting common sense grammars

into the dictionary) improves the menu classification greatly by increasing the accuracy to 90% for the LSTM-fc7 input features and 100% for the CNN input features. This result is not surprising, since the action classifier without any grammar already performs well with an accuracy of 76.35% when using a CNN alone and with an accuracy of 75.50% when using an LSTM trained on CNN extracted fc7 features. Then adding a grammar that corrects for classification mistakes and that represents all training and testing grammars data is expected to enhance action detection and to boost the menu classification accuracy. On the other hand, adding a neutral state between action states although did not result in a large improvement to the menu classification accuracy, it had an interesting effect on the menu classification output. It allowed for correctly classifying the Kinshi-eggs menus because the grammar now allowed the system to classify the "turning" action as "no-action" which now agrees with one of the Kinshi-eggs grammars. However, adding the neutral states results in the system to confuse menus Scramble-eggs and Omeletts to one another because they have the same set of actions and in a similar sequence; it also results in decreasing the action detection accuracy because the grammar now is allowing for more cooking actions to be corrected as "no-action".

The table also shows that the action detection and menu classification results when using an LSTM trained on fc6 CNN features are not comparable neither to a CNN performing alone nor to an LSTM trained on fc7 features with or without a grammar. That is because the fc6 features are not the best features to start with, which is due to the fact that more sophisticated features are learned in the final layers of the CNN, and thus the final fully connected layer (fc7) would provide the best learned features to be either used directly in a classifier (CNN alone) or used for learning temporal features with an LSTM. For that reason, we will omit reporting results on LSTM-fc6 approach when comparing our results to previously reported results in the literature, and will refer to the LSTM network trained on fc7 CNN features when we refer to our LSTM approach. We will also report all the experiments and results on the common-sense grammar insertion dictionary.

Table 6: Action detection and menu classification accuracy over different dictionaries

Input features	Accuracy without grammar	Accuracy per grammar insertion type			Task
		No insertion	Neutral state	Common sense	
LSTM (fc6)	73.51	71.49	70.52	70.95	All action
LSTM (fc7)	75.50	76.45	75.43	77.91	detection
CNN	76.35	78.36	77.39	79.16	
LSTM (fc6)	65.37	61.28	59.85	61.32	Cooking
LSTM (fc7)	67.97	68.35	66.76	70.32	action
CNN	70.00	70.77	68.91	72.08	detection
LSTM (fc6)	-	70	60	50	Menu
LSTM (fc7)	-	80	90	90	classification
CNN	-	80	80	100	

Figure 29 shows the confusion matrix of the action detection labels when using a common-sense grammar insertion on CNN input features. The figure shows the overall action detection accuracy along with individual actions detection accuracies. The diagonal cells correspond to actions that are correctly detected; while the off-diagonal cells correspond to incorrectly detected actions. Both the number of frames and the percentage of the total number of frames are shown in each cell. The intensity of each diagonal and off-diagonal cell reflects the percentage in the cell, where more intense colors represent higher percentages and washed out colors indicate low percentages. The column on the far right shows the percentages of all the actions predicted to belong to each class that are correctly (precision) and incorrectly detected. The row at the bottom shows the percentages of all actions belonging to each class that are correctly (recall) and incorrectly detected. The cell in the bottom right of the plot shows the overall detection accuracy. It could be seen that the system makes its biggest mistakes when detecting "turning" actions, as it only successfully detects 11.0%

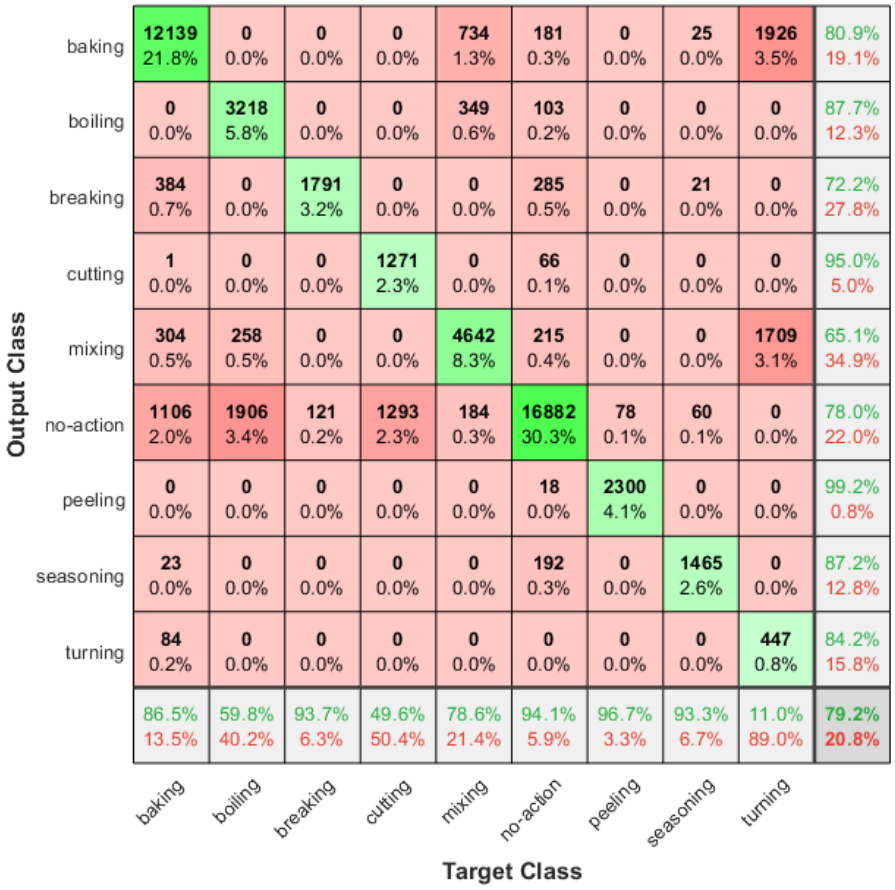


Figure 29: Confusion matrix for action detection using sneak-peek grammar insertion on CNN features

of all "turning" actions, while the rest 89.0% are detected as either "baking" or "mixing" (this could also be seen in Figure 32 where the bottom graph shows mis-classifying action "turning" as action "mixing" and "baking"). This is most likely because action "turning" always comes either before or after actions "baking" and "mixing". That's why it gets confused more by the two actions especially when adding a grammar (it is worth mentioning that the detection accuracy for action "turning" was 13.6% , which is 2% higher before correcting it with a common-sense grammar). The other source of big mistakes is due to mis-classifying actions "boiling" and "cutting" as "no-actions", where near 50% of all

"cutting" and "boiling" actions are miss-classified as "no-action". The plot also shows that it is easy for the system to correctly classify "no-action" labels; on the other hand, it is easy for the system to confuse cooking actions to "no-action". The confusion matrix also shows that the system is very precise in the detection it makes. This could be seen from the column at the right, where all the detection made are correct with a percentage of at least 65.1% and up to 99.2% at most with an overall precision of 83.3%.

5.5.2 The effect of the number of tested video frames on menu classification

In the previous sub-section, we reported the accuracy results for action detection and menu classification when every single frame of the test videos is processed. It is good to test our system against all frames to know how well it performs (especially for action detection), however, do we need to process every single frame to make a conclusion on the actions performed in the video and the menu class it belongs to? To answer this question we investigate the effect of processing part of the video on the overall menu class. We will not use any perfectly engineered methods to select the part of the video frames we process, unlike what is reported in [86] where only the video summary frames are processed. Instead we, roughly, process every other F frames of the test video, where F is referred to as the frame step. When processing every other F frame, the action detection accuracy could be reported in two ways: either by comparing processed frames' action labels with their corresponding true frames' action labels (which means comparing fewer labels) as shown in the left plots of Figure 30 ; or by comparing all the true action labels to the labels sequence constructed by interpolating the sequence of processed frames so that all unprocessed frames have the same label of their neighboring processed frames as shown in the right plots of Figure 30. Figure 30 shows that the two accuracy measures are very similar; therefore, for the remainder of this chapter, we will report on the accuracy achieved by comparing processed frames labels to their corresponding true labels only.

The graphs in Figure 30 also show the action detection accuracy for all actions (top plots) and for cooking actions, excluding "no-action" labels, (bottom plots). The plots show that cooking actions are detected with a lower accuracy than all actions; which is due to the

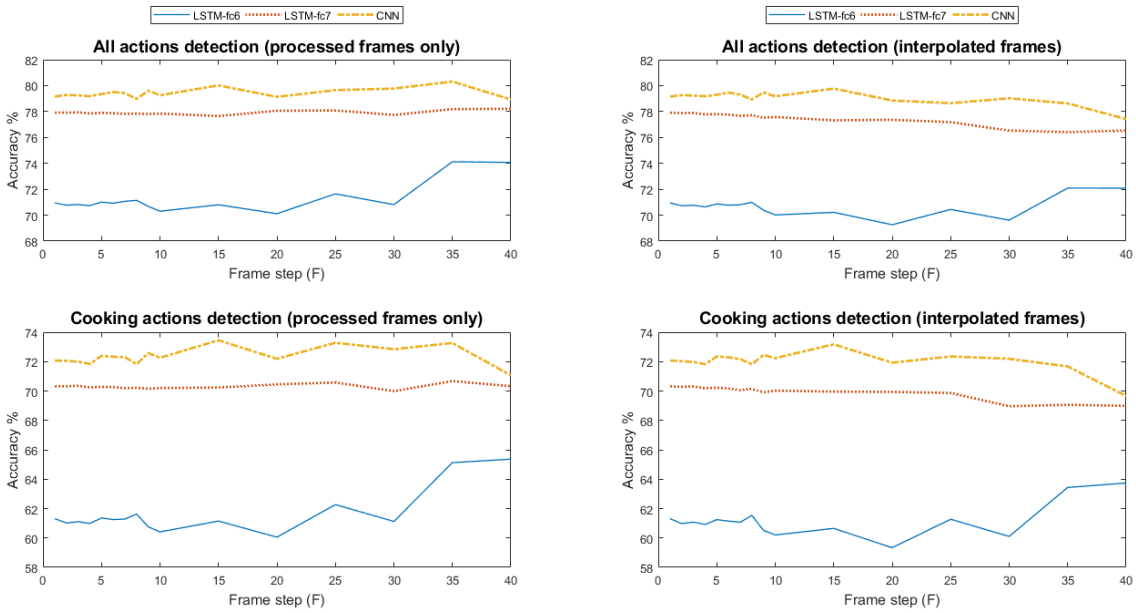


Figure 30: Action detection accuracy with different frame steps

high accuracy of detecting "no-action" labels and the large number of "no-action" frames, as seen in Figure 29. The plots also show that increasing the frame step (processing fewer frames) increases the accuracy for most cases and does not result in reducing the accuracy for any case (compared to processing every video frame). This indicates that processing fewer frames results in noise reduction, due to not processing frames that contain redundant or noisy information. This could be seen especially in the case of using LSTM-fc6 features, that could be thought of as noisy features, where the detection accuracy is notably enhanced when processing fewer frames. The plots also show a maximum accuracy of 80% for the CNN extracted features when processing every other 35^{th} frame. The longest test video we process is composed of a little less than 6510 frames, which means that we are processing less than 185 frames of each video to achieve an accuracy of 80%. This result could be compared to the of 78% accuracy reported in [86] where 10 frames per every (already segmented) action are processed, i.e. a total of 90 frames per video. Those 90 frames however are not only already segmented, but also are selected to best describe each action (segmented action summary).

In contrast, for our case at most 185 equally spaced frames of each full video are processed and achieve a higher accuracy. It is important to note that a deep CNN with no grammar will output the same action labels for the same frames when processing all frames or part of them, however, using a grammar forces the system to output a different sequence of action labels when a different number of frames is processed so that the output sequence continues to follow the grammar.

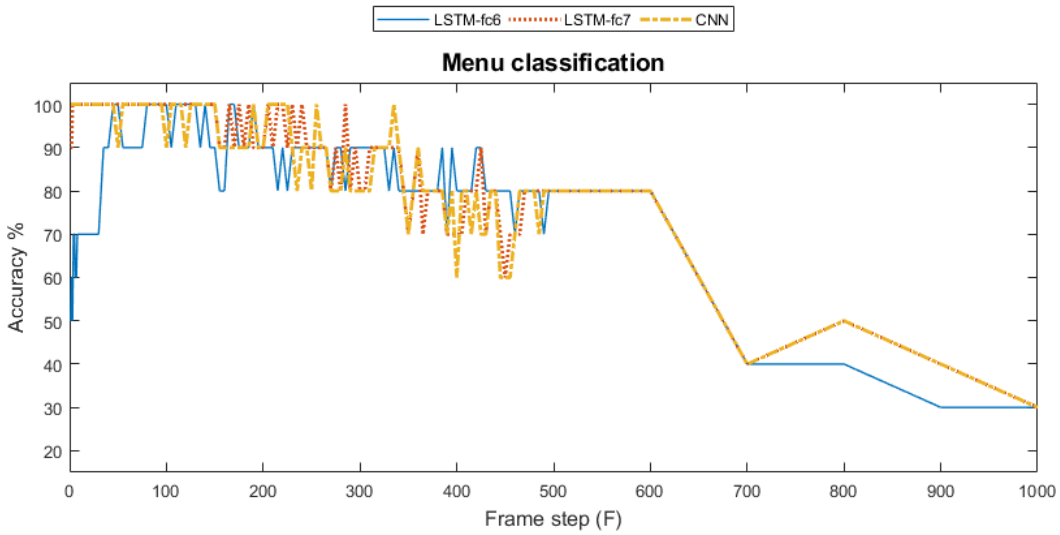


Figure 31: Menu classification accuracy with different frame steps

The effect of changing the number of processed frames may not be very obvious on action detection performance, but it is more visible on the menu classification performance. Figure 31 shows the menu classification accuracy with a big range of frame steps. The plots show that for the CNN approach, even when processing 1 of each 150 frames (which is equivalent to processing as few as, at most, 44 video frames), the menu classification accuracy stays at 100%, then it decreases as even fewer frames are processed. Since the longest test video has less than 6510 frames, this means that our menu classification system is very accurate even when processing less than 1% of the test video frames. On the other hand, for the LSTM-fc6 approach, the menu classification increases as fewer frames are processed. It could be seen from the plots that the accuracy becomes in the range of $80-100\%$ when using a frame step

larger than $F = 40$ (compared to 50% when processing every frame), this again is a result of noise reduction due to processing fewer noisy features. These results are very interesting because increasing the frame step enhances (or at least does not reduce) both the action detection and menu classification accuracy; at the same time it reduces the computation cost.

5.5.3 The effect of state delay on menu classification and action detection

Figure 32 shows a time series representation of the different cooking actions for every single frame of two sample test videos. The two sample videos are chosen to showcase all cooking actions; it is also worth noting that the bottom video is the worst performing test video for the action detection task. The top three time series in the figure show the true video action labels, the action detection output when using a CNN with no grammar and the system's action detection output when using a CNN with the common-sense grammar insertion along with their associated detection accuracy. We will comment on the bottom two time series representations later. The plots illustrate how adding a grammar results in changing some action detection decisions so that the output action sequence is in agreement with the menu grammar. The graphs also show that in many cases, the output of our system bounces between cooking action states which leads to more detection mistakes due to fast transitioning between states. This issue could be addressed by introducing a state delay τ in the grammar, which forces the decoder to stay in each state for a certain number of frames before allowing it to make a transition to a different state. This is achieved, as sketched in Figure 33, by inserting a number of states replicas between each cooking state (it is important not to add state delays to "no-action" states, since the output state already spends a long time at "no-action"). The state delay affects the grammar as illustrated in Figure 33, where introducing a single state delay ($\tau = 2$) results in forcing the decoder to output the same state for at least two consecutive frames, in other words for a minimum of τ frames.

Figure 34 shows the relationship between the system's performance and the state delay for two different frame steps, $F = 1$ (left plots) and $F = 35$ (right plots). The accuracy plots in Figure 34 show that action detection benefits slightly from introducing state delays, this

Note: for the last case of each video example (grammar with state delay when processing every other 35 frames), the processed frames are marked with white bars while the detected actions of the unprocessed frames are interpolated to be equal to the processed frames actions. The accuracy between parentheses indicates the accuracy when comparing the labels of the processed frames to their corresponding true labels; while the interpolated actions accuracy is the one outside the parentheses

Figure 32: Visualization of original action labels vs output of CNN without grammar and with common-sense grammar insertion for different state delays and frames steps

could be seen especially for the CNN and the LSTM-fc6 cases (for both $F = 1$ and $F = 35$ for upto a state delay of $\tau = 6$). The figure also shows that even though a large state delay does not affect the system's performance negatively when processing every single video frame, the performance severely deteriorates with large state delays ($\tau > 6$) when processing a frame out of every other 35 frames. This is because we are processing fewer frames and at the same time forcing the decoder to remain in the same state longer while following the grammar, which in turns results in the system outputting states sequences that follow the shortest grammar sequence even if it is not the correct one, which results in making mistakes on both action detection and menu classification. In Figure 32, we show the effect of introducing a state delay of $\tau = 30$ and $\tau = 6$ on the detected actions when processing every single video

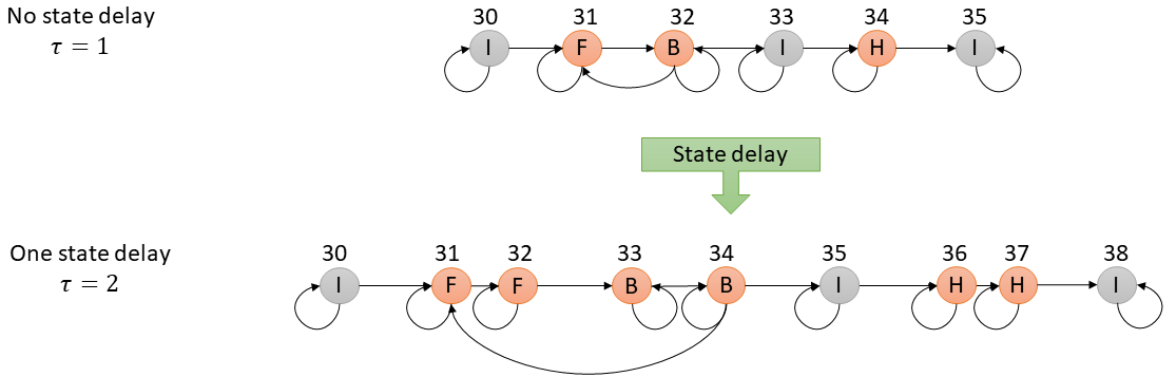


Figure 33: State delay effect on menu grammar

frame and one of every 35 frames, respectively, for the CNN approach. It could be seen from the forth sequences of both examples in Figure 32 that introducing a state delay of $\tau = 30$ forces the output state sequence not to transition to another state until it stays in the same state for at least 30 consecutive frames, in contrast with having no state delay ($\tau = 1$) as in the third sequence, which allows for fast transitions between states.

It is important to mention that adding a state delay increases the size of the transition probability matrix (from $A_{N\times N}$ to less than $A_{\tau(N\times N)}$; we can only place an upper bound on the size since no state delay is applied to no-action states) and thus increases the computation complexity of the system greatly with large state delays. Therefore, even though a state delay of $\tau = 30$ (when processing every video frame) increases the overall action detection to 80.1% (compared to 79.2% with no state delay), using a high state delay is not recommended as it slows down the computations. However, processing fewer frames reduces the computation complexity, therefore we try to balance the state delay τ and the frame step F by examining the system's performance over a grid of state delays and frame steps. Figure 35 shows 3-D surface plots of the menu classification and all action detection accuracy as a function of changing frame steps and state delays when using CNN features with common-sense grammar insertion. The plots confirm that adding a state delay enhances the action detection accuracy

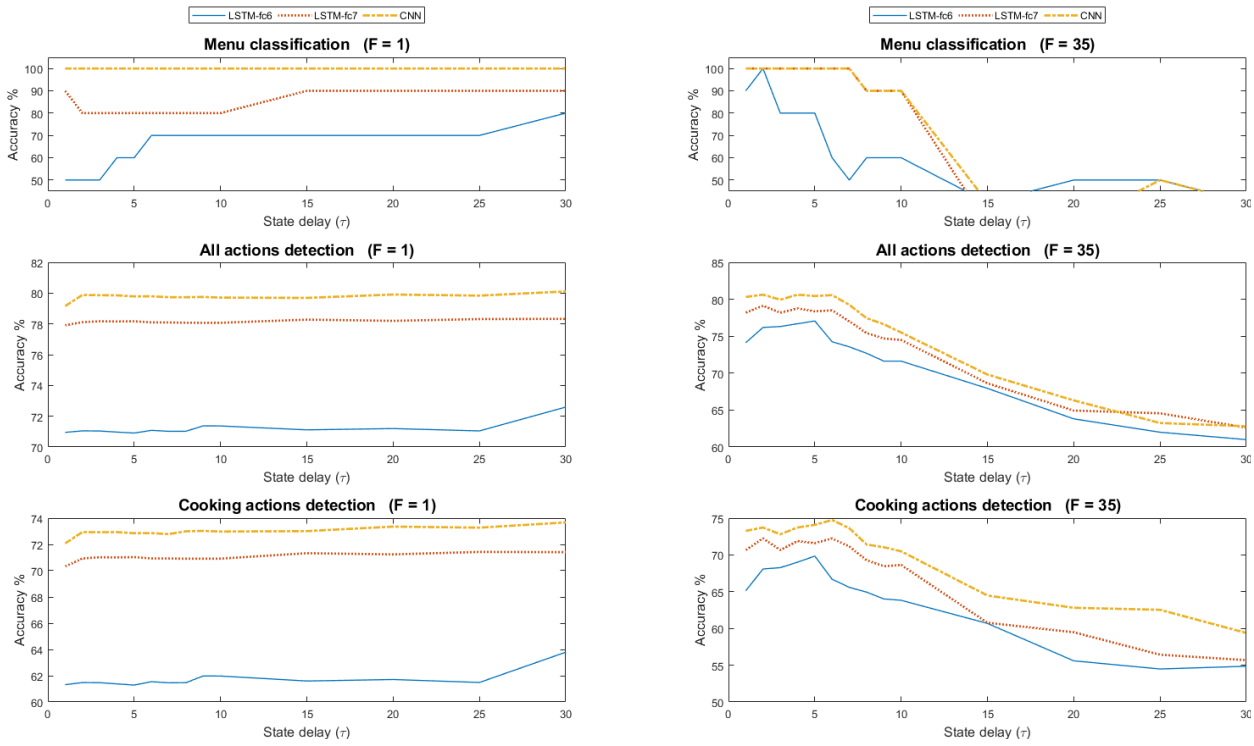


Figure 34: System performance with different state delays

when processing more video frames. However, using large state delays is only feasible with small frames steps, which is computationally expensive and only increases the accuracy slightly. The plots show that increasing the state delay and the frame step extremely reduced the accuracy of both action detection and menu classification. It could be seen from the plots that the best performance is achieved with a state delay of $\tau = 6$ and a frame step of $f = 35$, 100% for menu classification and 80.56% for all actions detection, which is also a good balance between computational cost and system performance.

5.5.4 Comparison with previous results

After testing our system over a range of factors, we compare our best achieved results (CNN features with common-sense grammar insertion dictionary, with a frame step of 35 and a

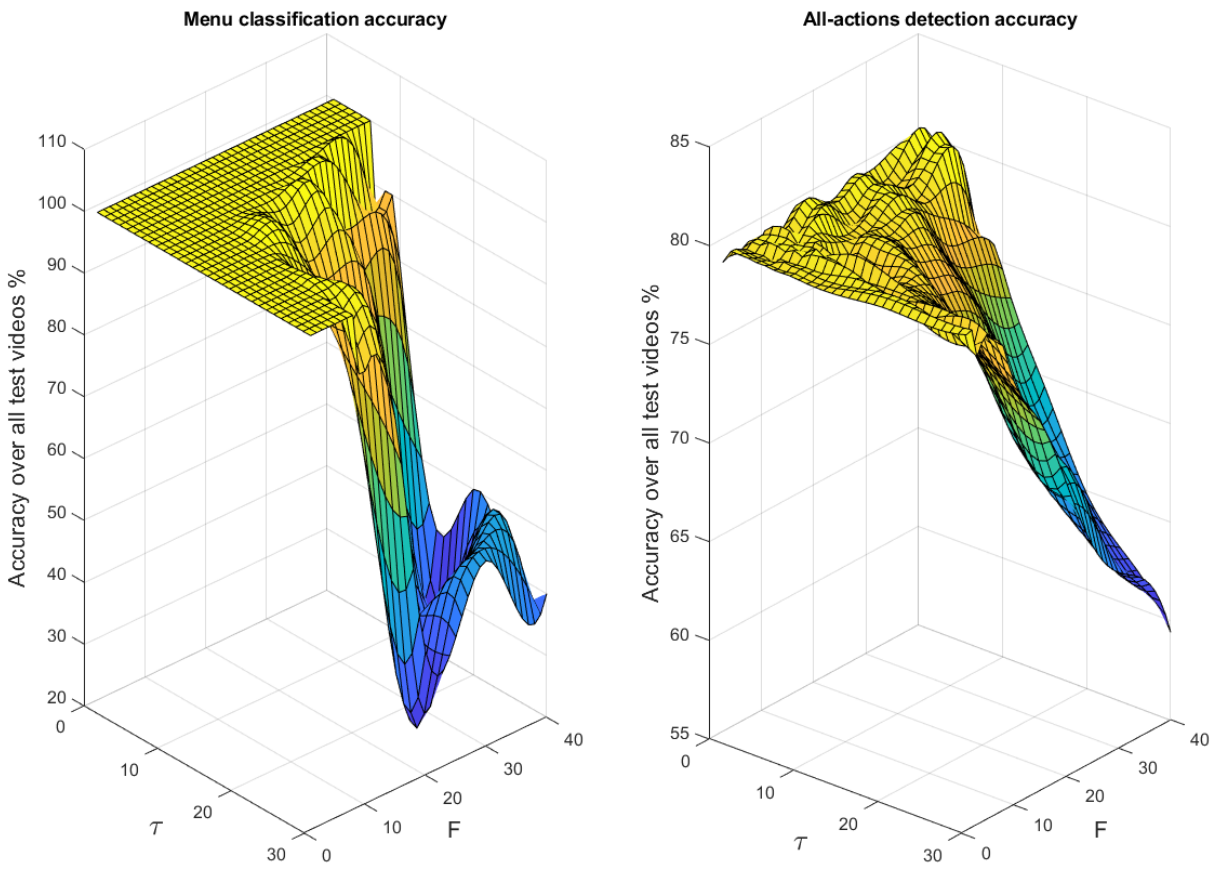


Figure 35: System performance over a grid of state delays τ and frame steps F

state delay of 6) to some of the best reported results in the literature. Although we have previously used only accuracy as an evaluation criterion, to compare our classification results to previous results published on the ACE data set, we also use precision, recall and f-measure criteria. The f-measure f is defined as [87],

$$f = 2 \,.\, \left(\frac{\text{precision} \;.\; \text{recall}}{\text{precision} + \text{recall}} \right) \tag{5.13}$$

where precision and recall are defined in terms of true-positives tp, false-positives fp and false-negatives fn as,

$$\text{precision} = \frac{tp}{tp + fp} \quad , \quad \text{recall} = \frac{tp}{tp + fn} \tag{5.14}$$

Action detection and menu classification evaluation results are summarized in Table 7 and 8; all the values in the tables are rounded percentages and dashes indicate that the results have not been reported by the publishers. The tables show results for our propsed system where a Viterbi algorithm is used with a common-sense grammar while processing one of every 35 frames and a state delay of $\tau = 6$. The table compares the results achieved by our proposed system to those reported and available in previous work; therefore, it contains some blank fields where the results are not investigated.

Table 7: Action detection and menu classification evaluation results summary

Approach	Precision	Recall	f-measure	Accuracy	Task
CNN	**86**	**76**	**81**	**81**	
LSTM (fc7)	82	74	78	79	
Scene-context [82]	68	68	68	72	All
Deep NN [83]	72	73	72	73	action
SVM-HOGV [84]	-	-	74	76	detection
Hybrid CNN & SBR [85]	-	-	-	70	
detection on summary [86]	-	-	-	80	
CNN				**100**	Menu
LSTM (fc7)				**100**	classification
Hybrid CNN & SBR [85]				10 - 30	

Table 7 shows that our proposed system outperforms previously reported approaches both on the action detection and menu classification tasks. For the menu classification task, the hybrid CNN & SBR approach [85] was the only approach that addressed the problem. However, the results reported are not accuracies but rather a rate of correct menu prediction. The hybrid CNN& SBR approach outputs multiple possible menu classes for each test video, then calculates the rate of correct menu prediction for each video and finally averages those rates to achieve the final rate of correct menu predictions for all test videos. This rate is

calculated per test video by counting the number of correct menu classes divided by the total number of possible recipes the system finds. For example, if the system outputs four possible menus and only one of them is the correct menu, the video is scored a rate of correctness of 25% In terms of the rate of correct menu predictions, we achieve a rate of 100% for our LSTM(fc7) and CNN approaches and a rate of 80% for the LSTM(fc6) approach compared to the reported rate of 18% for the hybrid CNN & SBR approach. The $10-30\%$ range of accuracy displayed in the table indicates the worst and best cases (only 1 correct menu prediction and three correct predictions respectively). The table and previous plots also show that our best performing system is the one that uses features extracted using a CNN alone (without an LSTM). Table 8 compares the per-action detection evaluation results of our CNN and LSTM-fc7 approaches to previously reported available results. The table shows that our systems precision outperforms the previous approaches for five actions and is second for four actions. In accuracy, our systems performs best for six actions and second for three actions.

Table 8: Precision and accuracy percentages results comparison for each cooking activity

		Cooking Action								
	Approach	break	mix	bake	turn	cut	boil	season	peel	none
Accuracy	CNN	**98**	**80**	**91**	16	58	56	96	96	**93**
	LSTM-fc7	89	78	89	10	63	48	**98**	**100**	92
	Deep NN [83]	35	45	69	01	52	51	65	38	69
	SVM-HOGV [84]	73	67	83	15	**93**	**79**	97	95	81
	CNN&SBR[85]	27	29	67	**34**	58	28	35	09	64
Precision	CNN	69	**67**	**83**	**95**	**98**	89	**98**	99	79
	LSTM-fc7	75	62	82	92	88	84	88	88	78
	SVM-HOGV [84]	**79**	61	76	56	55	**96**	88	**100**	**80**
	CNN&SBR[85]	44	67	74	77	87	61	89	72	65

6.0 GRAPH DECODING FOR NEXT ACTION PREDICTION IN VIDEOS

In this chapter, we address the problem of next action prediction. Assuming that a video is processed online or if the video stream stops where the system sees only part of the test video. We are interested in finding a scheme that is able of outputting a prediction for the next performed action. Our problem is thus stated as: given a training data set composed of a number of training videos, where each video is labeled with an overall class label and each frame of each video is labeled with an action label. And given a testing data set composed of a number of unlabeled videos and unlabeled video frames: if the test video is not fully delivered to the system, predict the next performed action after the last seen video frame. We use the same graph decoding approach described in the previous chapter to tackle this problem.

6.1 ACTION PREDICTION IN VIDEO SIGNALS

The problem of action detection, especially human action detection, has received a lot of attention from the research community over the past ten years. Having machines that are able to automatically detect the actions performed in a video clip has many applications especially in the field of improving the quality of life through health care and in surveillance. The problem of action prediction however, is still an ongoing research topic that is receiving a lot of interest due to its important applications like preventing crimes or traffic jams through surveillance or preventing traffic accidents in smart cars systems. Action prediction could be defined as being able to make a guess on the action performed in future frames. Multiple previous approaches [88, 89, 90, 91, 92, 93] has focused on attempting to guess the action

performed in video clips in an early stage. In other words, the problem has been defined as: given that an action is performed in a video clip, how to be able to guess that action when seeing only a small portion of the video clip, which could be viewed as action detection on early stages. This problem is very challenging mainly because actions on early stages are not discriminative enough to be classified as one class rather than the other.

Another definition of action prediction is to be able to make a guess on the next performed action given a sequence of actions that were performed in the past and present. This is the problem we try to address in this chapter, which is very different than the problem addressed in previous action prediction research. It also requires data sets that are constructed so that videos contain multiple labeled actions within the same video, rather than short video clips of single actions. Therefore, we continue to use the ACE data set to investigate this problem. The action prediction problem based on this definition, to our knowledge, has not been previously addressed, we present a novel graphical approach to tackle the problem.

6.2 VITERBI DECODING FOR NEXT ACTION PREDICTION

The graphical approach decribed in Sections 5.3 and 5.4 of the previous chapter not only offers action detection and menu classification, it also allows for next action prediction. Suppose that the video stream was interrupted at a certain frame or that the processing is performed online where video frames come in to the processor sequentially in time. Then as video frames are coming in, the processor would be able to predict the action performed in the coming frames before they come in. Next action prediction is achieved using Viterbi decoder and grammar matching, similar to menu classification. Since we are assuming that the frame stream stops before seeing the final action, it is important to relax the end state restriction condition in the termination step of the Viterbi process. So instead of forcing the algorithm to find the state that maximizes the path probability score within the end states only, the algorithm is free to find any state that maximizes the path probability.

The process of predicting the next action starts with applying the Viterbi algorithm to the available stream of classification scores to generate a Viterbi path of most probable

states. The next step is to match the generated path with a grammar in the dictionary. Once again, every single state in S is unique, therefore, only one state in a Viterbi path is sufficient to match the whole path to its corresponding grammar in the dictionary. Once a grammar is fetched from the dictionary, it is compared to the Viterbi path, where the location of the final state in the path is determined in the corresponding grammar; then the next action prediction would simply be the next action (state) in the grammar that comes after the final seen state in the Viterbi path, as illustrated in Figure 36.

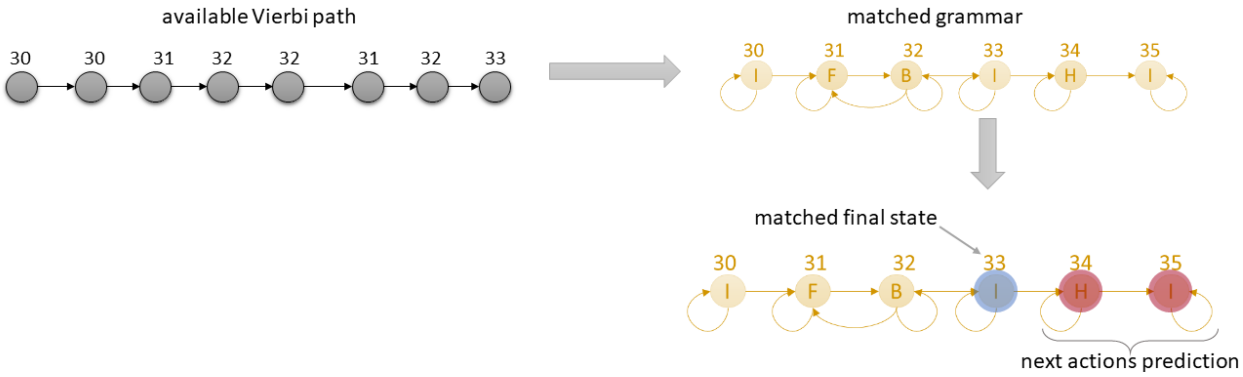


Figure 36: Next action prediction process

It is important to emphasize that the next action prediction is not a prediction of the action performed in the next frame, but rather a prediction of the next performed action; which could be in the immediate next frame or after a number of frames. The predicted action is found from the matched grammar by identifying the next new action. In other words, if the grammar allows for a state to transition to itself (which is always the case) so that the same action is performed throughout a number of frames, or if the grammar allows for the state to transition to a previous state so that two actions are alternating through a number of frames, then the next predicted action should not be chosen as neither the same action nor a previous one but rather the one that comes after both. This rule eliminates the possibility of the system to output more than one next action once a grammar is assigned to a sequence of input video frames. However, this may result in making prediction mistakes if

two actions are indeed alternating for a number of frames, which means that the prediction accuracy of our system is going to be under-evaluated.

A related point to consider is that even though two cooking menus could share a sequence of actions up to a certain video frame, the system will never confuse two menus or two next action predictions. That is because the grammar matching process for both menu classification and next action prediction is performed before mapping the numerical states back to letter states. And since every numerical state is unique and is associated with only one cooking menu and has its own set of next actions, then the system will not confuse multiple menus or multiple action predictions.

6.3 NEXT ACTION PREDICTION ASSESSMENT

After making a prediction on the next performed action as described in Section 6.2 and Figure 36, we need to assess the system's predictions. We report on the action prediction accuracy which we calculate by processing a percentage of each test video and comparing the system's next predicted actions to the true next performed actions in each video, then finally counting the successful predictions and dividing by the total number of test videos. We choose to report the accuracy as different percentages of the test videos are available since different test videos have different number of frames. The process of assessing the next action predictions is illustrated in Figure 37. The figure shows the action prediction and assessment steps on a test video labeled as "omelette-eggs" as only 40% of the video is available, when processing every 35^{th} video frame with a state delay of $\tau = 6$.

The first step in the illustrative example is to match the available Viterbi path to a menu grammar. The figure shows that the last seen state is numerical state 673 (which corresponds to label "no-action"). The next stage is to match this state to its corresponding state in the grammar. Notice that both the grammar and the Viterbi path exhibit the state delay effect, where no state is allowed to transit to a different state until 6 consecutive frames has the same state. As seen from the second stage of step 1, numeric state 673 is matched to its corresponding state in the delayed grammar (matched states are shaded in the figure).

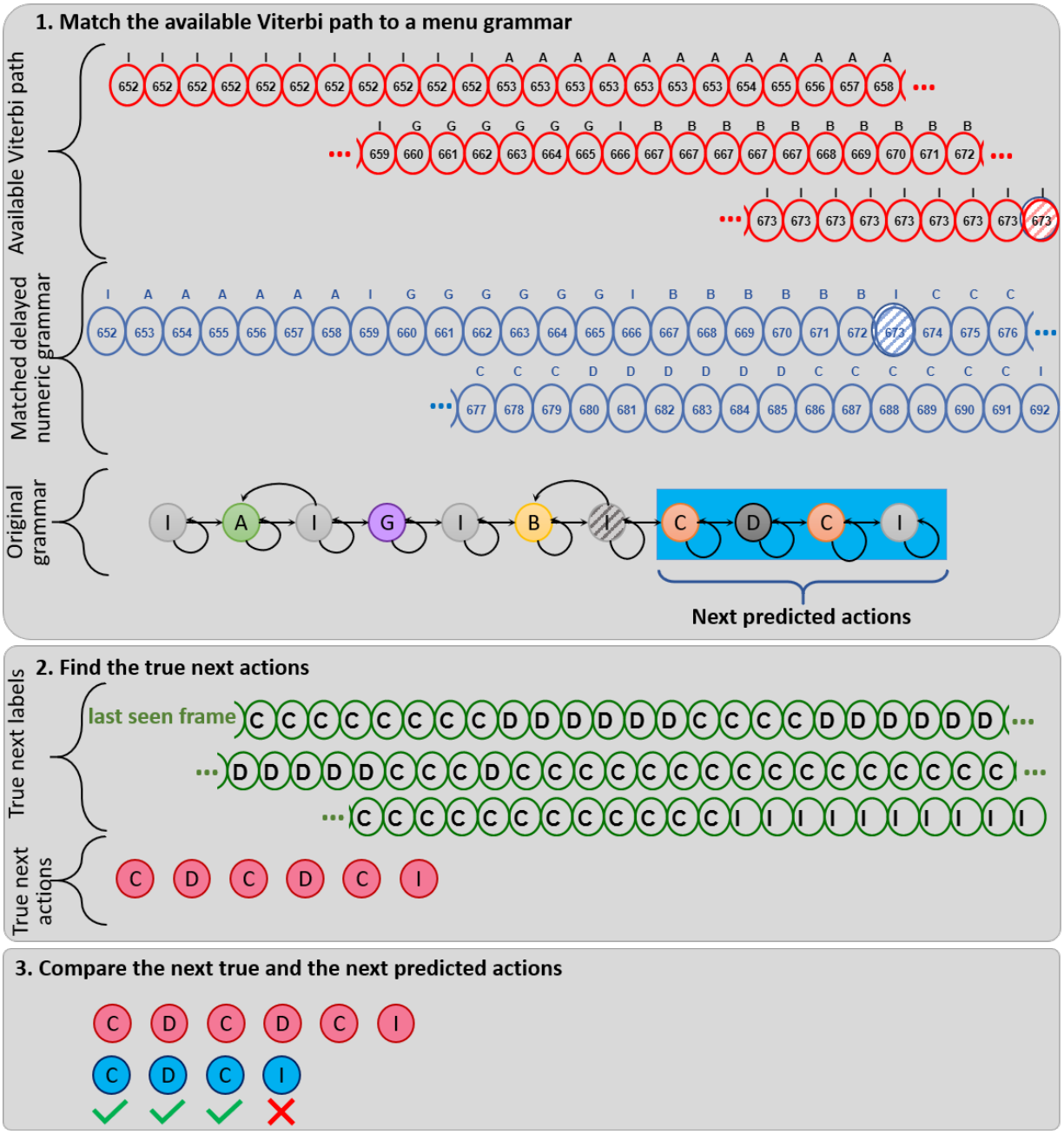


Figure 37: Illustration of next action prediction assessment steps

And in the final stage, the delayed grammar is mapped back to its original (non-delayed) version where the current state (I which corresponds to "no-action") is shaded and the next predicted actions are all the actions that come after shaded action I.

The second step in the assessment process is to determine the true next actions which is easily achieved by looking at the true test video's labels right after 40% of the video is seen. The green frame actions illustrated in the figure are the true next frame labels right after the last seen frame. The true next actions sequence is found by simply considering unique actions.

Finally, to assess the predictions, the true and predicted next actions as compared and a comparison score of 1 is assigned to a correct prediction and of 0 to an incorrect prediction. For the example shown in Figure 37 the test video is assigned a 1 for the first next action prediction and a 1 for the next second action prediction. Then the total prediction accuracy is calculated by taking the summation of all the comparison scores and dividing by the total number of test videos.

6.4 RESULTS AND DISCUSSION

Using the scheme illustrated in Figures 36, 37, we investigate the accuracy of the next first and second predicted actions when only a certain percentage of the video frames is available for processing. Figure 38 shows the prediction performance when processing test videos online for the case of the common-sense grammar insertion when processing every other 35^{th} frames with a state delay of $\tau = 6$. The plots show that like all other tasks, the CNN and LSTM-fc7 features result in better perfomance than the LSTM-fc6 features. It could be seen from the plots that for the CNN and LSTM-fc7 features, the system manages to predict the first next action with an accuracy higher than 30% even when less than 10% of the test video frames are available for processing. The accuracy of the prediction of the first next action (for all features) continues to increase above 50% as more than 60% of the video frames become available. The plots show that the performance of the next second action prediction is similar to that of the first next action. However, it could be seen that it is more difficult to correctly predict the next second performed action on early stages.

We further investigate the online performance of the system for the action detection and menu classification tasks. Figure 39 summarizes the online performance of the system

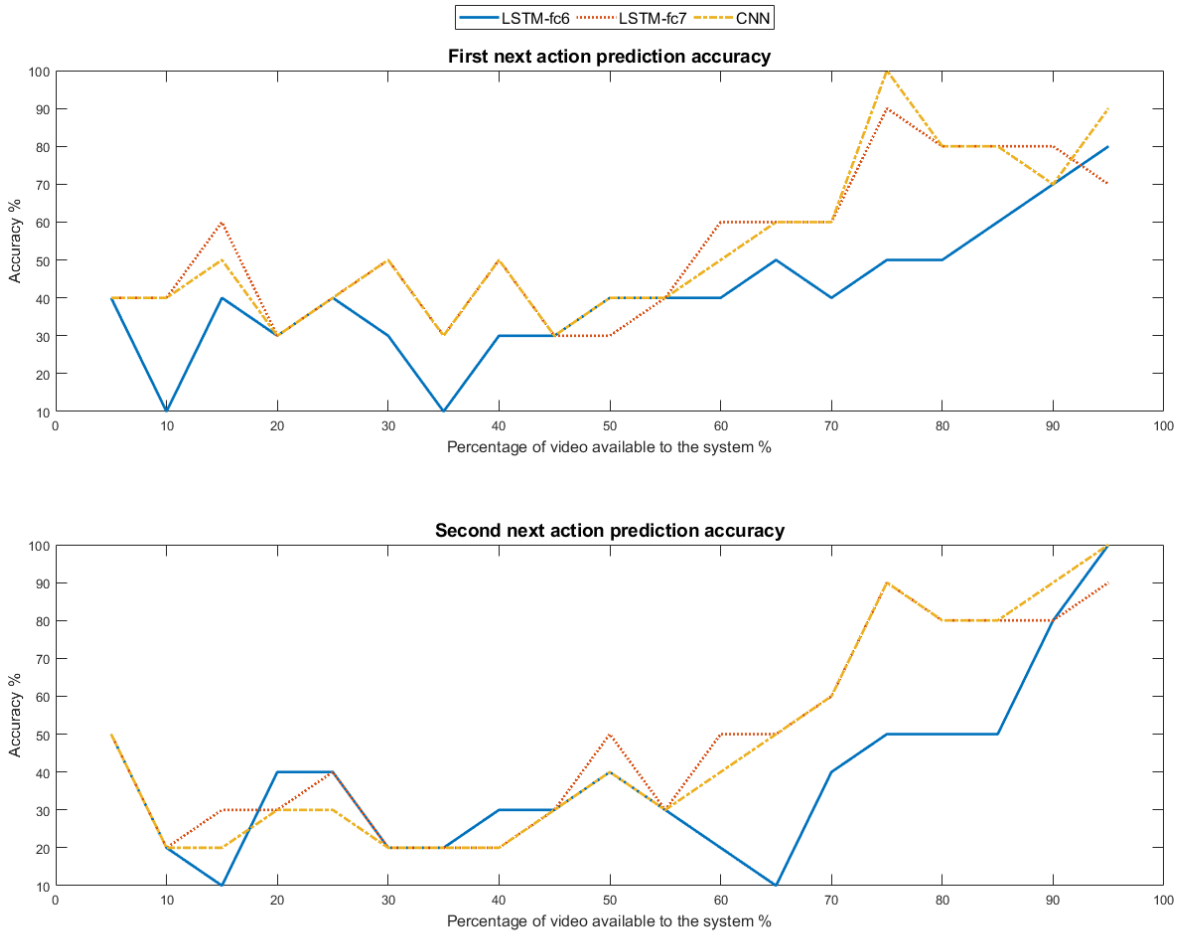


Figure 38: Next action prediction accuracy

for all three tasks (action detection, menu classification and next first and second action prediction). The plots show that the proposed system performs well for action detection and menu classification tasks even when it does not see the full video. It can be seen that once over 20% of the video is available, and when only processing every 35^{th} frame, the proposed system performance on action detection and menu classification is comparable to when it has access to all video frames, which means that the system performs well on the task of action detection and menu classification on an early stage.

The plots in Figure 39 show that action detection accuracy starts at 100% when less than 10% of video frames are available; this result is misleading, however, justified by the

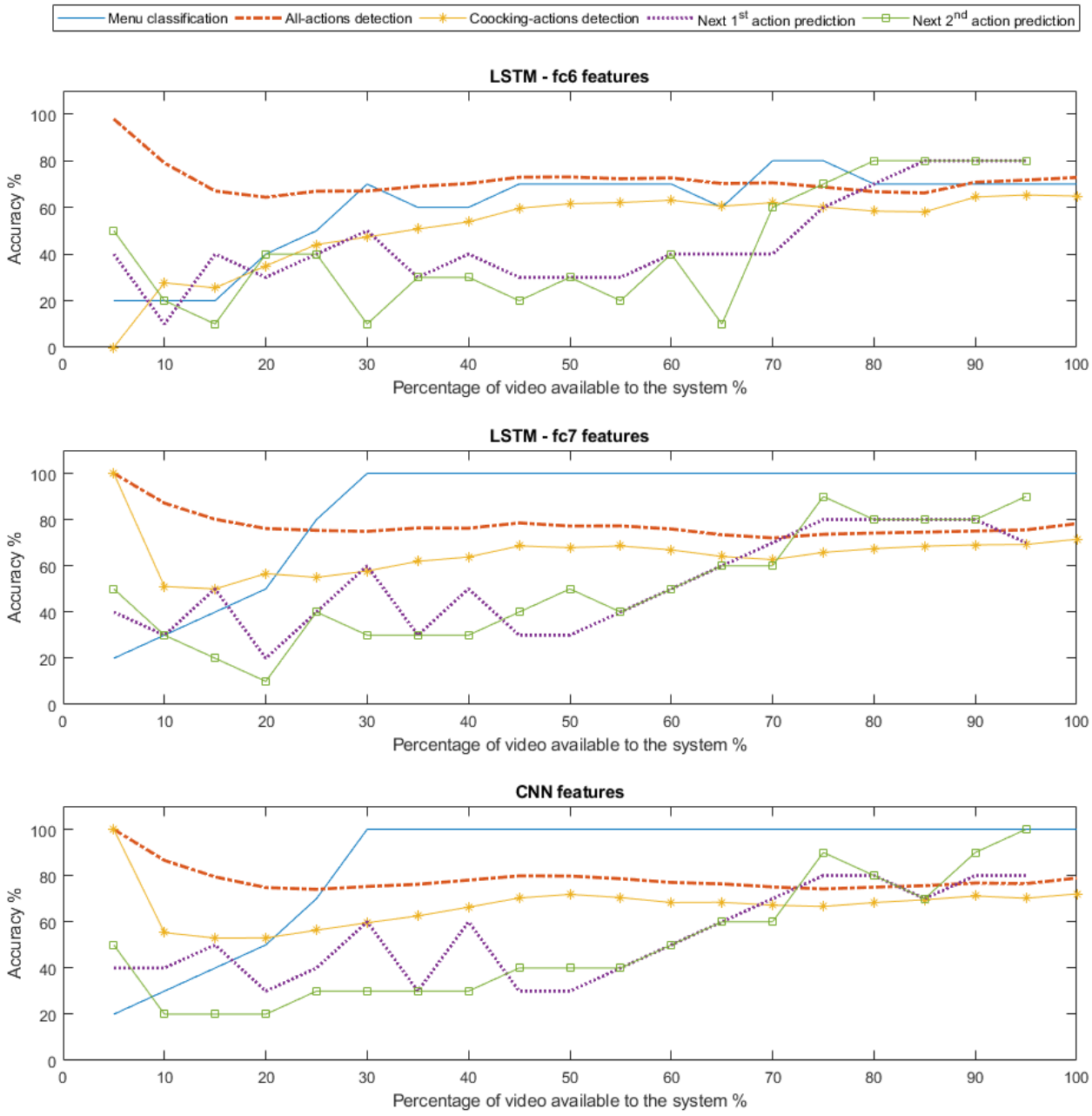


Figure 39: Online system performance

fact that all videos start with a no-action state (which is detected with a high accuracy as shown in the confusion matrix in Figure 29). Then as the system receives more frames

the accuracy drops close to the average accuracy. The plots show that menu classification and next action prediction are more sensitive to the percentage of seen frames than action detection. It can be seen that menu classification achieves a 100% accuracy when only over 30% of video frames are available. As for next action prediction, it could be seen that the system needs to process more than 60% of the video before being able to predict the next two actions with an accuracy above 50%. As the system sees more frames, it gains confidence in the next action predictions it makes and the prediction accuracy increases.

We further investigate the effect of the frame step and state delay on online processing. Figures 40, 41 show 3-D plots illustrating the system performance (action detection, menu classification and next first action prediction) with varying available video frames; while changing the frame steps and the state delays respectively. The plots, although hard to visualize in 2-D, show that increasing the frame step upto $F = 35$ enhances the online performance of the system for the menu classification task and slightly increase the accuracy of the action detection as shown in the top left and top right graphs of Figure 40 respectively. However, it doesn't have a clear effect on the online performance of the next action prediction task as seen from the rippling surface plots of the bottom graph of Figure 40. On the other hand increasing the state delay to more than $\tau = 10$ deteriorates the system's online performance for all tasks as shown in Figure 41. It is still more clear to visualize for the menu classification (left graph) and action detection (right graph), but it could still be seen in the next first action prediction task (bottom graph) as the prediction accuracy is as low as 50% when seeing 90% of the video with a state delay of $\tau = 10$ compared to 80% with $\tau = 6$. The plots of Figures 40, 41 therefore confirm our earlier conclusion that using a frame step of $F = 35$ with a state delay of $\tau = 6$ yields the best performance.

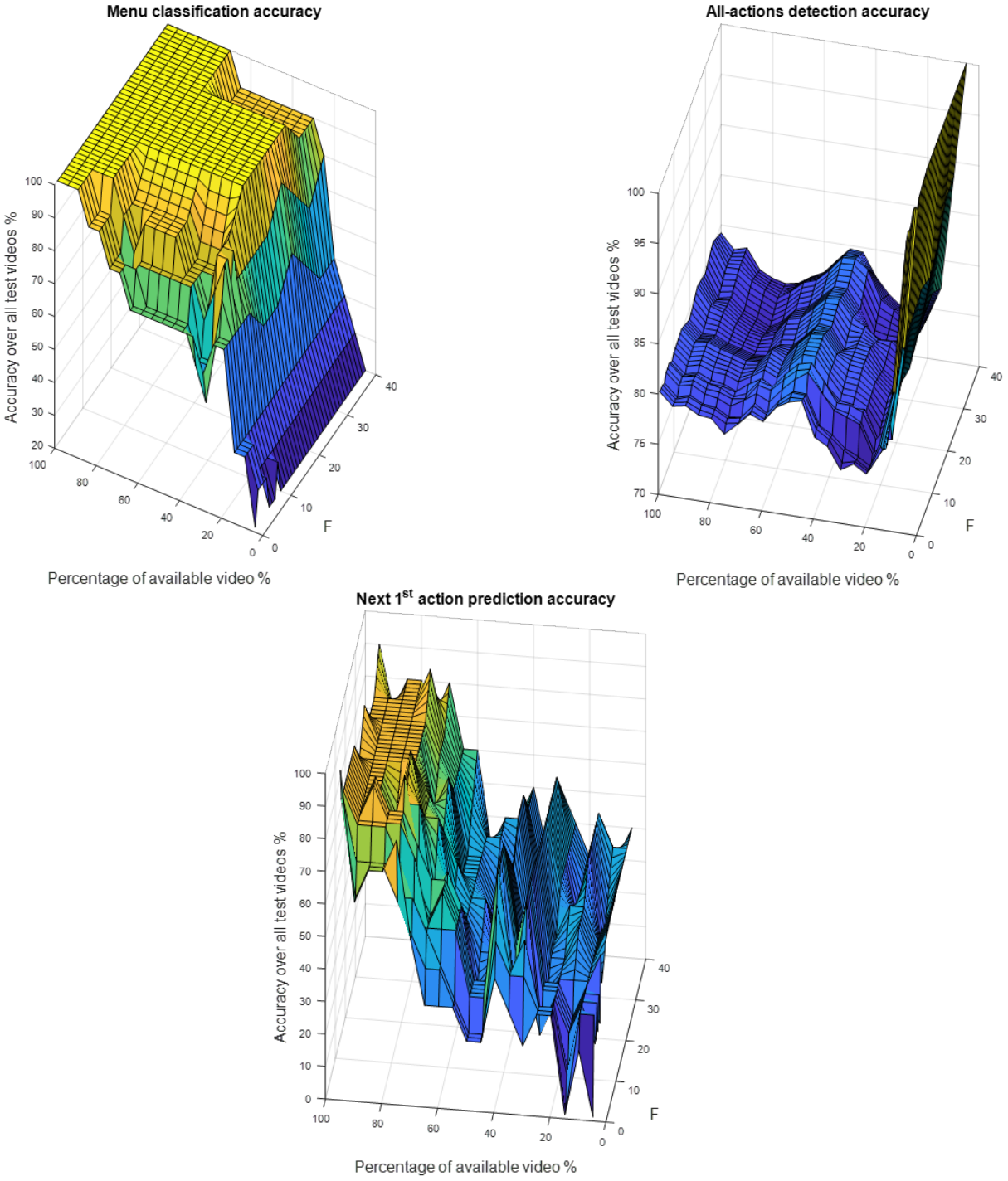


Figure 40: Online system performance as a function of changing frame steps

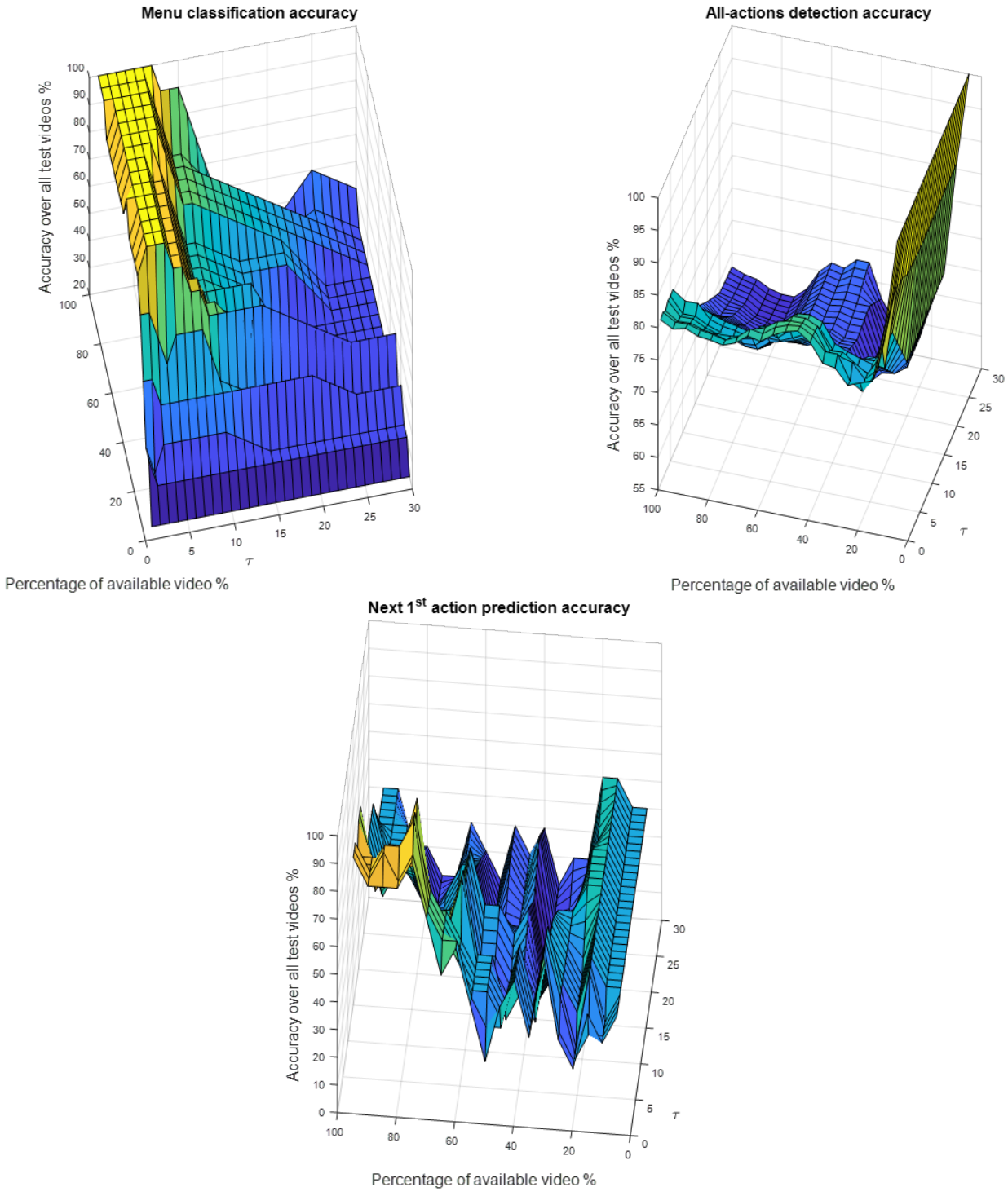


Figure 41: Online system performance as a function of changing state delays

7.0 RESEARCH SUMMARY

7.1 CONCLUSIONS

In this dissertation, we have developed a video analysis framework based on deep learning. The analysis of videos was performed on two major video comprehension tasks, video classification and action prediction in videos. The video classification task was split into two problems, video classification on one level (where the entire video is assigned a single class) and video classification on two levels (where not only the entire video is assigned a label, but also every video frame is assigned a label that reflects the action performed in the frame).

The one level classification problem was approached by standard deep learning techniques for image and video processing. A deep CNN was trained on still images and the trained network was used to extract the spatial features of the images. Two different sources of images were used to train the CNN; one is an independent data set of images with labels related to the video class and the other one is a subset of the video frames. Then to make a decision on the video's overall label, two approaches were investigated. One approach naively assumes that a video is a combination of independent image frames and a label was assigned to a video by averaging the classification scores of a number of randomly selected consecutive video frames. The second approach makes use of the temporal content of video streams and thus treats a video as a sequence of correlated frames. Shallow LSTM networks were trained on video spatial features extracted from video frames to learn the temporal features of videos. It was shown that LSTMs are very powerful in extracting the temporal content of videos, which results in a classification accuracy of 97% compared to just 73% when classifying videos based on independent frames (those classification results were achieved on a subset of the Sports-1M video data set). We also showed that using an independent data set for

training the CNN results in extracting better spatial features only when considering just the spatial content of videos, however, if the temporal content is to be taken into consideration, then training on a subset of the data set yields better classification accuracy.

The video classification on two levels and the action prediction problem were attended to using a combination of deep learning and classical machine learning techniques. Deep learning architectures, namely CNN and LSTM, were used to extract video frames features and a classical HMM was used to map the sequence of extracted features to a sequence of underlying video actions. A dictionary of action grammars was constructed from training videos and used to build the HMM model. This HMM model mapped video frames features into a sequence of video actions that agrees with the built dictionary, then the sequence of detected actions was mapped to a video class label classifying a video on two levels into actions and an overall class. Our approach successfully classifies videos into a single label with a classification accuracy of 100% while detecting frames actions with an accuracy of 81% on the ACE data set.

The action prediction task was also addressed using the hybrid deep network and HMM approach. Action prediction within videos is a new outgoing research topic, however it is addressed on segmented video clips in the literature. We addressed the problem on raw un-segmented video signals composed of multiple actions and proposed a scheme that is able of predicting the next performed action after a sequence of performed actions rather than predicting the current action in an early stage. Our approach was able of predicting the next performed action with an accuracy higher than 50% after seeing 60% of the ACE test videos frames, with an increasing prediction accuracy as more frames are accessed.

7.2 FUTURE WORK

Although an increasing number of large video data sets have become available over the past ten years, a large labeled video data set with multiple labels per video still does not exist. We recommend the use of unsupervised learning techniques to label large sets of videos into multiple actions to address the lack of multiple actions video data sets.

We also suggest to go back to the idea of using different data sets for training and testing, like what we did in Chapter 3. However, the two different data set are all video sets and are used for more than just video classification, but further for action detection and prediction. We suggest using a training set that is a combination of the kitchen cooking actions that come from three different data sets ADL65 [70], the URADL [71] and a part of the CMU-ADL data set [73]. The first two training sets are composed of video clips each representing one cooking action, while the third one is composed of full lengthed videos with multiple actions (and multiple annotations) per video. The testing data set is the full CMU-ADL data set that only provides a single menu label for each video.

We propose to use the CMU-ADL set to build a dictionary of actions for cooking menus. Then we could apply our proposed hybrid CNN-HMM system to examine the rich CMU-ADL data for action detection, menu classification plus next action prediction. Our proposed system would be evaluated on the part of the CMU-ADL that provides labels for all frames. The system could be then used to segment single labeled videos in the rich and growing CMU-ADL set into multiple action labels to help annotate more cooking videos with sub-labels.

BIBLIOGRAPHY

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[2] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European Conference on Computer Vision*. Springer, 2014, pp. 740–755.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[4] H. Liu, R. Feris, and M. Sun, "Benchmarking human activity recognition," *CVPR Tutorial, CVPR*, 2012.

[5] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *CVPR*, 2014.

[6] A. Shimada, K. Kondo, D. Deguchi, G. Morin, and H. Stern, "Kitchen scene context based gesture recognition: A contest in icpr2012," in *Advances in depth image analysis and applications*. Springer, 2013, pp. 168–185. [Online]. Available: http://www.murase.m.is.nagoya-u.ac.jp/KSCGR/

[7] C. A. Poynton, *A technical introduction to digital video*. John Wiley & Sons, Inc., 1996.

[8] D. Brezeale and D. J. Cook, "Automatic video classification: A survey of the literature," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 38, no. 3, pp. 416–430, 2008.

[9] C. Lu, M. S. Drew, and J. Au, "An automatic video classification system based on a combination of hmm and video summarization," *International Journal of Smart Engineering System Design*, vol. 5, no. 1, pp. 33–45, 2003.

[10] M. K. Geetha and S. Palanivel, "Hmm based automatic video classification using static and dynamic features," in *Conference on Computational Intelligence and Multimedia Applications, 2007. International Conference on*, vol. 3. IEEE, 2007, pp. 277–281.

[11] J. Wang, C. Xu, and E. Chng, "Automatic sports video genre classification using pseudo-2d-hmm," in *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on*, vol. 4. IEEE, 2006, pp. 778–781.

[12] X. Gibert, H. Li, and D. Doermann, "Sports video classification using hmms," in *Multimedia and Expo, 2003. ICME'03. Proceedings. 2003 International Conference on*, vol. 2. IEEE, 2003, pp. II–345.

[13] L. Xie, S.-F. Chang, A. Divakaran, and H. Sun, "Structure analysis of soccer video with hidden markov models," in *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*, vol. 4. IEEE, 2002, pp. IV–4096.

[14] J. Huang, Z. Liu, Y. Wang, Y. Chen, and E. K. Wong, "Integration of multimodal features for video scene classification based on hmm," in *Multimedia Signal Processing, 1999 IEEE 3rd Workshop on*. IEEE, 1999, pp. 53–58.

[15] S. Eickeler and S. Muller, "Content-based video indexing of tv broadcast news using hidden markov models," in *icassp*. IEEE, 1999, pp. 2997–3000.

[16] A. Girgensohn and J. Foote, "Video classification using transform coefficients," in *Acoustics, Speech, and Signal Processing, 1999. Proceedings., 1999 IEEE International Conference on*, vol. 6. IEEE, 1999, pp. 3045–3048.

[17] J. Nam, M. Alghoniemy, and A. H. Tewfik, "Audio-visual content-based violent scene characterization," in *Image Processing, 1998. ICIP 98. Proceedings. 1998 International Conference on*, vol. 1. IEEE, 1998, pp. 353–357.

[18] A. Hauptmann, R. Yan, Y. Qi, R. Jin, M. G. Christel, M. Derthick, M.-y. Chen, R. Baron, W.-H. Lin, and T. D. Ng, "Video classification and retrieval with the informedia digital video library system," 2002.

[19] B. T. Truong and C. Dorai, "Automatic genre identification for content-based video categorization," in *Pattern Recognition, 2000. Proceedings. 15th International Conference on*, vol. 4. IEEE, 2000, pp. 230–233.

[20] Z. Rasheed, Y. Sheikh, and M. Shah, "Semantic film preview classification using low-level computable features," in *3rd International Workshop on Multimedia Data and Document Engineering (MDDE-2003)*, 2003.

[21] X. Yuan, W. Lai, T. Mei, X.-S. Hua, X.-Q. Wu, and S. Li, "Automatic video genre categorization using hierarchical svm," in *2006 International Conference on Image Processing*. IEEE, 2006, pp. 2905–2908.

[22] G. Hong, B. Fong, and A. Fong, "An intelligent video categorization engine," *Kybernetes*, vol. 34, no. 6, pp. 784–802, 2005.

[23] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[24] K. Simonyan and A. Zisserman, "Two-stream convolutional networks for action recognition in videos," in *Advances in neural information processing systems*, 2014, pp. 568–576.

[25] M. Baccouche, F. Mamalet, C. Wolf, C. Garcia, and A. Baskurt, "Sequential deep learning for human action recognition," in *International Workshop on Human Behavior Understanding*. Springer, 2011, pp. 29–39.

[26] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 2625–2634.

[27] S. E. Kahou, X. Bouthillier, P. Lamblin, C. Gulcehre, V. Michalski, K. Konda, S. Jean, P. Froumenty, Y. Dauphin, N. Boulanger-Lewandowski *et al.*, "Emonets: Multimodal deep learning approaches for emotion recognition in video," *Journal on Multimodal User Interfaces*, vol. 10, no. 2, pp. 99–111, 2016.

[28] J. Yue-Hei Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici, "Beyond short snippets: Deep networks for video classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 4694–4702.

[29] S. Ji, W. Xu, M. Yang, and K. Yu, "3d convolutional neural networks for human action recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 1, pp. 221–231, 2013.

[30] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas, "Supervised machine learning: A review of classification techniques," 2007.

[31] P. Kamavisdar, S. Saluja, and S. Agrawal, "A survey on image classification approaches and techniques," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 2, no. 1, pp. 1005–1009, 2013.

[32] C. Bishop, "Bishop pattern recognition and machine learning," 2007.

[33] I. G. Y. Bengio and A. Courville, "Deep learning," 2016, book in preparation for MIT Press. [Online]. Available: http://www.deeplearningbook.org

[34] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.

[35] R. Rojas, *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

[36] J. Bouvrie, "Notes on convolutional neural networks," 2006.

[37] P. Sollich and A. Krogh, "Learning with ensembles: How overfitting can be useful," in *Advances in neural information processing systems*, 1996, pp. 190–196.

[38] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[39] A. El-Jaroudi and J. Makhoul, "A new error criterion for posterior probability estimation with neural nets," in *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*. IEEE, 1990, pp. 185–192.

[40] T. M. Mitchell *et al.*, "Machine learning. wcb," 1997.

[41] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[42] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting." *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[43] C. M. Bishop, *Neural networks for pattern recognition*. Oxford university press, 1995.

[44] D. C. Plaut *et al.*, "Experiments on learning by back propagation." 1986.

[45] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[46] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits," 1998.

[47] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[48] D. L. Ruderman, "The statistics of natural images," *Network: computation in neural systems*, vol. 5, no. 4, pp. 517–548, 1994.

[49] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv preprint arXiv:1603.07285*, 2016.

[50] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010, pp. 807–814.

[51] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *arXiv preprint arXiv:1312.6229*, 2013.

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

[53] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on.* IEEE, 2009, pp. 248–255.

[54] A. Vedaldi and K. Lenc, "Matconvnet – convolutional neural networks for matlab," in *Proceeding of the ACM Int. Conf. on Multimedia*, 2015.

[55] L. Medsker and L. C. Jain, *Recurrent neural networks: design and applications.* CRC press, 1999.

[56] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[57] A. Graves, "Supervised sequence labelling with recurrent neural networks. 2012," *ISBN 9783642212703. URL http://books. google. com/books.*

[58] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems*, 2016.

[59] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[60] A. Graves, S. Fernández, and J. Schmidhuber, "Bidirectional lstm networks for improved phoneme classification and recognition," in *International Conference on Artificial Neural Networks.* Springer, 2005, pp. 799–804.

[61] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm and other neural network architectures," *Neural Networks*, vol. 18, no. 5-6, pp. 602–610, 2005.

[62] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning spatiotemporal features with 3d convolutional networks," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 4489–4497.

[63] S. Zha, F. Luisier, W. Andrews, N. Srivastava, and R. Salakhutdinov, "Exploiting image-trained cnn architectures for unconstrained video classification," *arXiv preprint arXiv:1503.04144*, 2015.

[64] Z. Wu, X. Wang, Y.-G. Jiang, H. Ye, and X. Xue, "Modeling spatial-temporal clues in a hybrid deep learning framework for video classification," in *Proceedings of the 23rd ACM international conference on Multimedia*. ACM, 2015, pp. 461–470.

[65] S. Abu-El-Haija, N. Kothari, J. Lee, P. Natsev, G. Toderici, B. Varadarajan, and S. Vijayanarasimhan, "Youtube-8m: A large-scale video classification benchmark," *arXiv preprint arXiv:1609.08675*, 2016.

[66] K. Soomro, A. R. Zamir, and M. Shah, "Ucf101: A dataset of 101 human actions classes from videos in the wild," *arXiv preprint arXiv:1212.0402*, 2012.

[67] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[68] P. Turaga, R. Chellappa, V. S. Subrahmanian, and O. Udrea, "Machine recognition of human activities: A survey," *IEEE Transactions on Circuits and Systems for Video technology*, vol. 18, no. 11, p. 1473, 2008.

[69] R. Poppe, "A survey on vision-based human action recognition," *Image and vision computing*, vol. 28, no. 6, pp. 976–990, 2010.

[70] M. Rohrbach, S. Amin, M. Andriluka, and B. Schiele, "A database for fine grained activity detection of cooking activities," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, 2012, pp. 1194–1201.

[71] R. Messing, C. Pal, and H. Kautz, "Activity recognition using the velocity histories of tracked keypoints," in *ICCV '09: Proceedings of the Twelfth IEEE International Conference on Computer Vision*. Washington, DC, USA: IEEE Computer Society, 2009. [Online]. Available: http://www.cs.rochester.edu/u/rmessing/uradl/

[72] A. Hashimoto, S. Mori, M. Iiyama, and M. Minoh, "Kusk object dataset: Recording access to objects in food preparation," in *2016 IEEE International Conference on Multimedia Expo Workshops (ICMEW)*, July 2016.

[73] F. D. la Torre Frade, J. K. Hodgins, A. W. Bargteil, X. M. Artal, J. C. Macey, A. C. I. Castells, and J. Beltran, "Guide to the carnegie mellon university multimodal activity (cmu-mmac) database," Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-22, April 2008. [Online]. Available: http://kitchen.cs.cmu.edu/

[74] M. N. N. T. Team, "Neural network toolbox model for vgg-19 network," https://www.mathworks.com/matlabcentral/fileexchange/61734-neural-network-toolbox-model-for-vgg-19-network, March 2018.

[75] C. Bhadane, H. Dalal, and H. Doshi, "Sentiment analysis: Measuring opinions," *Procedia Computer Science*, vol. 45, pp. 808–814, 2015.

[76] D. Jurafsky and J. H. Martin, *Speech and language processing*. Pearson London, 2014, vol. 3.

[77] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[78] B. Ni, V. R. Paramathayalan, T. Li, and P. Moulin, "Multiple granularity modeling: A coarse-to-fine framework for fine-grained action analysis," *International Journal of Computer Vision*, vol. 120, no. 1, pp. 28–43, 2016.

[79] B. Ni, P. Moulin, and S. Yan, "Pose adaptive motion feature pooling for human action analysis," *International Journal of Computer Vision*, vol. 111, no. 2, pp. 229–248, 2015.

[80] N. T. Hung, J. Y. Kim *et al.*, "Gesture recognition in cooking video based on image features and motion features using bayesian network classifier," in *Emerging Trends in Image Processing, Computer Vision and Pattern Recognition*. Elsevier, 2015, pp. 379–392.

[81] W. Ohyama, S. Hotta, and T. Wakabayashi, "Spatiotemporal auto-correlation of grayscale gradient with importance map for cooking gesture recognition," in *Pattern Recognition (ACPR), 2015 3rd IAPR Asian Conference on*. IEEE, 2015, pp. 166–170.

[82] S. Bansal, S. Khandelwal, S. Gupta, and D. Goyal, "Kitchen activity recognition based on scene context," in *Image Processing (ICIP), 2013 20th IEEE International Conference on*. IEEE, 2013, pp. 3461–3465.

[83] J. Monteiro, R. Granada, R. C. Barros, and F. Meneguzzi, "Deep neural networks for kitchen activity recognition," in *2017 International Joint Conference on Neural Networks (IJCNN)*, May 2017, pp. 2048–2055.

[84] S. Kojima, W. Ohyama, and T. Wakabayashi, "Gesture recognition based on spatiotemporal histogram of oriented gradient variation," in *Informatics, Electronics and Vision & 2017 7th International Symposium in Computational Medical and Health Technology (ICIEV-ISCMHT), 2017 6th International Conference on*. IEEE, 2017, pp. 1–4.

[85] R. Granada, R. F. Pereira, J. Monteiro, R. Barros, D. Ruiz, and F. Meneguzzi, "Hybrid activity and plan recognition for video streams," in *The AAAI 2017 Workshop on Plan, Activity, and Intent Recognition*, 2017.

[86] F. Hussein and M. Piccardi, "V-jaune: A framework for joint action recognition and video summarization," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 13, no. 2, p. 20, 2017.

[87] D. L. Olson and D. Delen, *Advanced data mining techniques*. Springer Science & Business Media, 2008.

[88] M. S. Ryoo, "Human activity prediction: Early recognition of ongoing activities from streaming videos," in *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE, 2011, pp. 1036–1043.

[89] T. Lan, T.-C. Chen, and S. Savarese, "A hierarchical representation for future action prediction," in *European Conference on Computer Vision*. Springer, 2014, pp. 689–704.

[90] Y. Kong, S. Gao, B. Sun, and Y. Fu, "Action prediction from videos via memorizing hard-to-predict samples." in *AAAI*, 2018.

[91] Y. Kong, Z. Tao, and Y. Fu, "Deep sequential context networks for action prediction," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 1473–1481.

[92] K. Li and Y. Fu, "Prediction of human activity by discovering temporal sequence patterns," *IEEE transactions on pattern analysis and machine intelligence*, vol. 36, no. 8, pp. 1644–1657, 2014.

[93] V. Vukotić, S.-L. Pintea, C. Raymond, G. Gravier, and J. C. van Gemert, "One-step time-dependent future video frame prediction with a convolutional encoder-decoder neural network," in *International Conference on Image Analysis and Processing*. Springer, 2017, pp. 140–151.