ACTIVITY RECOGNITION IN VIDEOS USING DEEP LEARNING

by

Mahesh R. Shanbhag

APPROVED BY SUPERVISORY COMMITTEE:

Dr. Vibhav G. Gogate, Chair

Dr. R. Chandrasekaran

Dr. Nicholas Ruozzi

This thesis is dedicated

to my wife Apoorva, to my parents Ramaray and Mangala Shanbhag,

to my friends, for the ever lasting support and love they give me.

ACTIVITY RECOGNITION IN VIDEOS USING DEEP LEARNING

by

MAHESH R. SHANBHAG, BS

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

MASTERS OF SCIENCE IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

May 2018

ACKNOWLEDGMENTS

The author would like to thank Dr. Vibhav Gogate and Dr. Tahrima Rahman for giving me an opportunity to work with them and for their continuous guidance and support. I would also like to thank Dr. R. Chandrasekaran and Dr. Nichols Ruozzi for encouraging me by being on my supervisory committee.

Specials thanks to my colleagues, Chiradeep Roy and Pracheta Sahoo for all the productive brainstorming sessions and helpful suggestions.

I would also like to thank my lovely wife Apoorva, my parents Ramaray and Mangala Shanbhag for their continued support towards my Masters's Thesis.

May 2018

ACTIVITY RECOGNITION IN VIDEOS USING DEEP LEARNING

Mahesh R. Shanbhag, MS
The University of Texas at Dallas, 2018

Supervising Professor: Dr. Vibhav G. Gogate, Chair

Automatically recognizing activities in a video is a long standing goal of computer vision and artificial intelligence. Recently, breakthroughs in deep learning have revolutionized the field of computer vision and today deep models can solve low-level tasks such as image classification and object detection more accurately than humans and even highly trained (human) experts. However, inferring high-level activities from low-level information such as objects in a video is a difficult task because the objects interacting with humans can be too small or similar activities might be captured at different spatial locations or angles. In this thesis, we propose an effective and efficient supervised learning model for solving this difficult task by leveraging advanced deep learning architectures. Our key idea is to formulate activity recognition as a multi-label classification problem in which the input is a set of frames (a video) and the output is an assignment of most probable labels to five components of each activity: *action*, *tool*, *object*, *source* and *target* at each frame. We begin with a network pre-trained on objects appearing in a large image classification dataset and then modify it with an additional layer that helps us solve the much harder multi-label classification problem. Then, we tune and train this new network to our video data by presenting each labeled frame in the video as input to the network. We train, evaluate and benchmark the model using a popular Cooking activities dataset and also interpret the learned model by visualizing the network at various levels of hierarchy.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Recognizing objects in high resolution images is a challenging task. Typically, these images yield significant improvement in accuracy over low resolution ones but processing them is computationally intractable. For example, consider building a fully connected neural network (FCN) in which the input is a 1000×1000 image with 3 color channels. Thus, the input layer is a vector having 3 million elements. If the first hidden layer has 1000 units, each unit fully connected to the input layer, the network will have over 3 billion parameters. Learning large number of parameters from data is practically infeasible because fundamental machine learning principles tell us that we would need a large amount of annotated (labeled) data to avoid *overfitting*, namely since the number of labeled examples is much smaller than the number of parameters, the learned model will exhibit high accuracy on training data but would not generalize well (exhibit poor accuracy) to unseen examples.

Deep Convolutional networks circumvent this computational difficulty by leveraging the *parameter sharing* (also called *parameter tying*) idea. They force several units in the same layer to have the same parameters, which greatly reduces the number of parameters that need to learned. This motivates us to design much deeper and wider networks since the computational cost is much smaller than FCNs. Today, these deep networks with corresponding advances in deep learning algorithms are able to achieve incredibly low error rates on various computer vision tasks such as image classification, object detection and object localization. For instance, the Res-Net architecture (Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian, 2016) was able to achieve top-5 error rate of 3.57% and won the 2015 ILSVRC image classification challenge. Depending on their skill and expertise, humans generally hover around a 5-10% error rate. This suggests that convolution network architectures are very well suited for visual recognition tasks.

In this thesis, we leverage advances in deep learning and architectures to design a model that automatically recognizes activities from high resolution videos. This task is much harder than low-level tasks such as object detection in images because we have to extract semantic relations between objects and their relative position in various frames in the video (a video is a sequence of images called frames). Thus, even though object detection is accurate, their relationship and transitions from one frame to another may not be discovered correctly and as a result the accuracy of activity recognition be quite low.

We address this problem by decomposing it as follows. We formulate activity recognition as a multi-label classification problem in which the input is a set of frames (a video) and the output is an assignment of most probable labels to five components of each activity: *action*, *tool*, *object*, *source* and *target* at each frame. We begin with a network pre-trained on objects appearing in a large image classification dataset and then modify it with an additional layer that helps us solve the much harder multi-label classification problem. Then, we tune and train this new network to our video data by presenting each labeled frame in the video as input to the network. At test time, our network assigns most probable labels to each unlabeled video frame. We envision these labels to be stored in a probabilistic database, which can then be used to recognize high level activities such as cooking and sub-activities such as cutting by performing probabilistic inference over the database.

We use the pre-trained GoogLe Net architecture (Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, 2015) which won the 2014 ILSVRC image classification challenge. This architecture uses 8.9% ($\sim$6.7M parameters) fewer parameters than the winning 2015 architecture. As such, this architecture performs $\sim$1.5B operations at inference time which makes it suitable to use on large data sets at reasonable costs. The network uses a special module called *Inception* which performs all convolutions in parallel and lets the model pick the best one based on the data (M. Lin, Q. Chen, and S. Yan., 2013). Each frame of the

video is modeled by this convolutional network pre-trained on 1.2M+ images with category labels. Tuning this network to learn and detect activities in video scenes is easier, quicker and computationally inexpensive.

Ideally, a video model should allow processing of variable length input sequences, and also provide for multi-labeled outputs, including prediction of actions and objects that go beyond the traditional one-vs-all prediction. In this thesis we propose an efficient and novel method for visual recognition and activity prediction that is end-to-end trainable. To date, deep convolutional models for video processing have successfully considered learning of 3-D spatio-temporal filters over raw sequence data (S. Ji, W. Xu, M. Yang, and K. Yu., 2013), and learning of frame-to-frame representations which incorporate instantaneous optic flow or trajectory-based models aggregated over fixed windows or short video segments (A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei., 2014). Such models explore two extrema of perceptual time-series representation learning: either learn a fully general time-varying weighting, or apply simple temporal pooling. Following the same principles that motivates current deep convolutional models, we advocate for video activity recognition by recognizing the elements *action*, *tool*, *object*, *source/target* that implicitly composes an activity.

We show here that this frame-to-frame convolution model that has been previously employed can provide significant improvement in detecting activities when used for multi-label classification. We explore our proposed model by end-to-end mapping of an image to the elements that composes the activity. The resulting model can be trained end-to-end on large-scale video datasets, and even with modest training provides competitive results. We use the MPII Cooking activities dataset. This dataset has 225 annotated cooking videos. The annotations available are *startFrame*, *endFrame*, *activity*, *tool*, *object* and *source/target*. Each video is decomposed into frames where each frame is $\frac{1}{24}^{\text{th}}$ of a second and is annotated with the corresponding annotation. This method allows us to generate ample training data

to learn and refine the visual representation. Each frame of the video is forward propagated through the model and the set of activity elements are selected using *sigmod cross entropy loss* cost function. At the same time the confidence level for each activity element is learned. At test time each frame is labeled with action elements depending on their confidence level.

Our experiments show that this model has much smaller training time and memory overhead relative to the size of the dataset. Moreover, the gain gets larger as the number of activity elements increases. Our experiments also show that adaptive weight learning algorithm converges much faster than the stochastic gradient descent algorithm.

The rest of the thesis is organized as follows. In chapter 2, we present the background on fully connected and convolutional neural networks. In chapter 3, we present our main contributions - activity detection in videos. In chapter 4, we present experimental results. In chapter 5, we will try to interpret and visualize the learned model. In chapter 6 we explain the tools used to design and learn the model. Finally, we conclude in chapter 7 with a brief summary and provide avenues for future work.

CHAPTER 2

BACKGROUND

In this chapter, we review four commonly used deep learning architectures: Fully Connected Networks (FCN), Convolution Neural Networks (CNN), GoogLe Net and Inception Model (Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, 2015).

2.1 Fully Connected Networks

Fully Connected Network (FCN) is a robust approach for approximating real valued, discrete valued or vector valued functions. As shown in Figure 2.1, a typical FCN system is a composition of layers of nodes stacked over each other. A single layer is a composition of disjoint set of nodes. Each node in turn (i.e. circle) corresponds to the output of a single network unit. The first layer at the bottom is the *input layer* whose input is a vector of input features. The input features are real valued. The last layer is the *output layer* which is a set of disjoint units used for prediction or classification. The layers between the input and output layer are called *hidden layers*, because their output is available within the network and is not global to the network. A cost/loss function is associated with the *output layer* to measure the similarity/difference between the output of the learned model and the outputs in the training data.

Each unit in each layer is either a *linear* or *non-linear perceptron* unit. Typically FCN's almost always use non-linear units. Each perceptron unit is made of two segments; *summation segment* and the *activation function*. The *summation* segment is the sum of product of the input values (x_i) and weights (w_i) that is associated with each input (see Figure 1). The input to the *activation function* is the output of the summation segment; This function is either a linear function (threshold function) or non linear functions such as Sigmoid (σ) or ReLu (Rectified Linear Unit).

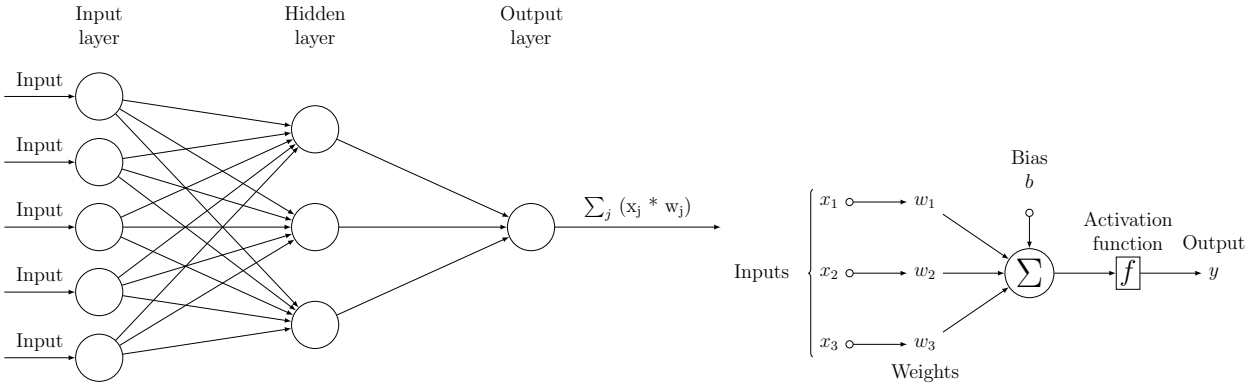


Figure 2.1: (left) Fully Connected Network (FCN) and (right) a single perceptron unit.

The key property of FCN is that each parameter (w_i) is independent from the others and are not shared among the disjoint units in the same layer. This network structure when used with a non-linear activation function, represents a differential function, which can be exploited to optimize the parameters of the network. We use *Back Propagation* (Lecun, Y, 1992) algorithm, an efficient algorithm that uses dynamic programming to perform gradient descent over the whole network to learn this differential function.

2.2 Convolution Neural Networks

Convolutional Neural Networks (CNNs) are similar to FCNs in that they have learnable weights and biases. Each neuron in CNNs is identical to FCNs: it computes the dot product of its inputs and the weight vector followed by non-linearity, and expresses a differential function that maps from raw input to output scores. CNNs also use the same loss function on the last fully connected layer as well as all the tips and tricks to learn the parameters as FCNs. They differ in that they make the implicit assumption that the input is a 3D volume (width, height, depth) and not a 2D volume. For example, a FCN has input vector composed of 3072 values for a 32 x 32 x 3 (width, height and depth) image. Each of the neurons in the second layer is connected to every neuron in the layer below it plus a bias term. But in

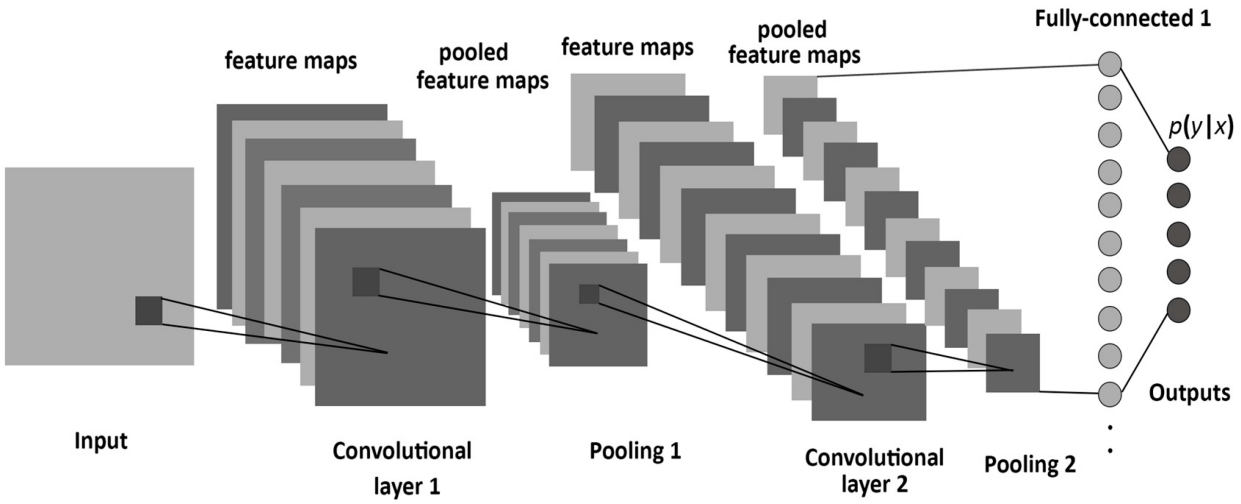


Figure 2.2: Typical Convolutional network

a CNN, each neuron is localized and is connected to a small region of neurons in the layer before it (Figure 2.2).

2.3 Layers used to build CNN

There are three main types of layers used to build CNN: *Convolutional Layer*, *Pooling Layer* and *Fully Connected Layer*. A CNN in its simplest form is a list of layers that transforms the input volume to an output volume. Each layer accepts a 3D input volume and outputs a 3D volume through a differentiable function. Sometimes, the layers are associated with hyper-parameters that help tune the layers to a particular task of interest. The *Fully Connected Layer* is exactly the same as the FCN's hidden/output layer.

2.3.1 Convolutional Layer

The Convolution layer is the core layer of CNN. The parameters of this layer is defined as a set of learnable feature maps called *filters or kernels*. Each filter is defined over a small spatial volume (along width and height) but is complete over the depth. For example, a typical filter at the first layer of CNN is of size 5 x 5 x 3 (5 pixels along the width, 5 pixels

along the height and 3 pixels along the depth since images have 3 channels). This filter is slid along the width and height of the input volume during the forward pass. At every stride of the filter the dot product of the input and the weights of the filter is computed. This dot product produces a 2D feature map that records the responses of that filter at every spatial position in the input. This feature map represents some aspect of the input such as egdes of some orientation, negative of the input image and borders of the object. We have an entire set of such 2D feature maps stacked along the depth dimension to form an output volume. A visualization of the Convolution Layer is show in Figure 2.3.

Often, it is impractical to connect a neuron to all the neurons in the previous layer for large images. Instead we connect each neuron to a local region of the previous layer called the *receptive field*. The size of the output volume depends on the receptive field and is controlled by three hyper-parameters:

1. The **Depth** hyper-parameter corresponds to the number of filters in the convolution layer. Each filter learns a different feature of the input.

2. The **Stride** hyper-parameter determines the step size with which we slide the filter along the width and height of the input volume. For example, if the stride is set to 2, the filter traverses two pixels at a time.

3. The **Zero Padding** hyper-parameter allows us to control the spatial size of the output volume. Sometimes it is necessary to maintain the same dimensions for the input and output volume.

   The spatial size of the output volume can be calculated as a function of the input volume, the stride and the padding hyper-parameter. See equation ((2.1)). The hyper-parameter settings are considered to be valid if the output volume is an integral value. A valid setting means that the neurons fit neatly and symmetrically across the input and output volume.

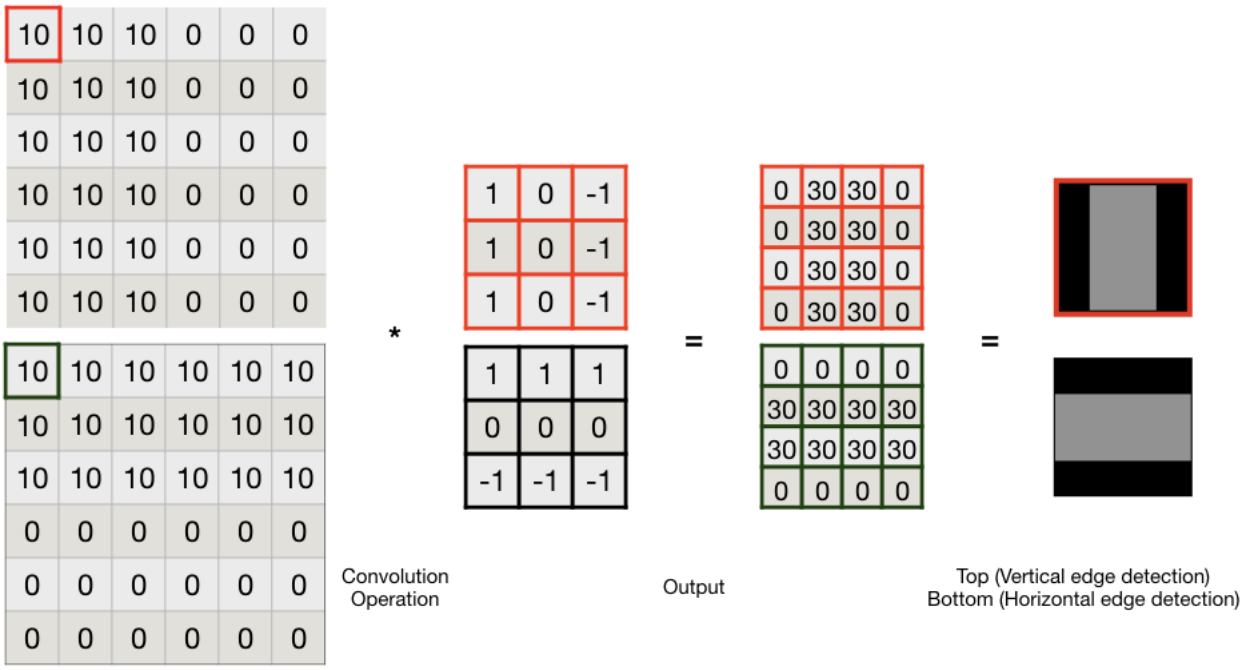


Figure 2.3: Convolution Layer with P = 0, S = 1 and 3 x 3 feature mapping

$$O = \frac{W - F + 2P}{S} + 1 \tag{2.1}$$

where:

W = size of the input volume;

F = volume of the receptive field;

P = amount of zero padding;

S = stride size;

In figure 2.3 we show an example of how convolutions can be used to detect edges. We have input volume W = 6 x 6 and two filters with receptive field of size F = 3 x 3. We use a stride of 1 (S = 1) and zero padding (P = 0). From the above equation we have a output volume of size

$$O = \frac{6 - 3 + 0}{1} + 1 = 4. \tag{2.2}$$

The output volume is a 4 x 4 two-dimensional matrix. As we slide the filters on the input with a stride of one in the horizontal and vertical direction we take the sum of the product of

the overlapping cells (dot product) and place the result into the output cell that corresponds to the offset of the filter in the input along the x and y directions. We can see from the outputs, the two filters detect vertical and horizontal edges in the input images. Hence we use convolution layers to detect features (including non-linear features) in the input image and map these features to the output labels. For detecting non-linear features we apply a non-linear function such as $\sigma(sigmoid)$, ReLu (Rectified linear unit) to the dot product. See figure 5.2 for examples of non-linear features.

2.3.2 Parameter Sharing

Convolution Layer uses the concept of parameter sharing (tying) to control the number of learnable parameters. Each feature map uses the same parameters across the input volume. For example if the input has a dimension of 227×227×3 neurons and if we choose the parameters F = 11×11×3, P = 0 and S = 4, then from Eq. (2.1) the output volume is 55×55×96 (where 96 is the number of feature maps) which maps to 290400 neurons. Each of the neurons have 11×11×3 = 363 weights and bias summing to a total of 55×55×96 ×11×11×3 = 105,705,600 parameters which is quite high and impractical. It is reasonable to reduce the number of parameters under the assumption that if one feature is useful to be computed at one location (x, y) it should also be useful at a different position. Hence in CNN the parameters for a feature map are shared across the input volume, such that when the filter makes a jump using the stride size to location l_1 from l_0 the same parameters are reused. Thus, for the above example the parameter sharing assumption produces 11×11×3×96 = 34,848 unique weights and 96 biases summing to a total of 34,944. All the 55×55 neurons at each output slice (also called *depth slice*) will share the same parameters. During back propagation, every neuron will add the gradient at each depth slice and only a single set of weights are updated per slice. As we saw in figure 2.3, for a single filter the same values (weights of the filter) are being used (shared) across the spatial dimension of the input.

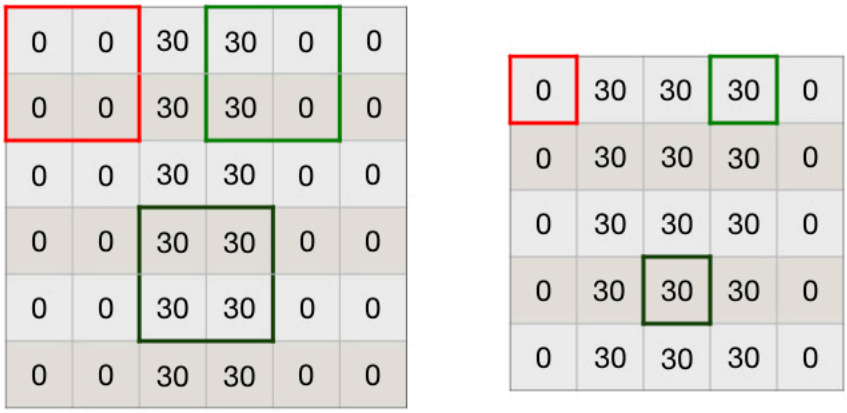


Figure 2.4: Max Pooling with pooling dimension 3 x 3 and stride S = 1.

2.3.3 Pooling Layer

It is normal to use pooling layers in between the successive convolutional layers in CNN. Pooling layers are use to reduce the spatial size of the visual representation as the network progresses thus reducing the number of parameters and hence avoiding overfitting. The two commonly used variations of pooling are *Max Pooling* and *Average Pooling*. Max Pooling has shown to work better in practice. A visualization of max pooling is shown in Figure 2.4. Here the dimension of the pooling layer is 3×3 and has a stride of 1. When we place the pooling layer on top of the convolution, we take the maximum of the elements in the pooling window and place the resulting value in the output cell that has an offset equivalent to the offset of the pooling layer in the convolution layer. We highlight three such mappings in the figure. We also see from the figure that the original input is down sampled from 6×6 to 5×5. While updating the parameters during back propagation, the index of the max activations is cached. This helps in propagating the gradient in the correct direction. It is found that discarding pooling layers has improved models, such as Generative Adversarial Networks (GAN's). In such situations the parameters can be reduced by down sampling the visual representation using longer strides in the convolution layers.

2.4 GoogLe Net

GoogLe Net (Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, 2015) strayed away from the traditional CNN architecture from stacking convolution layers and pooling layers over top of each other followed by a FCN in a sequential manner. It is a 22 layer deep architecture and places notable consideration on the computation complexity and memory usage. The key components of the architecture is the *Inception Module* which is a new level of organization of the convolution layers. It makes an extensive use of 1×1 convolutions proposed by (M. Lin, Q. Chen, and S. Yan., 2013) to increase the representational power of the network without increasing the size of the set of learnable parameters; increasing the number of parameters to learn more expressive models can lead to overfitting if the dataset is scarce.

2.4.1 1×1 Convolutions

In GoogLe net 1×1 convolutions have dual purpose: to increase the network depth and width without compromising performance. They are used mainly as dimension reduction modules to avoid increasing the size of parameters set, that would otherwise limit the size of the network. Let us see with an example the power of 1×1 convolution. Say the input volume of convolution layer is 28×28×192 and we want the output volume to be 28×28×32 (reduced depth). If we use a 5×5×32 filter, each unit in the output volume is a dot product of 5×5×192 units from the input volume. This totals to over 120M parameters. If we use 1×1×16 filter followed by a 5×5×16 filter, the total number of parameters is just over 12M. Thus, there is a ∼90% reduction in the number of parameters that has to be learned. This structure allows us to build deeper and wider networks by removing computational bottle-necks.

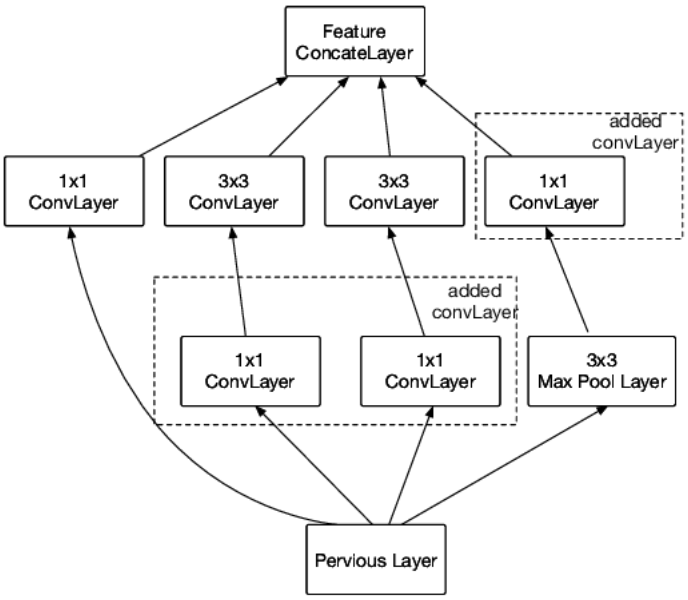


Figure 2.5: Inception Module with 1 x 1 convolutions

2.4.2 Inception Module

In the lower convolutional layers, the ones closer to the input, the correlation between the input pixels is concentrated in the local regions. This results in a high degree of clusters concentrated in local regions. This also suggests that the number of more spatially spread out clusters decreases over larger patches. 1×1 convolutions is used for capturing the semantic relations of the pixels in the cluster and 3×3, 5×5 convolutions are used to cover the semantic relations between spatially spread out clusters and patches of clusters (see Figure 2.5). This is a convenient way of increasing the expressive power of the model, since it strays away the problem of deciding the type of filter to be used at each convolution layer (patch alignment issues). The outputs of all the filters are concatenated into a single volume of output along the depth dimension that forms the input to the next layer. Since pooling layers have shown to be beneficial in CNNs, an additional Max Pooling is added in parallel to the module (see Figures 2.5 and 2.6).

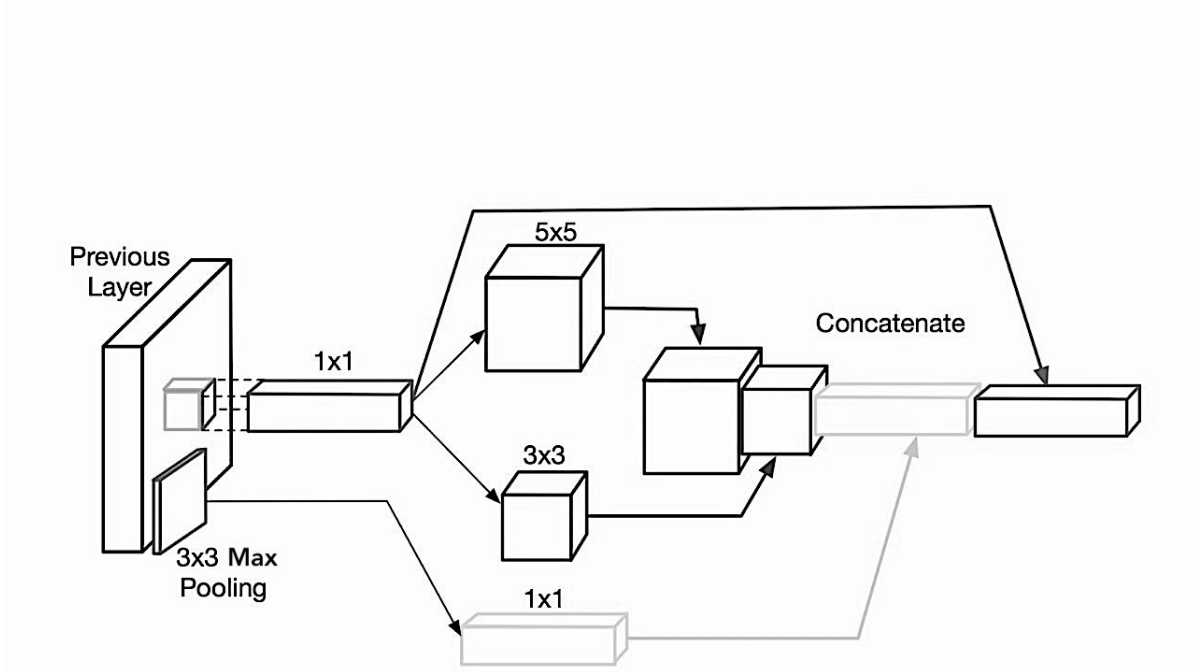


Figure 2.6: Concatenation in Inception Module along the depth.

2.5 Convolutional networks and visual recognition patterns

One of the important reasons for using Feed Forward Networks (like FCN and CNN) is to eliminate manual feature extraction. We can rely on back-propagation to convert the few initial layers to feature extractors. FCN can then be used as classifier to categorize these features into classes. While FCNs can be used as feature extractors for raw inputs like images, there are some problems.

First, inputs like images typically have a large number of variables (orders of hundreds). If the data source is scarce, then it could lead to overfitting. Moreover, it also significantly increases the time and space computational costs. But the main problem of unstructured networks for images is that there is no in built invariance with respect to translations. Sending an image to fixed size input requires it to be size normalized and centered to the input. This pre-processing is difficult since the objects in the image may occur at different positions, may have different degree of rotations and translations. Learning this reliably

using an FCN requires a large amount of labeled training data and since labeling is typically quite expensive, FCNs are impractical.

Second, in FCNs the input can be provided in any fixed order without affecting the output. But in images, on the contrary, pixels that are spatially or temporarily closer are highly correlated. This local correlation is the key reason for the extraction and combining of local features before recognizing objects.

CNNs overcome these problems by adapting variable sized filters and the concept of weight sharing. This ensures translational and distortion invariance to some extent. The idea of connecting the neurons to local receptive fields is to model the local correlations. This also forces the set of neurons whose receptive fields are at different location of the input to share the same weight (see 2.3.2). The initial layers extras features such as edges, corners, image negative which are then combined in the higher layers. Therefore, a convolution layer uses multiple feature maps to capture multiple features. The strides of the receptive fields for the feature maps ensures that the image is center normalized. The different sizes of the receptive field allows CNNs to capture the correlations in the local cluster and spatially spread out clusters. Hence once the feature has been extracted, its exact position becomes less important as long as its relative position to other features is preserved. If there is a shift in the input, the output shifts. Therefore typically the convolution layers are followed by pooling to sub-sample the input features and reduce the effect of distortion and translation. Consequently, these architectural components of CNN make it invariant to translations and distortions and ideal for visual recognition tasks.

CHAPTER 3

ACTIVITY DETECTION IN VIDEOS

Detecting activities in videos requires understanding the semantic relationship between the objects, actors and actions. Learning this relation in a naive way requires a large dataset consisting of all the variations of objects, actors, actions and the annotations for the Cartesian product of the features. This is impractical as the size of the resulting set of feature combinations is very large. Hence we use CNN architecture for feature extraction and for prediction of the activity in a scene.

3.1 Objective

The objective here is to take a video as a input and extract the activities in the video scenes. In other words, the video should be able to narrate the activities that take place at different time during playback .

3.2 Dataset

For this task we are using the *TACoS Multi-Level corpus - MPII Cooking 2* dataset by Anna, Rohrbach, Marcus, Qiu, Wei, Friedrich, Annemarie, Pinkal, Manfred, Schiele, Bernt. (2014).

3.2.1 Annotations

Each video in the dataset is annotated with respect to each frame. The annotations are *fileName* (e.g. s13-d21), *startFrame* (e.g. 315), *endFrame* (e.g. 1291), *descriptionIdx* (e.g. 4), *ignore* (e.g. 0), *sentenceProcessed* (e.g. the person took out a cutting board , knife , cucumber , and a plate), *activity*(e.g. enter), *tool* (e.g. knife), *object* (e.g. cutting-board), *source* (e.g. drawer), *target* (e.g. counter). Out of these numerous annotations, we use the following four annotations that form the elements of an activity:

- **Activity** annotation can be interpreted as an action. It represents the action performed by the subject on a object in the scene.
- **Tool** annotation represents the element of the activity that is acted on something.
- **Object** annotation represents the main object in the activity. This is the object that is being acting upon.
- **Source/Target** annotations represent the secondary object in the activity.

Since these four elements represent an activity, our learning problem is a multi-label classification problem. Note that an activity can be composed of a subset of the four elements. For example enter or dry - towel - hand.

3.2.2 Data Processing

The videos in the dataset have different cooking scenes. By different we mean, different object, tool, action, source and target. Since the dataset is scarce in the different activity elements, we decompose the video into frames. Each frame is $\frac{1}{24}^{\text{th}}$ of a second. With this pre-processing we can generate ample images for the different elements. For example images see Figures 3.1 & 3.2.

Further, we use a single patch of the frame. Each frame is right centered after scaling the shortest dimension (height) to 224 pixels. We then adjust the width of the frame to 224 pixels from the right hand side by cropping the extra pixels on the left part of the image. We use the resulting image of size 224 x 224 pixels as input to the network. Each preprocessed image is stored in the lmdb database as a key-value pair with frame number as the key.

3.3 Architectural Details

We will be using the *BAIR/BVLC GoogleNet Model* (Christian Szegedy, Sergio Guadarrama., 2014) that is pre-trained on the ILSVRC dataset. It is a 22 layer deep network. The network

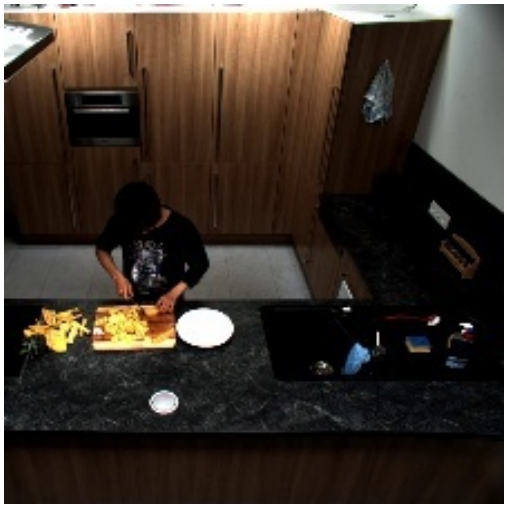


Figure 3.1: hand + chopping + orange

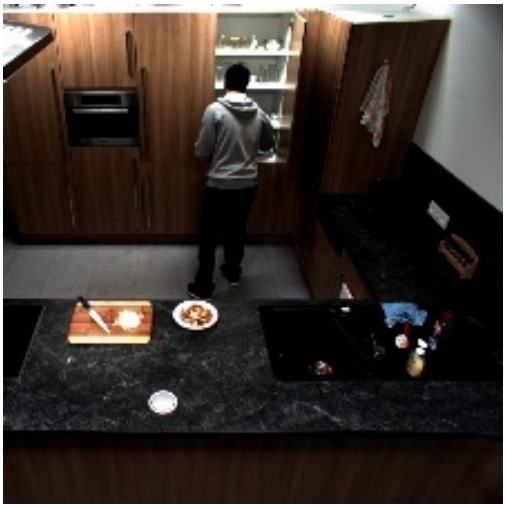


Figure 3.2: hand + pull + plate + cupboard.

incorporates the translational invariance property using convolutional layers. As discussed in section 2.4 the architecture is a combination of all the layers that captures the correlation between local clusters that are spread spatially and the correlation between the local pixels in a single concatenated output volume. The *Inception modules* are stacked on top of each other and hence their output correlation statistics are bound to vary: as features of higher abstraction are captured by higher layers, their spatial concentration is expected to decrease. The detailed architecture is shown in Table 3.1.

The design of the GoogLe net ensures that there is no computational blow up as the number of layers increases by using the 1×1 convolutions (reduction layers). Moreover these 1×1 convolutions use the rectified linear units to knockout vanishing gradient problems and avoid the introduction of sparse activations in the hidden layers of the network. The design of the network helps us to abstract features at various scales using different patch sizes for the filters (1×1, 3×3 and 5×5) and then aggregating these features so that the next layer can abstract still higher representation from different scales simultaneously.

We propose a small change to this architecture by appending a FCN to the top layer. Specifically the **softmax layer** is removed and we add a new **fully connected layer** con-

sisting of **28** units with **Sigmoid Cross Entropy loss** function. The additional layers are shown in bold in Table 3.1. The modified network is shown in Figure 3.3.

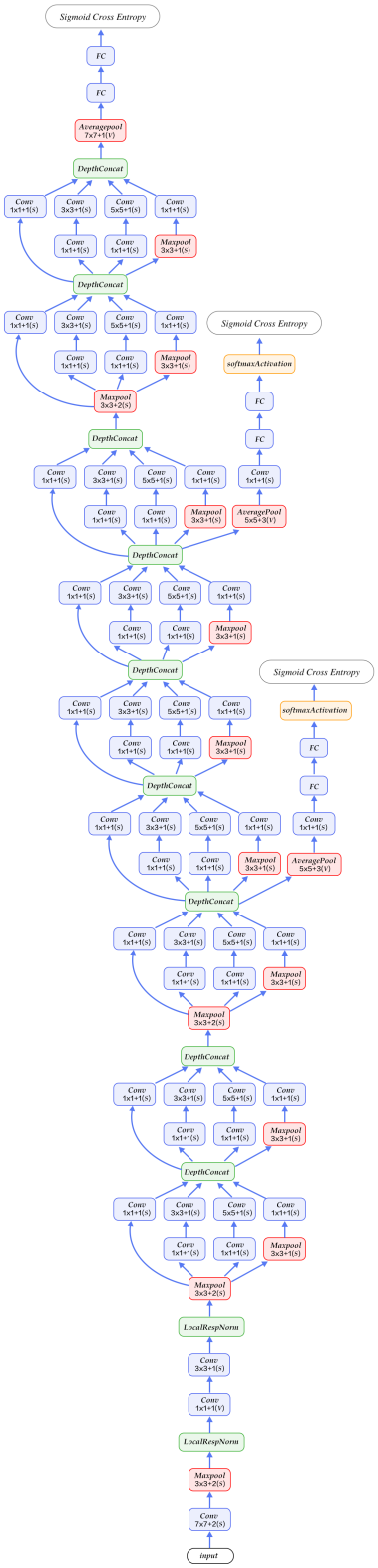


Figure 3.3: GoogLeNet.

Table 3.1: GoogLeNet incarnation of Inception architecture. Source:(Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, 2015).

type	**patch size/ stride**	**output size**	**depth**	**#1 x 1**	**#3 x 3 reduce**	**#3 x 3**	**#5 x 5 reduce**	**#5 x 5**	**pool proj**	**params**	**ops**
convolution	7 x 7/2	112 x 112 x 64	1							2.7K	34M
max pool	3 x 3/2	56 x 56 x 64	0								
convolution	3 x 3/1	56 x 56 x 192	2		64	192				112K	360M
max pool	3 x 3/2	28 x 28 x 192	0								
inception (3a)		28 x 28 x 256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28 x 28 x 480	2	128	128	192	32	96	64	380K	304M
max pool	3 x 3/2	14 x 14 x 480	0								
inception (4a)		14 x 14 x 512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14 x 14 x 512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14 x 14 x 512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14 x 14 x 528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14 x 14 x 832	2	256	160	320	32	128	128	840K	170M
max pool	3 x 3/2	7 x 7 x 832	0								
inception (5a)		7 x 7 x 832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7 x 7 x 1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7 x 7/1	1 x 1 x 1024	0								
dropout (40%)		1 x 1 x 1024	0								
linear		1 x 1 x 1000	1							1000K	1M
softmax		1 x 1 x 1000	0								
fc		**1 x 1 x 28**	**1**							**28K**	**56K**
sigmoid cross entropy		**1 x 1 x 1000**	**0**								

CHAPTER 4

EXPERIMENTAL RESULTS

The aim of the experimental evaluation is two fold: comparing the speed of training in terms of CPU time, and accuracy measured in terms of test-set k-group scores and the conventional accuracy measure.

4.1 Training Methodlogy

For the slightly modified network, the weights until the first fully connected layer were loaded from the pre-trained network. The weights for the last fully connected network were initialized using "xavier" distribution instead of "gaussian". As specified in the dataset section 3.2 the images are stored in a lmdb database. This database stores the key-value pairs in lexicographical order. Since each image is a frame that is $\frac{1}{24}^{\text{th}}$ of a second, a series of sequential images contains very small degree of variation in the pixel values. Training the network on images in such an order is a bad idea because the model might converge to local optima of the function being learned very quickly leading to a low accuracy. Therefore, the keys associated with the images were mapped to a 128 character (alpha-numeric) hash. The hash function produces unique key such that no two images have the same key. Further these keys were sampled uniformly and stored back in the database. This sampling method yields a uniform distribution over the images. The images are then retrieved in lexicographical order for training the network.

The output of the last fully-connected layer is a set of disjoint units, each representing a unique activity component available in the dataset. For evaluating the model we used a toy example that consisted of the activity components *carrot*, *cut off ends*, *cut dice*, *cutting-board*, *chefs-knife-cutting-board*, *paper-bag-pot*, *chefs-knife*, *wash*, *plate*, *drawer-counter*, *shake*, *cut apart*, *fridge-counter*, *drawer-cutting-board*, *throw in garbage*, *scratch off*, *hand*, *knife*, *cutting-*

board-plate, *cutting-board-counter*, *cupboard-counter*, *peel*, *enter*, *move*, *slice*, *take out*, *Nothing*. Hence the last fully connected network has 28 outputs. The *Nothing* element represents the state zero activity.

As discussed in the annotations section 3.2.1, this is a multi-label classification problem and therefore we need a cost function that works very well for many-of-many labels (as opposed to one-of-many labels). The Sigmoid Cross Entropy function is a perfect candidate for this problem. It computes the cross-entropy (logistic) loss:

$$E = -\frac{1}{n}\sum_{i=1}^{n} y^{(i)} \log(o(x^{(i)})) + (1 - y^{(i)}) \log(1 - o(x^{(i)})) \tag{4.1}$$

where $\{x^{(1)}, \ldots, x^{(n)}\}$ is the set of input examples, $\{y^{(1)}, \ldots, y^{(n)}\}$ is the corresponding set of labels for those input examples in the training dataset and $o(x) \in (0, 1)$ represents the output of the neural network given input x.

This function can be interpreted as a measure of surprise. When $o(x) \in (0, 1)$ and $y << 1$, it corresponds to the situation where the model is not confident about the class, and yet in reality it is the actual class. As a result, the "surprise" of your model is high: the model did not account for that event. Each such surprise is summed over all the output labels. This describes how different the model is from the true model. It is then used by the back-propagation algorithm to update the weight parameters. Note that during the test time we replace the Sigmoid Cross Entropy function with *Sigmod function*:

$$y = \frac{1}{1 + e^{-\mathrm{x}}} \tag{4.2}$$

This function is independently applied to each output unit.

4.1.1 Gradient Descent Optimizers

The weights are learned using back propagation (Lecun, Y, 1992). Two variants of the gradient descent algorithm that are used to find the error gradient are:

- Stochastic Gradient Descent with learning rates 1e^{-5} and 1e^{-6}. Stochastic gradient descent (type: "SGD") updates the weights W by a linear combination of the negative gradient $\bigtriangledown L(W)$ and the previous weight update V_t. The learning rate α is the weight of the negative gradient. The momentum μ is the weight of the previous update.

$$V_{t+1} = \mu V_t - \alpha(\bigtriangledown L(W_t)) \tag{4.3}$$

$$W_{t+1} = W_t + V_{t+1} \tag{4.4}$$

- Adaptive Moment Estimation with base learning rates 1e^{-5} (Sebastian Ruder., 2017). The ADAM gradient optimization algorithm, proposed in (Diederik P. Kingma, Jimmy Ba., 2014), is a gradient-based optimization method (like SGD). This includes an *adaptive moment estimation* (m_t,v_t) and can be regarded as a generalization of AdaGrad. The update formulas are

$$(m_t)_i = \beta_1(m_{t-1})_i + (1-\beta_1)(\bigtriangledown L(W_t))_i \tag{4.5}$$

$$(v_t)_i = \beta_1(v_{t-1})_i + (1-\beta_1)(\bigtriangledown L(W_t))_i^2 \tag{4.6}$$

$$(W_{t+1})_i = (W_t)_i - \alpha * (\frac{\sqrt{1-(\beta)_2)_i^t}}{1-(\beta)_1)_i^t} * \frac{(m_t)_i}{\sqrt{(v_t)_i} + \epsilon}) \tag{4.7}$$

(Diederik P. Kingma, Jimmy Ba., 2014) proposed to use β_1=0.9, β_2=0.999, ϵ=1e^{-8} as default values. We uses the values of *momemtum*, *momentum 2*, *delta* for β_1, β_2, ϵ. respectively. See table 4.1.

As seen in the Data Processing section 3.2 each image is a 224×224×3 three dimensional matrix we use a batch size of 16 images per iterations, and evaluate the evaluation at the end of 3016 iterations (*epoc*). For all the three gradient descent algorithms we use **Early Stopping** heuristics to avoid overfitting the network.

Table 4.1: Comparison of gradient descent optimization algorithms.

Gradient Descent Algorithms	**Base Learning Rate**	**Step Size (Iterations)**	**Maximum Iterations**	**Weight Decay**	**Momentum**	**Momentum 2**	**Gamma**	**Convergence (Iterations)**	**Epocs**	**Delta**
SGD - 1	$1e^{-5}$	320000	10000000	0.0002	0.9		0.96	233200	77	
SGD - 1	$1e^{-6}$	320000	10000000	0.0002	0.9		0.96	835240	277	
ADAM	$1e^{-6}$		10000000		0.9	0.999	0.96	186000	51	$1e^{-7}$

4.2 Training Analysis

Since we are using the trained network, the initial drop in the error rate is very high (drops below 10 within the first 900 iterations). In figure 4.1 we can see the error rates for the different gradient descent optimizers for the first 8000 iteratios. We see that ADAM optimizer learns quickly compared to SGD with learning rates e^{-5} and e^{-6}. This is reflected from the number of iterations or epochs taken by the optimizers as shown in table 4.1. Table 4.2 compares the three optimizers as a plot of Iterations v/s error. This graph provides us valuable insights about the different optimizers. We can see that ADAM converges to a better local minimum than SGD. Also the former converges very quickly compared to the latter and this can be attributed to the fact that ADAM not only considers the exponentially decaying average of past squared gradients but also exponentially decaying average of the past gradients. In other words, it performs larger updates for infrequent and smaller updates for frequent parameters.

We can also see in Figure 4.3 the state of the model when trained on frames in their order of occurrence in video playback. Initially the error rate starts decreasing but starts increasing with high variance as the number of iterations increase. There seems to be no sign of convergence. The error keeps oscillating. Figures 4.1 and 4.3 strongly suggests that the network needs to be trained using uniform random sampling of frames.

4.3 Accuracy and Training time

We evaluate the accuracy using the K - Group measures and the conventional accuracy measure. Since we are learning the activity in a scene, we assume that each activity is made up of at most four elements *action*, *tool*, *object* and *source-target*. Because each activity is a subset of the four elements, we evaluate the accuracies using K-1, K-2, K-3 and K-4 group heuristics:

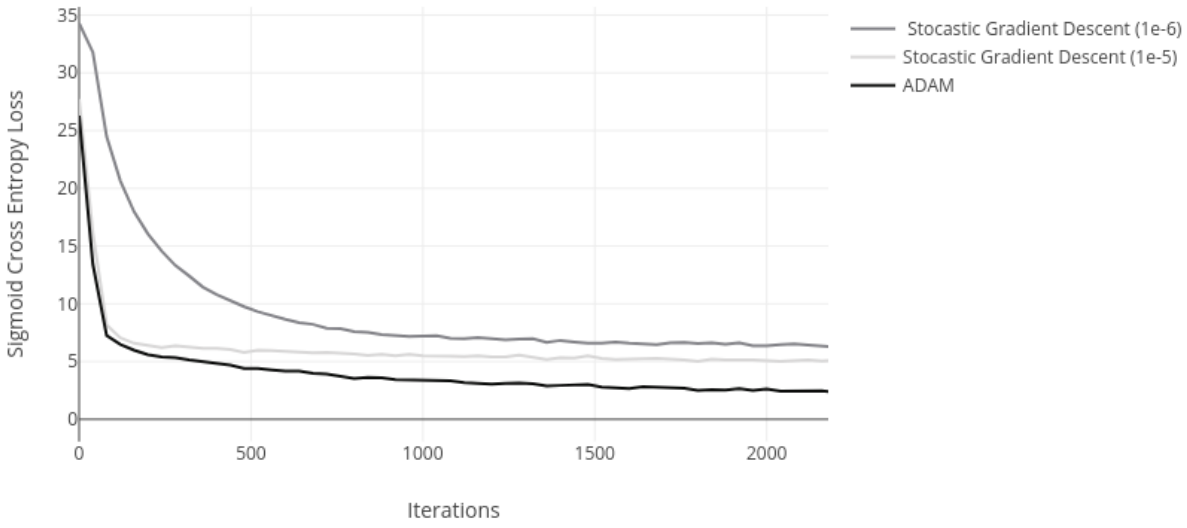


Figure 4.1: Learning rate of different gradient descent optimizers.

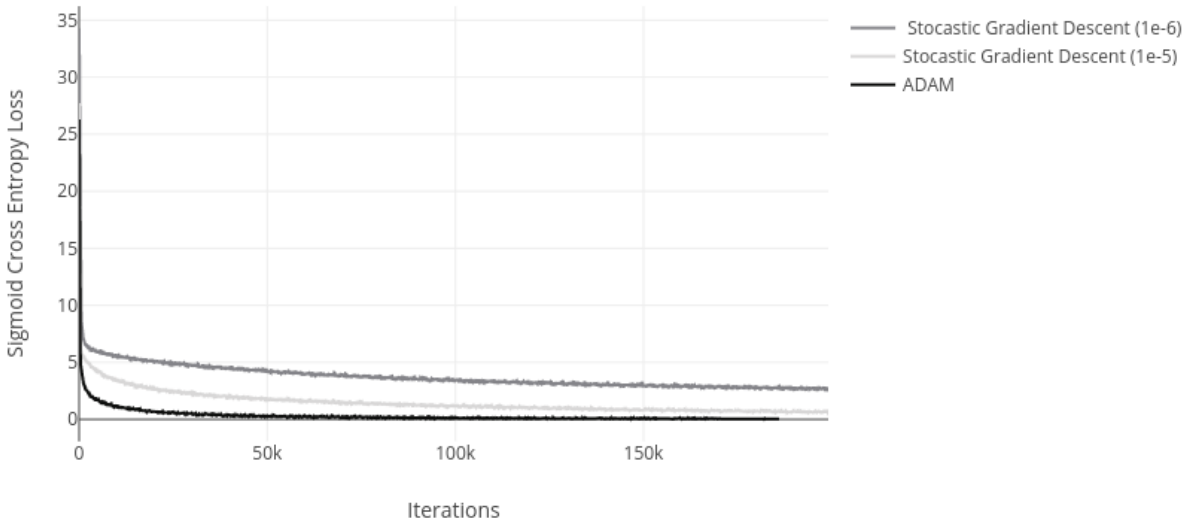


Figure 4.2: Error (Iterations v/s sigmoid cross entropy loss).

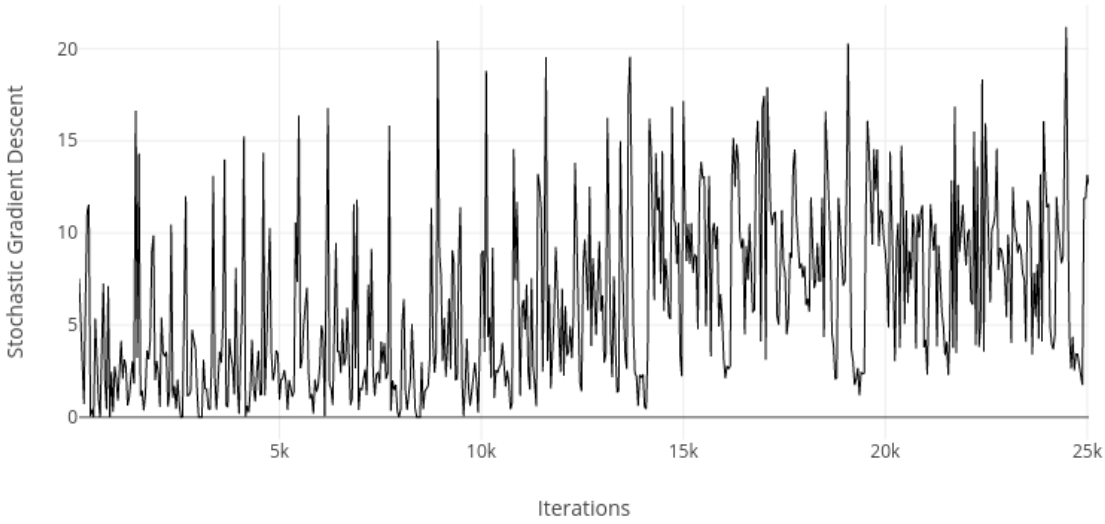


Figure 4.3: Error rate when the network is trained in lexicographical order of the frames. In other words, when the videos frames are trained in sequential order.

- **K-1** This heuristic considers each element of the activity as independent of each other. We just count the number of positive predictions and negative predictions for each element individually.

- **K-2** This represents the number of times the model predicted two ground truths correctly for the same instance.

- **K-3** This represents the number of times the model predicted three ground truths correctly for the same instance.

- **K-4** This represents the number of times the model predicted four ground truths correctly for the same instance .

We train and evaluate the model on 18 distinct videos. We train the model on 78995 unique frames that are 1/24th of a second of the video playback. These frames are further

divided into training (70% split, 55,297 frames) and evaluation dataset (30 % split, 23,698 frames) using uniform random sampling. In this data set there are 28 unique activity elements as shown in Table 4.4. The test dataset consists of frames from 6 different videos containing the same activity elements as the training dataset. The model was tested on 16,247 unique frames in the lexcographical order of the frame number in the video playback. In other words, the video frames were fed into the network in sequential order.

To measure the accuracy of the network we need to determine the correct activity for each frame. Because each frame can be composed of at most four elements of the activity we cannot assume that every frame is composed of all the elements. Determining which elements of the activity is represented by the frame, we first need to know how confident the model is for each activity element in the trained model. We simply cannot apply *argmax* function over the output layer to get top four activity elements. This is incorrect because:

- Not every frame is composed of four activity elements.
- The dataset might contain noise in that the same object may be labeled with different ground-truth labels. Moreover, there may be too few instances containing a specific object.

To remedy this problem, we use thresholds for each activity element. The threshold provides a bound above which the model is confident in its prediction of the corresponding activity element. The value of these bounds lies in the range [0, 1]. For our toy dataset, the bounds for the activity elements and their corresponding prediction accuracy (Independent of other elements) is show in Table 4.4.

We trained our model on a Amazon EC2 GPU instance (g2.2xlarge: NVIDIA GRID GPU (Kepler GK104) and 8 x hardware hyperthreads from an Intel Xeon E5-2670). For modeling CNNs, we used the Caffe framework. The training data had 60313 image instance and the evaluation set had 9327 image instances. During test time we used videos that our

Table 4.2: Accuracy of the model using different gradient descent optimizers from table 4.1. The accuracies are in the scale [0, 1] (0 being the lowest and 1 being the highest).

Gradient Descent Optimizer	**K-1**	**K-2**	**K-3**	**K-4**
SGD -1	0.79	0.73	0.70	0.65
SGD - 2	0.83	0.80	0.76	0.71
ADAM	**0.96**	**0.91**	**0.89**	**0.87**

Table 4.3: Time taken by the model using different gradient descent optimizers.

Gradient Descent Optimizer	**Time**
SGD -1(Base learning Rate: 1e-6)	4 days
SGD - 2(Base learning Rate: 1e-5)	4 days
ADAM (Base learning Rate: 1e-5)	**2 days**

model has not seen before and these videos had similar actions as the videos in the training set. The time taken by the model is show in Table 4.3.

Table 4.4: Activity element and their corresponding threshold & accuracy

Activity Element	**Optimal Threshold**	**Accuracy**
fridge	0.99	1
carrot	0.99	1
cut off ends	0.99	0.98
cut dice	0.99	1
cutting-board	1	0.65
chefs-knife-cutting-board	0.99	0.99
paper-bag-pot	0.99	1
chefs-knife	1	0.77
wash	0.99	0.96
plate	0.8	0.97
drawer-counter	0.8	0.97
shake	0.99	0.99
cut apart	0.99	0.99
fridge-counter	0.99	0.99
drawer-cutting-board	0.99	0.97
throw in garbage	0.99	0.99
scratch off	0.99	0.99
hand	1	0.53
knife	0.85	0.71
cutting-board-plate	1	0.92
cutting-board-counter	0.99	1
cupboard-counter	0.99	0.97
peel	1	0.91
enter	0.8	0.98
move	1	0.91
slice	0.99	0.85
take out	0.935	0.88
Nothing	0.99	0.80

CHAPTER 5

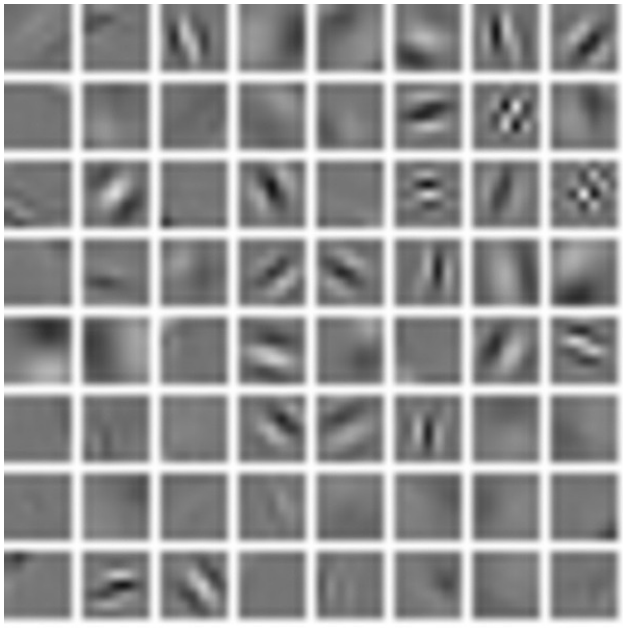
VISUALIZING CONVOLUTIONAL NEURAL NETWORK

One of the challenges of using neural networks is understanding what exactly goes on at each layer. We know that after training, each layer progressively extracts higher-level features of the image, until the final layer essentially makes a decision on what the image shows. For example, the first layer possibly looks for edges or corners. Intermediate layers interpret the basic features and try to model overall shapes or components such as a door or a leaf. The final few layers assemble these into complete interpretations - these neurons activate in response to very complex things such as entire buildings or trees. Several techniques have been developed to understand and visualize convolutional networks due to a lot of criticism that the learned features of these networks are not interpretable. We present two techniques here: one which show the activations for the input image at different level of the network and the other is a way to turn the network upside down and ask it to enhance an input image in such a way as to elicit a particular interpretation.

5.1 Visualizing activations from the layer weights

We will first visualize the network by retrieving the weights from several layers captured as a image. We will input an image to the network and forward propagate it. The representation is blobby and dense in the initial layers but as we proceed to the higher layers the activations are sparse and start localizing. Usually the filters in a well trained network are smooth. Figure 5.1a and 5.1b show the input image and the first layer of the convolution containing 64 filters. Notice the first layer of convolution is nice and smooth indicating the network has converged. Since the GoogLe network uses a single stream of input, we can see that the convolution layer has learnt both high frequency grey scale and low frequency color scale features.

(a) Input image with annotations: *hand* + *cutting* + *cutting board* + *counter*.



(b) First convolution layer with 64 filters (P = 3; S = 2; F = 7)

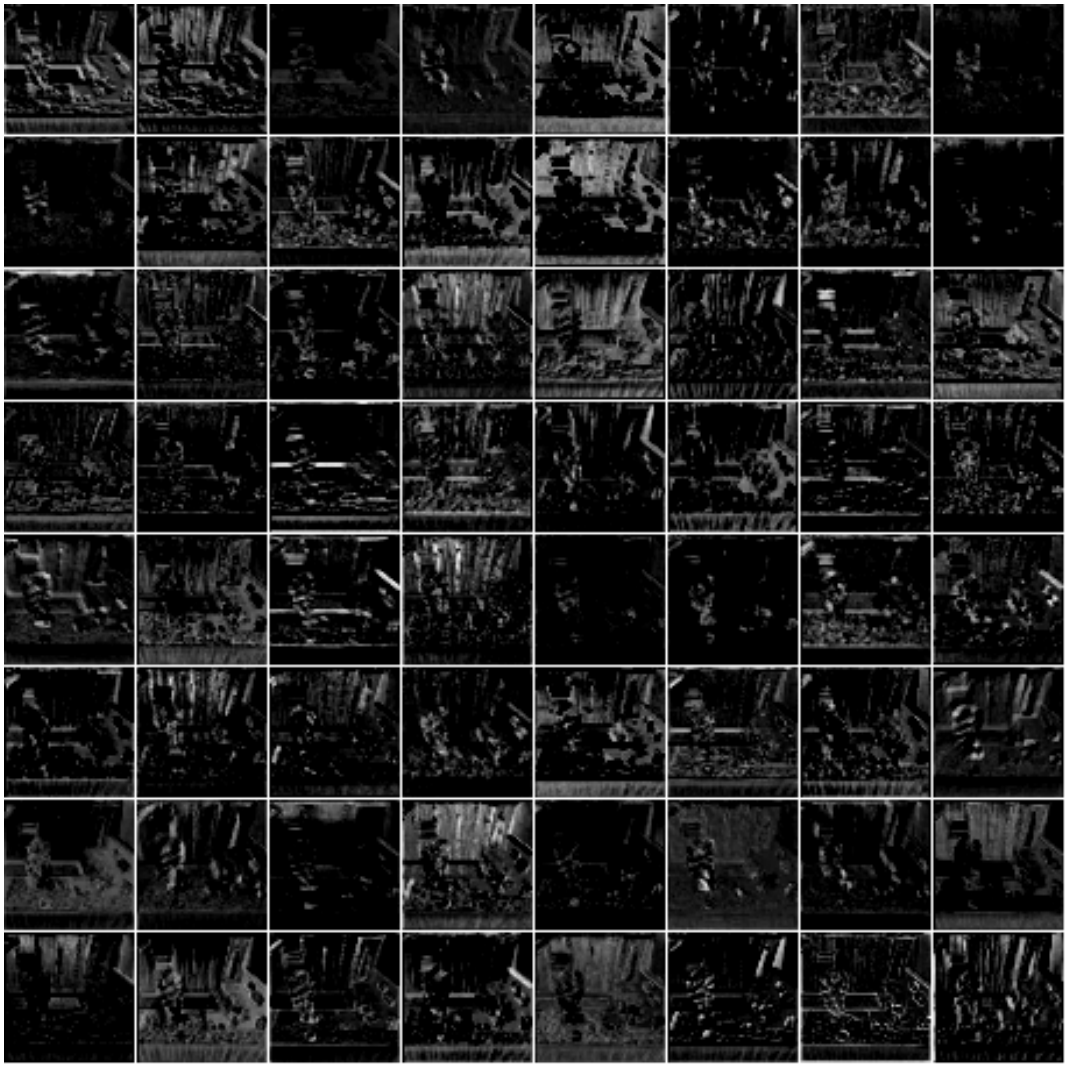
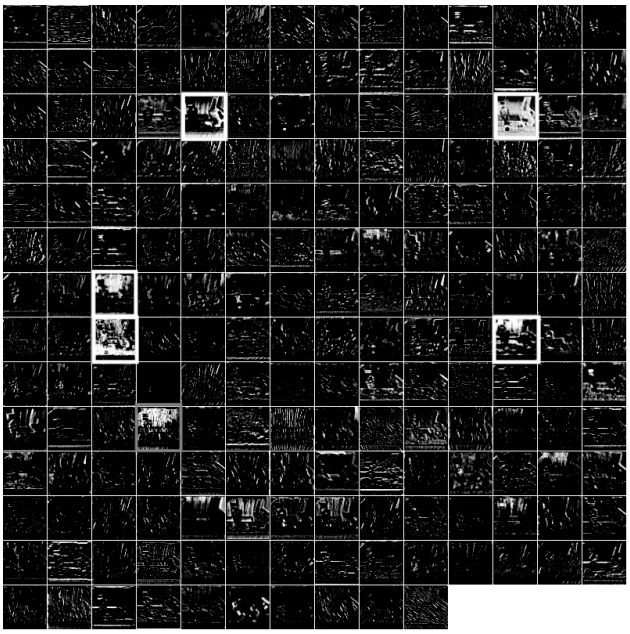


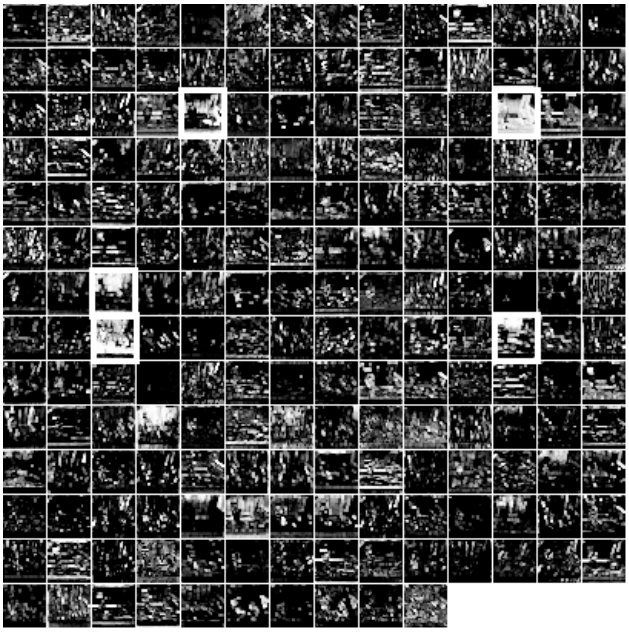
Figure 5.2: (left) Fully Connected Network (FCN) and (right) a single perceptron unit.

Figure 5.2 shows the representation of the input at the second convolution layer. We can see from the image that the layers closer to the input are very sensitive and can learn details at a granular level. The filter at row 8 and column 3 seems to have learned the borders. The filter at row 7 and column 8 seems to have learned the vertical edges. The color of this image is black and white as opposed to the first layer of the convolution.

Figure 5.3a and 5.3b shows the difference in the convolutional values after applying a *MAX* pooling function to the output of the second convolutional layer. The filters marked



(a) Second convolutional layer with 64 filters. Areas marked with white boxes are the filters with high activation.



(b) Max pooling on the second convolutional layer with F = 3 and S = 2. Areas marked with white boxes are the filters with high activation corresponding to the markers in figure 5.3a.

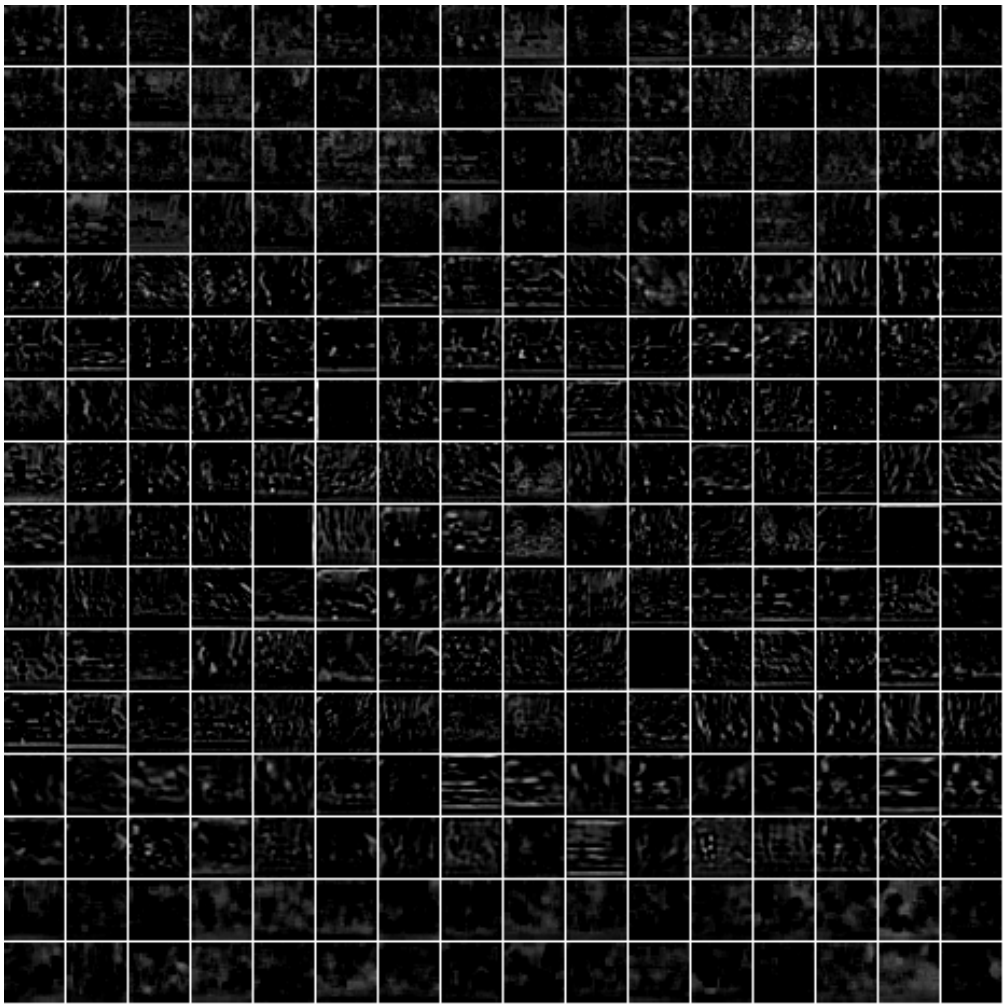
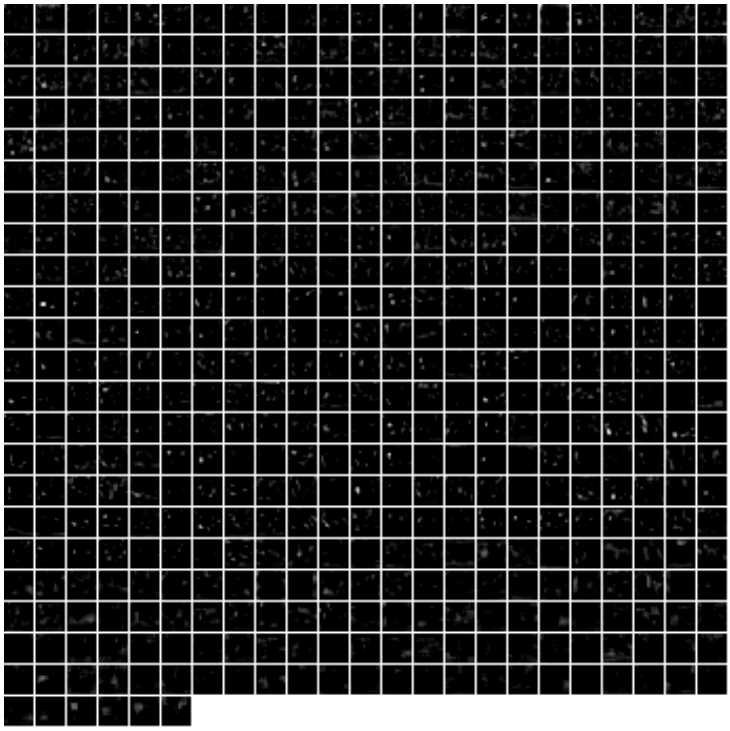


Figure 5.4: (left) Fully Connected Network (FCN) and (right) a single perceptron unit.

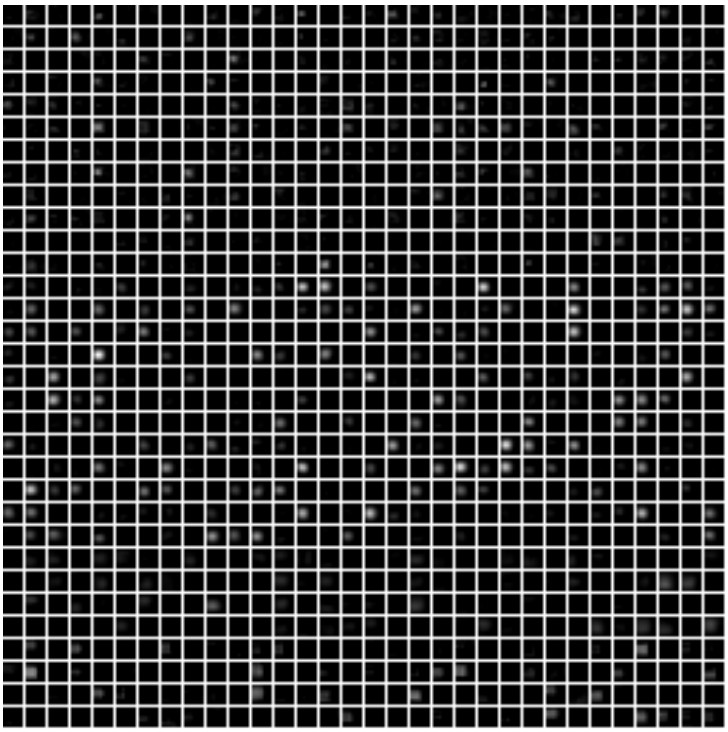
using a white box shows the filters with high activations. We can see the difference between the two images where the second image is a down sampled image by applying a pooling window of size 3 and stride 2. Because the image is being downsampled using a stride that overlaps with the previous window the same activated neuron in the previous window might be mapped to the current window and hence the high activations in the pooling layer.

Figure 5.4 shows the activations of the neurons at the output of the first inception module.

The activations in this layer are sparse and localized. Figure 5.5a and 5.5b shows the activations in the last two inception modules. From the images we can see that the repre-



(a) Convolutions at the 4^{th} Inception Module.



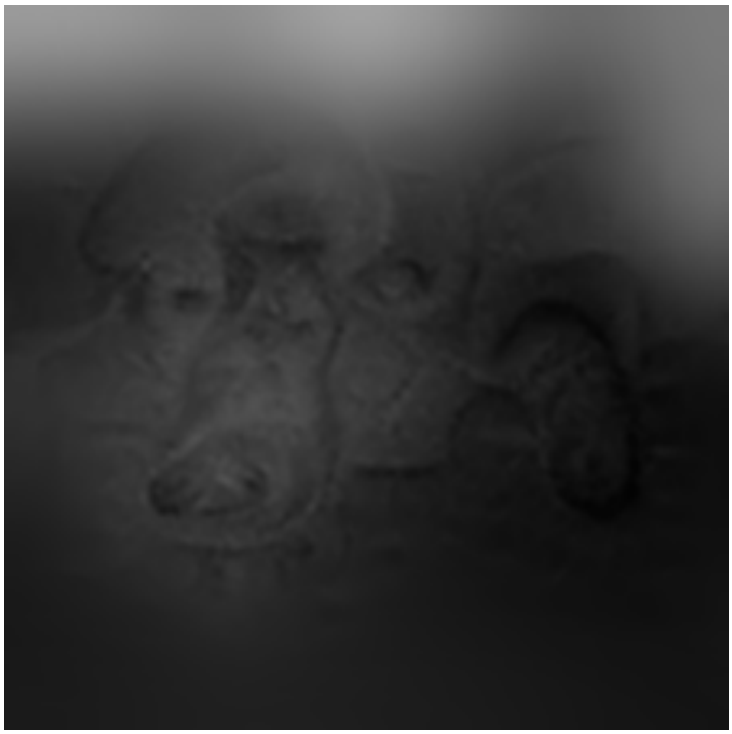
(b) Convolutions at the last Inception Module.

sentation at the last inception module (see Figure 5.5b) is highly localized compared to the inception module in Figure 5.5a.

With this type of visualization we can determine if the model is trained well enough. Noisy patterns can indicate that the model is not well trained or might have been over-fitted.

5.2 Visualizing activations from the layer weights by enhancing an input image

In this method, we provide a layer with an input image and let the network analyze the picture (Alexander Mordvintsev, Christopher Olah, Mike Tyka., 2015). We ask the network to pick a particular layer and enhance the image to whatever it has learned. Figure 5.6a and 5.6b show the images generated from using the last two inception modules. The network is responsible to pick the learned features to generate the images. Here we input a particular image. The image used to generate 5.6a and 5.6b is shown in Figure 5.1a. The input image adds constraints on the statistics such that the layer will try to generate an image close to the input image. We can see from the images that the higher layers identify more sophisticated features than the lower levels.



(a) Image generated from the second last inception module.

(b) Image generated from the last Inception Module.

CHAPTER 6

CONCLUSION

6.1 Conclusion

In this thesis, we presented a novel method for extending GoogLe Net (Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, 2015), yielding an end-to-end system for multi-label classification. In particular, we proposed a model for recognizing activities which uses Convolutional Neural Networks to map the pixels of the video frame to elements of the activity. Thus, our model demonstrates how GoogLe net can be tuned and extended for video domains. In an extensive experimental evaluation, we showed that our approach can provide very high accuracies for detecting activities. The key reason for this high accuracy is that we exploit temporal consistency in video data to generate a large dataset which allows us to perform frame-by-frame learning. Our approach also allows us to handle variable-length input and output while simultaneously modeling temporal structure. We also showed how we can test the convergence of our proposed model and visualize the learned features at different layers of the network.

6.2 Future work

Future work involves using the temporal information in videos by considering multi-frame input models: where we can input multiple frames to the network, specifically frames over a specified playback time or frames sampled from predefined temporal intervals. We would also like to stack a Probabilistic Graphical Model (PGM) on top of our network where the output elements of our models act as evidences for the variables in the PGM; which can be used to predict activities in the next frame, map the elements of the activity to a natural language sentence (video narration) and for generating video description.

REFERENCES

Yann LeCun, Yoshua Bengio (1998) Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, p. 1-11.

Lecun, Y. (1992) A theoretical framework for back-propagation. *In P. Mehra, & B. Wah (Eds.), Artificial neural networks: Concepts and theory Los Alamitos, CA: IEEE Computer Society Press*

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian (2016) Deep Residual Learning for Image Recognition. *Computer Vision and Pattern Recognition (CVPR)*, p. 3-8.

Marcus Rohrbach, Wei Qiu, Ivan Titov, Stefan Thater, Manfred Pinkal, Bernt Schiele (2013). Translating Video Content to Natural Language Descriptions. *IEEE International Conference on Computer Vision*.

Translating Videos to Natural Language Using Deep Recurrent Neural Networks (2014). *Computer Vision and Pattern Recognition (cs.CV); Computation and Language (cs.CL)*.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich (2015) Going Deeper With Convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

M. Lin, Q. Chen, and S. Yan. (2013) Network in network. *Network in network.* CoRR, abs1312.4400, 2013.

ILSVRC synsets. *http://image-net.org/challenges/LSVRC/2014/browse-synsets*.

S. Ji, W. Xu, M. Yang, and K. Yu (2013). 3D convolutional neural networks for human action recognition. Pattern Analysis and Machine Intelligence. *IEEE Transactions on, 35(1):221? 231, 2013. 2, 4*.

A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. (2014). Large-scale video classification with convolutional neural networks. *In CVPR, 2014. p. 2, 4, 5, 12*.

Anna, Rohrbach, Marcus, Qiu, Wei, Friedrich, Annemarie, Pinkal, Manfred, Schiele, Bernt. (2014). Coherent Multi-Sentence Video Description with Variable Level of Detail. *German Conference for Pattern Recognition (GCPR)*.

Christian Szegedy, Sergio Guadarrama. (2014). bvlc_googlenet.caffemodel. *http://dl.caffe.berkeleyvision.org/bvlc_googlenet.caffemodel*.

Sebastian Ruder. (2017). An overview of gradient descent optimization algorithms. *arXiv:1609.04747v2* .

Diederik P. Kingma, Jimmy Ba. (2014). Adam: A Method for Stochastic Optimization. *3rd International Conference for Learning Representations, San Diego, 2015*.

Alexander Mordvintsev, Christopher Olah, Mike Tyka (2015). Inceptionism: Going Deeper into Neural Networks. *Google Research Blog*.

BIOGRAPHICAL SKETCH

Mahesh R. Shanbhag is a masters student at the University of Texas at Dallas. He completed his bachelors from B.V.Bhoomaraddi College of Engineering and Technology affiliated to Visvesvaraya Technological University.