Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Predicting Human Behavior from Public Cameras with Convolutional Neural Networks

Master thesis

2018
Bc. Ondrej Jariabka

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Predicting Human Behavior from Public Cameras with Convolutional Neural Networks

Master thesis

Study program:	Informatics
Field of study:	2508 Informatics
Department:	Department of Informatics
Supervisor:	prof. Ing. Igor Farkaš, Dr.

Bratislava, 2018
Bc. Ondrej Jariabka

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Ondrej Jariabka
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Predicting Human Behavior from Public Cameras with Convolutional Neural Networks
Vyhodnocovanie ľudského správania z verejných kamier pomocou konvolučných neurónových sietí

Anotácia: Automatická detekcia hrozieb je jedným z najdôležitejších aplikácií počítačového videnia. V poslednej dobe bolo preukázané, že metódy učenia hlbokých neurónových sietí môžu produkovať zaujímavé výsledky pri problémoch počítačového videnia. Cieľom tejto práce je preskúmať možné použitie týchto metódik problému predikcie ľudského pohybu vo verejných priestoroch zo snímok extrahovaných priamo z videa.

Cieľ: 1. Preštudujte príslušnú literatúru a súčasný stav metód analýzy davu, predikcie ľudského správania a pohybu.
2. Navrhnite a vyhodnoťte model uvedeného problému pomocou konvolučných neurónových sietí.

Literatúra: Yi S., Hongsheng L., Xiaogang W. (2016). Pedestrian Behavior Understanding and Prediction with Deep Neural Networks. In European Conference on Computer Vision. Springer International Publishing.
Zhang Y. et al. (2016). Single-image crowd counting via multi-column convolutional neural network. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.

Vedúci: prof. Ing. Igor Farkaš, Dr.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 15.12.2016

Dátum schválenia: 19.12.2016 prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Ondrej Jariabka
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science, Informatics
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Predicting Human Behavior from Public Cameras with Convolutional Neural Networks

Annotation: Automatic threat detection is one of the most important applications of computer vision. Recently, deep learning methods have been shown to produce interesting results when applied to computer vision problems. The aim of this thesis is to investigate a possible application of these methods to the problem of predicting human movement in public areas from raw frames of surveillance video.

Aim: 1. Study the relevant literature and current state-of-the-art methods for the problem of crowd analyses, prediction of human behavior and movement.
2. Design, implement and evaluate a novel model for the aforementioned problem using convolutional neural networks.

Literature: Yi S., Hongsheng L., Xiaogang W. (2016). Pedestrian Behavior Understanding and Prediction with Deep Neural Networks. In European Conference on Computer Vision. Springer International Publishing.
Zhang Y. et al. (2016). Single-image crowd counting via multi-column convolutional neural network. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.

Supervisor:	prof. Ing. Igor Farkaš, Dr.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	15.12.2016
Approved:	19.12.2016

prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Acknowledgement: I would like to thank my supervisor prof. Ing. Igor Farkaš, Dr. for his advises and consultations during writing of the thesis. I would like to also thank my family and friends for their help and continuous support.

Abstrakt

Predpoveď ľudského správania a analýza davu sú jednou z najdôležitejších aplikácií počítačového videnia. Nedávne pokroky v tejto oblasti ukazujú, že metódy hlbokého učenia prinášajú zaujímavé výsledky, aplikované na problémy počítačového videnia. V našej práci skúmame možné použitie týchto metód na problém predpovedania pohybu chodcov z videozáznamu zachyteného dozornou kamerou zameranou na verejný priestor, ako sú stanice metra, vlakov alebo autobusov. Na predpovedanie budúcej masky umiestnenia ľudí sme použili konvolučné neurónové siete. V našom prístupe sa zameriavame na predpovedanie budúcich lokací chodcov iba z jednotlivých obrazkov videa, kde naše modely nemajú žiadne informácie o predchádzajúcom pohybe chodcov. Taktiež prezentujeme naše výsledky a poskytujeme rozsiahlu analýzu chýb natrénovaných sietí a problemu ako takého.

Kľúčové slová: predpoveď pohybu chodcov, predpovede správania ľudí, analýza davu, hlboké učenie, neurónové siete, konvolučné neurónové siete, konvolučné enkodéry

Abstract

Prediction of human behaviour and crowd analysis is one of the most important applications of computer vision. Recently, deep learning methods have been shown to produce interesting results when applied to computer vision problems. In our work we investigate possible application of these methods to the problem of predicting a pedestrians movement from the video captured by surveillance camera capturing public space, such as metro, train or bus stations. We used convolutional neural networks to predict a future mask of pedestrians locations. In our approach we focus on prediction of these locations from raw video frames where our models do not have any prior information about each pedestrian movement. We present our results and provide extensive error analysis of trained networks and the problem as such.

Keywords: pedestrian movement predictions, human behavior prediction, crowd analysis, deep learning, neural networks, convolutional neural networks, convolutional encoders

Contents

List of Figures

Chapter 1

Introduction

In our work we will focus on a problem of pedestrian behavior modeling. We will try to predict a future movement of individual pedestrians currently present in the scene that is being captured with surveillance camera. Such a predictor then can be used for pedestrian behavior analysis, classification of pedestrian behavior, surveillance or outlier detection. One can predict future pedestrian paths and use this prediction to classify pedestrian movement as outlier and alert to such a behavior.

Path prediction is a difficult task because pedestrian behavior and consequent movement is influenced by many factors, such as other pedestrian moving across the scene, large crowd groups present in the scene or other points of interest. This problem was already addressed in the past with various techniques, but until the last decade this problem did not receive enough attention. Recent advances in computer vision brought interest to similar topics, for example, crowd analysis, crowd counting, behavior modeling etc.

In recent years a lot of attention from computer vision community was also dedicated to machine learning and deep neural networks. These models where used to solve various tasks ranging from classification of objects in the scene [9] to answering human given questions about pictures shown to the network [10]. Advances in this field also led to applications of deep learning techniques for previously mentioned similar topics where neural networks were used for crowd counting [3] or our task of pedestrian path prediction [5].

The main theme in works focusing on our topic is a prediction of future pedestrian locations from given exact previous pedestrian coordinates. This approach presents a problem since information about each individual pedestrian location is usually very hard to obtain and other techniques have to be used to extract individual pedestrian coordinates. In many cases, techniques used for a such task of extracting pedestrian coordinates, fail due to various factors related to the scene, for example, lighting condition, scene layout and perspective or high variance in pedestrian representation in one

image (pedestrians closer to the camera appear larger than pedestrians farther away).

We tried to solve this problem by using a deep convolutional neural networks that were trained on previous *raw frames* extracted from a video captured by a surveillance camera. We trained and tested our networks on a dataset proposed in [5], which is one of the largest datasets dedicated to this topic. Our models did not get any information about previous pedestrian locations and had to extract this information from frames of surveillance video. Trained models had to also make prediction based on extracted pedestrian locations. Using a convolutional neural network has many benefits as it allows us to combine various factors influencing individual behavior into one model that is also driven by the data. For example, convolutional neural network can use its learned filters to extract features about current layout of the scene as well as find each pedestrian location and try to predict future walking pattern in relation to this information.

We structure our work as follows. In Section 2 we describe current body of literature dedicated to previously mentioned topics of pedestrian behaviour modeling and crowd analysis. In Section 3 we introduce some basic machine and deep learning methodology to make our work more self contained. In Section 4 we analyze the dataset used in our work as well as describe our data processing steps. In Section 5, we present architectures of proposed and tested models. In Section 6 we present our results and provide detailed error analysis of our approach.

Chapter 2

Related Work

Topics of predicting human behavior and crowd analysis gained traction in the past decade, mainly because of the recent advances in computer vision. There has been a large number of works focused on motion tracking, crowd counting, behavior analysis, path planing, etc. Despite being a principal technique for various applications [1] and consistent attention given to these topics by computer vision community, many of these problems still remain unsolved. Although many advances in this field have been made, there is still insurmountable gap between machine intelligence and human capability to solve these problems [11]. One of the main topics in this field, which lately gained popularity is pedestrian behavior modeling, which can be used for various applications including behavior prediction [12], pedestrian detection and tracking [13, 3], crowd motion analysis [14], abnormal detection [15] and pedestrian path prediction [5].

2.1 Pedestrian Walking Behavior Modeling

Modeling pedestrian behavior is a challenging task, because pedestrian path can be influenced by many factors, such as decision making of individuals as subjects to "social force" [1] or by surrounding environment, for instance, by stationary or moving pedestrians [12]. Another factor affecting pedestrians is historical motion statistics of a scene.

2.1.1 "Social force" model

One of the first works in this field is [1], which introduces a model for describing pedestrian decision making while walking. This model is called "social force" model and is based on earlier works that try to model pedestrian behavior in terms of physics of gases and liquids, such as gas-kinetic pedestrian model. This "social force" model is

[1] Such as surveillance, tracking, autonomous driving for instance.

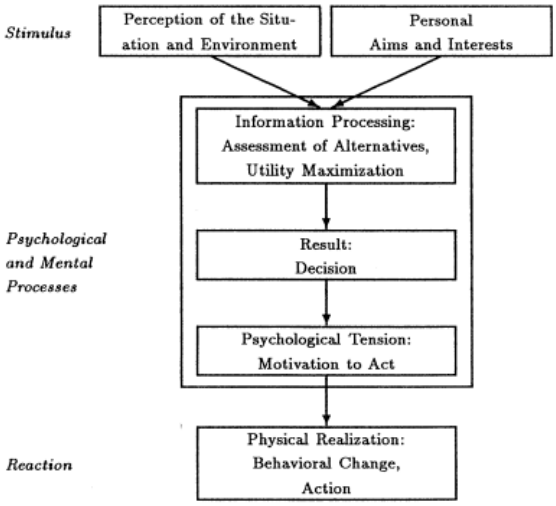


Figure 2.1: Graph of decision making of individual pedestrian based on "social force" model, Helbing, et al. [1].

based on the idea that pedestrian behavior is not chaotic but governed by simple "social forces" as internal motivations of certain movements (Figure 2.1). "It is suggested that the motion of pedestrians can be described as if they would be subject to "social forces." These forces are not directly exerted by the pedestrians' personal environment, but they are a measure for the internal motivations of the individuals to perform certain actions (movements)" [1]. Authors propose that the individual path is influenced by three factors. First, person is motivated to reach certain destination as comfortable as possible. This means that individuals tend to choose shortest paths without taking any detours if there are no obstacles in their paths. This path will usually have a shape of a polygon. Second, pedestrian is repulsed by other pedestrian, for instance, people tend to feel more uncomfortable closer they get to other unknown person or group of people. Pedestrians tend to also keep some distance from borders of walls or buildings. Authors model this repulsive behavior as an ellipse around each person. Third, person is attracted to points of interest, such as, friends, street artists or window displays. Each of these "forces" is expressed as mathematical equation. The "social force" model is then constructed by combining all of these equations with some fluctuation term, which is introduce to cope with fluctuations in individual pedestrian behavior.

2.1.2 Motion statistic models

While "social force" model can predict a path based on pedestrian decision making, it doesn't take into account historical motion statistics of a scene. There have been many works focused on this part of a problem. Many of these works used probabilistic models such as Bayesian networks or Hidden Markov Machines to generate *flow maps* of the scene and to extract and categorize different activities. In [16] authors present

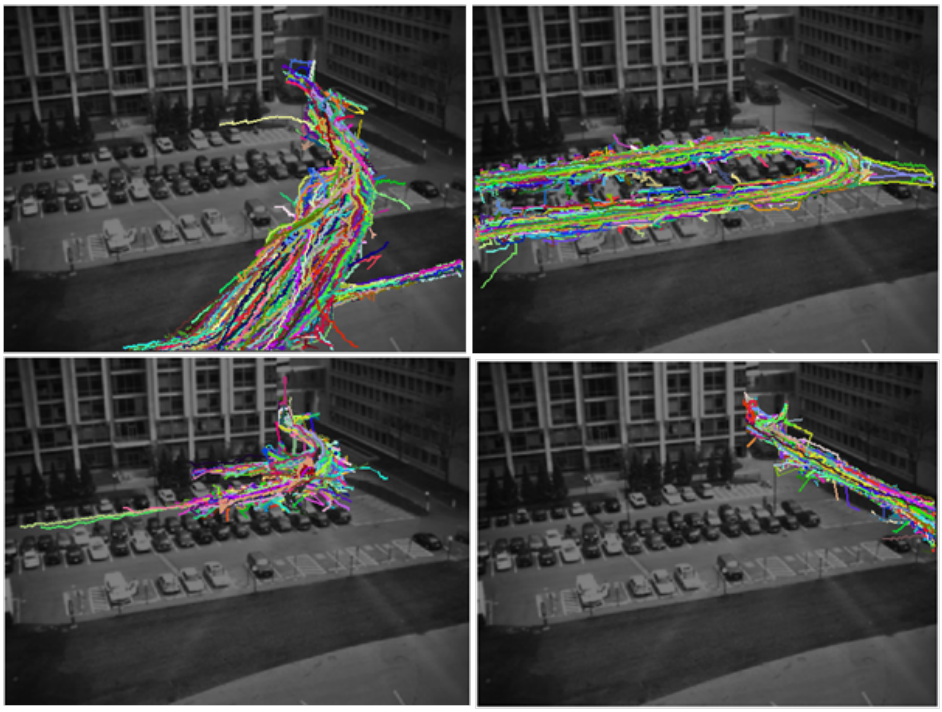


Figure 2.2: Cluster trajectories used model by Dual-HDP, X. Wang, et al. [1].

novel approach to trajectory clustering as methods to learn motion from the given scene. Authors propose a framework for unsupervised trajectory analysis and semantic region modeling using nonparametric Bayesian model - Dual Hierarchical Dirichlet Processes (Dual-HDP). This is done by treating trajectories as documents and individual observations on the trajectories as words. Trajectories and semantical regions are jointly modeled in Dual-HDP and learned using Dirichlet Processes. Found trajectories are then clustered based on the same semantical regions (Figure 2.2). Authors use this process to detect abnormal activities in surveillance settings. In [2] authors also construct flow map of the scene (Figure 2.3) by running rotation invariant multiview face detector proposed in [17]. This detector is used on individual frames of the video where over a number of frames constructs a probability map, which is then again used to guide the face detector. This is done by firstly setting the whole frame to have an equal probability of finding a top part of a human body. This probability is then slowly reduced to flow regions. Authors use head and shoulder detector to find only top parts of a human body as it is the most distinguishable part and also partly address the problem of occlusion that is commonly present in surveillance settings.

2.2 Machine learning

With growth of computer science and increase of computational power in recent years, the field of machine learning became one of the most desired and fastest growing fields

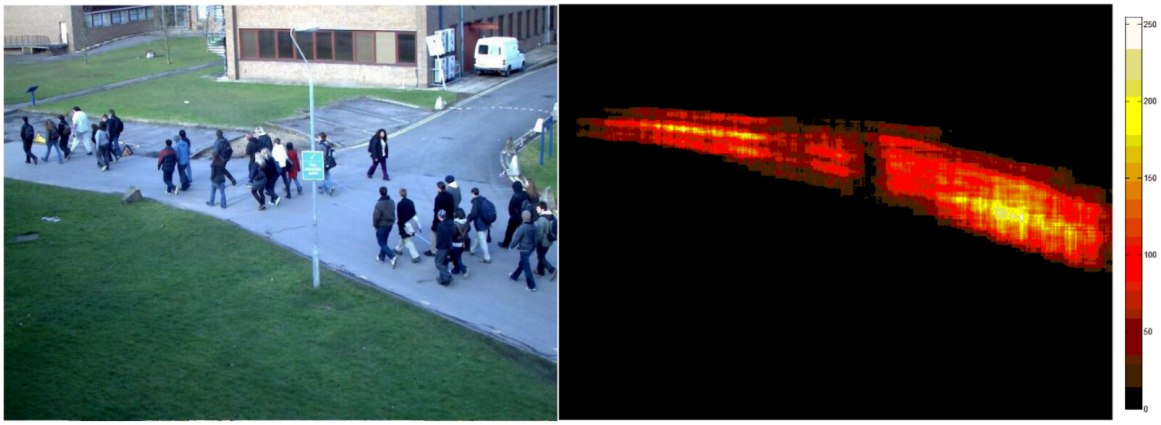


Figure 2.3: Constructed flow map of the scene model by multiview face detector, Xing, et al. [2]

in computer science. Its core idea is to create a model that is able to extract information from historically known data (datasets) and based on this data is able to correctly make prediction, classify the data or find previously hidden structures in the data. There has been much new advancement in this field in the past decade that revolutionized this field and made it more popular.

2.2.1 Behavior modeling

Work [12] is based on this principle, where authors used main concepts of "social force" models together with learnable parameters from data. Similarly to [1], authors propose three forces that influence pedestrian behavior while moving through the scene. First is a scene layout, there are many obstacles that can't be crossed. These obstacles generate repulsive force on pedestrians while moving trough the scene. Next, authors propose that pedestrian, as in "social force" model, is influenced by other moving pedestrians, which also generate some repulsive force. Last force exerted on a pedestrian is by stationary crowd groups, which are present in the scene. This work mainly focuses on the last aspect.

The authors speculate, that pedestrians are mostly influenced by stationary crowd groups that naturally form in the scene. Another difference between previously mentioned "social force" model (Section 2.1.1), is the absence of attractive force which drives people closer to points of interest in the scene. Every force that influences each pedestrian is then modeled by equations similar to those in "social force" model and is weighed by parameters $\theta_1, \theta_2, \theta_3$ respectively. These parameters are learned from data. Based on this authors construct general energy map M, that can be represented also as probability map of how likely is a pedestrian to walk through that spot in the scene, for example, obstacle in the scene that can not be crossed will have energy 0. Also

pedestrians that walk through the crowd groups are penalized. For this purpose the last parameter, group density weight θ_4 is introduced.

Since behavior of every pedestrian can be different, energy map M_P of every individual pedestrian is constructed. Parameter P, is first estimated based on previous data and similarity of walking patterns and trajectories. All the pedestrians can be classified into three categories based on their walking behaviors: aggressive, conservative, and abnormal. Aggressive pedestrians prefer to walk directly to their destinations. Conservative pedestrians prefer to walk longer paths to avoid close contact with others. Pedestrians who take long routes to their destination and also conservativeness no longer properly describe behavior of these pedestrians are classified as abnormal. Constructed maps are then used to predict paths of each individual. Based on similarity between the predicted path and true movement, abnormal behavior is detected. Authors also show from learned weights that stationary crowd groups have highest influence on pedestrians behavior and his/her walking patterns. Another contribution of this work is a large dataset of crowd scenes.

2.3 Deep learning

In recent years there has been a lot of attention dedicated to deep learning and to neural networks in general. Recent advances made it possible to train deeper models. Attention given to this topic sparked the creation of large datasets as well as the new and robust features and new models.

For instance, [14] constructed one of the largest and robust crowd scene datasets consisting out of 10000 videos from 8257 crowded scenes, and building an attribute set with 94 attributes that were used to label scenes depicted in given videos. They also constructed a deep convolutional neural network for labeling given scenes and proposed new features which can be extracted from individual frames of the video and can better describe the scene. These new features served as input to this model. Authors also created a study to evaluate human performance on the constructed dataset and compared it to their trained model.

Another dataset was proposed in [3] which focuses on crowd counting from single image. The dataset consists out of 1198 images with 330000 annotated heads. It is one of the largest datasets collected for this purpose to this date.

2.3.1 Crowd counting

Many of the works in the current body of literature, such as [2], use detector based crowd counting. People typically assume that a crowd is composed of individual entities which can be detected by some given detectors. The limitation of such detection-based

methods is that occlusion among people in a clustered environment or in a very dense crowd significantly affects the performance of the detector. Most frequent approach to crowd counting is the use of feature-based regression. The main step of this approach consists out of segmenting the foreground, extracting features from the foreground and utilizing a regression function to estimate the crowd count. These methods yield decent performance even with simple models such as linear regression. Or if they utilize a more complex model they usually require additional information about the scene.

Authors in [3] propose new Multi-Column Convolutional Neural Network (MCNN) for the problem of crowd counting. There are generally two approaches to crowd counting using neural network models. First, models whose input is an image and expected output is a headcount. Second approach is to use density map, from which headcount can be derived. Model takes image as input and produces density map as output. Authors choose the second approach because of two key advantages. "Density map preserves more information. Compared to the total number of the crowd, density map gives the spatial distribution of the crowd in the given image, and such distribution information is useful in many applications." And secondly, "In learning the density map via a CNN, the learned filters are more adapted to heads of different sizes, hence more suitable for arbitrary inputs whose perspective effect varies significantly." Since proposed dataset contained only label heads, authors also proposed a method for translating labels into a density map which is used as target variable in training of the MCNN. Each pixel in the image that contains a head is represented by some delta function δ. This is then convolved with a Gaussian kernel with parameters σ, which is based on the size of the head for each person within the image. This parameter is introduced to estimate geometric distortion caused by distortion between the ground plane and the image plane.

The model proposed in this paper is a stack of Convolutional Neural Networks (Figure 2.4). Each consists of 4 layers (conv-pool-conv-pool). Every network has filters of different size. These smaller networks are then stacked in column architecture and the outputs are merged together to produced a final density map. Each network was first trained separately by mapping the input directly to density map. After combining these models into one network with a final merge layer, the whole network was then fine-tuned on the proposed dataset. This architecture was chosen mainly based on the fact that many scenes contain heads of different sizes. This fact is represented by different sized filters for every smaller network. This is also due to the geometric distortion in the image, that also produces heads of different size.

The proposed architecture has main advantage against a single CNN. It can be easily adapted to different sets of conditions (head of different sizes based on different scene geometry), mainly due to the filters of different size that corresponds to different head sizes. Authors train this model on their proposed dataset and tested it on previ-

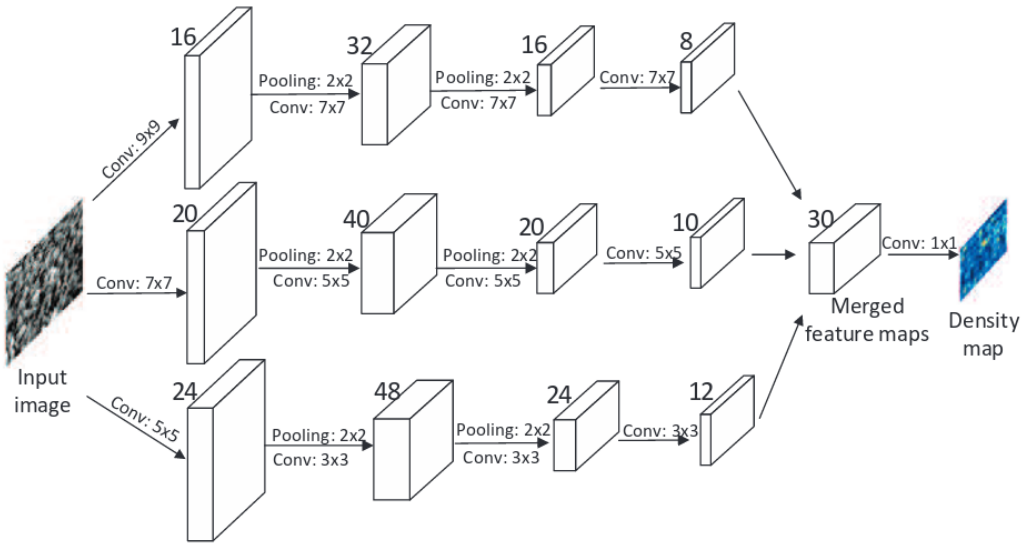


Figure 2.4: Architecture of Multicolumn Convolutional Neural Network proposed by Y. Zhang, et al. [3]

ously known smaller datasets. They experimented with different approaches to transfer learning their model. Results were better than previously proposed state-of-the-art on individual datasets. This also shows that proposed network is easily adaptable to different conditions despite its simplistic design.

2.3.2 Pedestrian detection

In [4] authors used multi-spectral images and Fast R-CNN to detect pedestrians. Authors test 4 architectures of convolutional neural networks on Caltchet pedestrian detection datasets, which all four achieve state-of-the-art performance. First was vanilla CNN which was also used as baseline detector. Next, authors propose Fast R-CNN models that fuse together information from RGB image and thermal images. Three models are proposed each fusing these two sources of information at different feature level (Figure 2.5). Low-level feature fusion model, merges information right after the first convolutional layer. Mid-level feature fusion model (Halfway fusion), uses two branches of basic CNN block to extract from both RGB and thermal image mid-level features (C4-features) and then used Network-in-Network (NiN) to fuse them together. Last model, high-level feature model, uses F7-features of both RGB and thermal images, which merges fully-connected features together. F7-features are conventionally used as new representations of objects. All of these networks are reported to achieve state-of-the-art performance on Caltech dataset. It is worth noting that to authors' best knowledge Fast R-CNN were not used for pedestrian detection before. From ex-

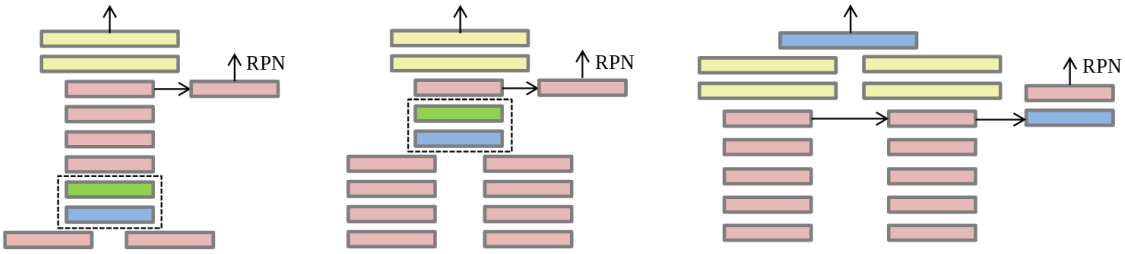


Figure 2.5: Three proposed architectures of Multispectral Fast R-CNN proposed by Liu, et al. [4], each merging two sources of input at different feature level. Low-level feature fusion model (left), halfway fusion model (middle), high-level feature model (right).

periments best results yielded Halfway model, which outperformed by small margin two other proposed fusion models.

2.3.3 Pedestrian path modeling

While previous methods all focused on few of the previously mentioned aspects of pedestrian behavior, in [5], authors combine all of these aspects of behavior modeling into a single convolutional neural network. Authors argue that the main problem with pedestrian path prediction using neural networks is how to efficiently encode walking information of pedestrians in scene throughout multiple frames. Straightforward way by using dense optical flow maps to describe motions of a whole frame will lead to ambiguities when merging and splitting events happen frequently in crowded scenes. That is why special pedestrian path encoding scheme is introduced in this work. Every pedestrian path throughout a set of frames up to some time t_m is expressed as a vector. These vectors are then encoded into displacement volume that is used as input to newly proposed convolutional neural network (Figure 2.6). The volume vectors consist of normalized spatial (x and y position in the frame) information x_i^m of individual pedestrians, where $m \in t_1, \ldots, t_m$ and $i \in 1, \ldots, N$ where N is the number of pedestrians present in the frame. Each position x_i^m is normalized by the width and height of the frame. This then creates $d_i \in \mathbb{R}^{2M}$ for every pedestrian i. Input to CNN is then 3D displacement volume of size $\mathbb{R}^{X \times Y \times 2M}$ based on d_i.

Authors then propose Behavior-CNN (Figure 2.6) that takes this input and produces same displacement volume but at times $t_M + 1 \ldots t_M + M$ which are future time points to predict. Behavior-CNN contains three bottom convolution layers, one max-pooling layer and an element-wise addition layer, three top convolution layers, and one deconvolution layer. Convolutional layers are followed by ReLU nonlinearity. Due to high sparsity of the input data new layer-to-layer learning scheme is propose. To avoid the possibility that the network may converge to a bad local minimum, if all the pa-

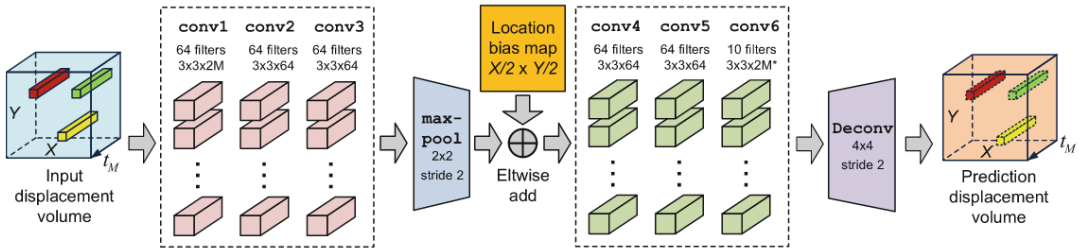


Figure 2.6: Architecture of Behavioral CNN proposed by S. Yi, et al. [5]

rameters are trained together, a simpler network with three convolution layers is first trained until convergence. Afterwards, the trained convolution layers are used as the bottom layers of Behavior-CNN. The remaining layers are then added. Lastly, all the layers are jointly fine-tuned.

The network was trained on two datasets with annotated walking paths and evaluated against state-of-the-art models. The network outperformed all state-of-the-art models on both datasets. Authors also added bias map of the scene to the input so that the network can more easily extract layout of the scene. This helped the network achieve lower error but hindered model transfer ability.

Location awareness property of the network was also tested. This experiment is conducted to test if the network has location invariant property. All of the testing samples are flipped horizontally and/or vertically. If predictions of the model has location invariance, flipping all the pedestrian paths at all spatial locations, in the same way, will not make difference on prediction errors. However this was not the case since testing error increased when images were flipped. Authors argue based on these results that different locations have different dependence on moving directions.

Authors also provide a deep analysis of the trained network as well as lower and high-level features. They also argue that pedestrians are highly influenced by other people in the scene. This is demonstrated by testing other network architectures with smaller filter sizes. Where networks with larger filters outperformed networks that had smaller filters. This is to show that larger filters can capture more information around pedestrians and capture activities around them which leads to better performance over all. Current filters are 10% of the scene. Authors also analyze filters at different depth of the Behavioral-CNN. They show that stationary groups also highly contribute to ones decision making when walking through a scene. Filters corresponding to higher level features, show high activation response to stationary pedestrians or groups of pedestrians. Filter corresponding to lower level features slowly learned to distinguish different motion patterns and spatial location for pedestrians in the scene.

In test settings, paths that where used in displacement volume where extracted with *Kanade–Lucas–Tomasi* (KLT) feature tracker [18]. Network was as part of the

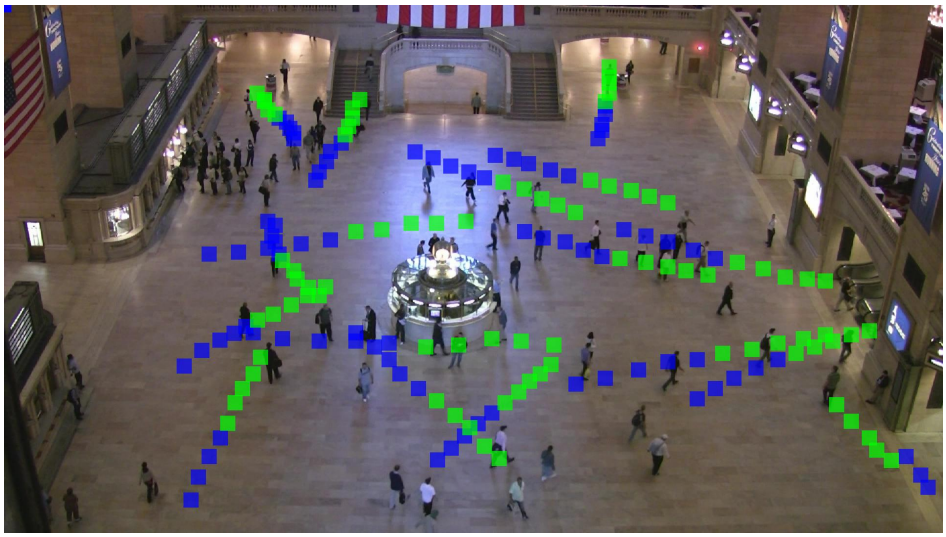


Figure 2.7: Paths proposed by Behavioral CNN, S. Yi, et al. [5]

experiment also trained on only paths extracted from KLT tracker. Although the error of the network was higher it was not by a large margin and the network still made correct predictions and displayed ability to correct fragmented or early terminated trajectories that is often case with KLT tracker. In current settings the best proposed model predicts trajectories of individual pedestrians 5 time points (4 seconds) ahead (Figure 2.7). Long-term prediction can be made by recurrently using predicted volumes as input to the network. The network was also evaluated in destination prediction. The destination is determined as the nearest exit to the predicted future walking path. Behavioral-CNN also achieved state-of-the-art results where it achieve Top3 accuracy of 84%.

As part of this work we try to replicate the results. Unfortunately, we were unable to train the proposed network and therefore replicate the reported results. The main reason was, as mentioned before, high sparsity of the input data that is construed by proposed encoding. This led to the network that learned to predict only zeros even when trying aformetioned layer-to-layer greedy learning scheme. The network after the first couple of iterations set all weights to zero and learning stopped. We also think, that proposed encoding, although being good for encoding pedestrian movement does not capture all the aspects of the scene. The scene is only represented by learnable bias map. This might underrepresent the true dynamics of the scene. Also, the preprocessing step uses, during test time, a KLT feature tracker which might extract various other interesting points from the scene and skip the walking pedestrian. The network is never trained to find individual pedestrian, only to repair mistakes made by the KLT and therefore is very depended on the performance of the tracker.

The main contribution of our work is to somewhat combine two of the proposed methods. Our goal is to merge Behavioral-CNN proposed in [5] and Multi-Column

Convolutional Neural Network proposed in [3] to remove the need of constructing displacement volume of all spatial location of all pedestrians present in the scene as input to our convolutional neural network. And train the network from raw frames, were network will be able to extract individual locations of the pedestrians and predict their movement. We believe this could then simplify the process of pedestrian path prediction.

Chapter 3

Methodology

Methods utilizing historical motion statistics of a scene often lack decision making process of pedestrian that is provided, for instance, by using "social force" models or other agent based models [19]. On the other hand, methods utilizing internal processes of pedestrian moving through the scene, lack the deeper understanding of the layout present in the scene. Many works in current body of the literature focus on one specific part of a problem while ignoring other aspects. In this work we are going to focus on machine learning approach to this problem. Machine learning can be used to extract and combine multiple aspects of the problem. More specifically in this work we are focusing on *Deep Convolutional Neural Networks*.

In recent years there has been a lot of attention dedicated to deep learning and to neural networks in general. We will discuss some basic methodology in this section. The aim of this chapter is not to explain the neural networks and deep learning in detail, but rather to briefly describe, repeat and approach each of the concepts of deep learning to make this work more self contained. Therefore, we expect the reader to have basic understanding of machine learning concepts, gradient based optimization and neural networks. More thorough explanation of these concepts can be found in [7].

3.1 Neural networks

Feedforward neural networks, or *multilayer perceptrons* (MLPs) represents one of the basic and yet one of the most powerful models of deep learning or rather machine learning in general. They are a basis for many commercial applications, for example, deep convolutional neural networks are used for object detection and recognition or recurrent neural networks are used for language translation or sentiment analysis.

The main purpose of a neural network is to approximate some arbitrary function f'. For basic classification problems these network define a mapping of some input x to class y as $y = f(x; \theta)$, where θ are parameters that network learns through gradient based

optimization to achieve best approximation. The networks are also called *feedforward* because the information flows in forward fashion through the function, meaning it flows from x through intermediate computations of the network to define f.

The neural network typically consists of many different functions stacked upon each other. In other words, one of the main reasons they are called *networks* apart from neurological similarity, is that the model is a *directed acyclic graph* (network) describing how individual functions are chained together. These individual function are called layers. First function in the graph is called input or the first layer, second is called second layer and so on. The last function in the network is called output layer. The final function is then computed by chaining these function together $f(x) = f^{(1)}(f^{(2)}(\ldots(f^{(n)}(x)))$. We also refer to the total number of layers as **depth** of the network. The layers which are in between the input and the output layer are also called **hidden layers**. This name comes from the fact that when training the network the data does not provide desired output to each of the layers and learning algorithm must decide how to adjust each layer in order to best approximate the function f'.

Each layer consists of units representing vector-to-scalar function. These units act in parallel and are called **neurons**. This is due to their resemblance to neurological neurons. The *neural* part of the name stems from this neurological context. The resemblance comes from the fact that each neuron computes its activation which is a vector-to-scalar function, similarly to biological neuron. This function represents neuron excitation. In other words, how much the given neuron responds to particular input. Formally, this is done by computing

$$y = f(\vec{w}x)$$

where w is called a **weight** of the neuron and provides importance mapping from input to output. The function f is called **activation function** and usually is nonlinear function, for example, a sigmoid function was often used. These neurons are then organized into layers and weights of each neuron can then be put into a matrix. This is very efficient since we can now compute each neuron activation in the layer by vector, matrix multiplications

$$\vec{y} = \hat{f}(W^T x)$$

where W are called *layer weights*. The networks were historically guided by neuroscience. Although, modern neural networks steer away from this trend and are rather guided by mathematical and engineering disciplines.

For regular neural networks, the most common layer type is the **fully-connected layer** in which neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections (Figure 3.1). Connections in between neurons in the same layer are called *lateral connections*. In the design of the network

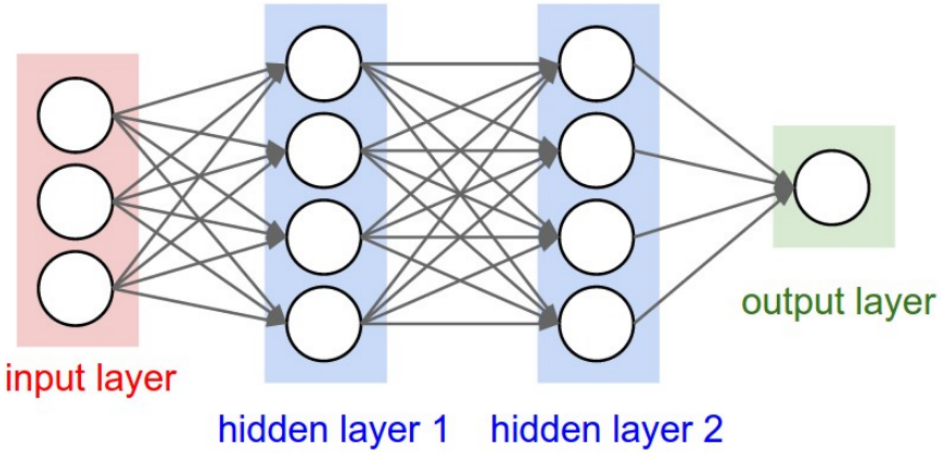


Figure 3.1: A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer[8].

there are multiple parameters that are arbitrarily chosen prior to training and are usually based on experiments with the network. We called these **hyper-parameters**. In typical feedforward network we choose the depth of the network, number of units per each layer also called **width** of the network. Each neuron has its **activation function** which are commonly non-linear functions.

Unlike all other layers, the output layer neurons typically do not have assigned any activation function (or we usually think of them as having a linear identity activation function). Output layer is usually taken to represent the class scores (e.g. in classification), which are arbitrary real-valued numbers, or some kind of real-valued target (e.g. in regression) and therefore we do not introduce any non-linearity to these layers[2]. The neural networks usually learn (update layer weights) through the process called **back-propagation**. This process is done with use of *computational graphs*. Description of this process is beyond the scope of this work and thorough explanation can be found in [7]. It is also worth noting that neural networks with at least one hidden layer with non-linear activation function are *universal approximators* of continuous functions on a discrete set of points [20].

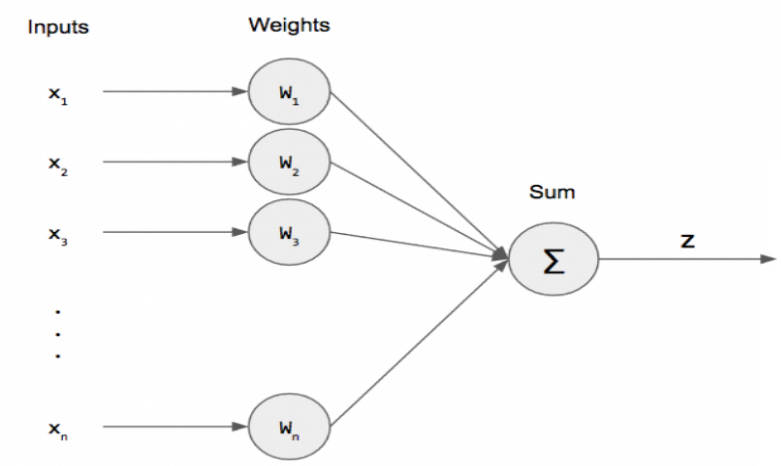


Figure 3.2: A simple perceptron with weights $w_1 \dots w_n$ taking input $x_1 \dots x_n$ and producing output activation z.

3.2 Units

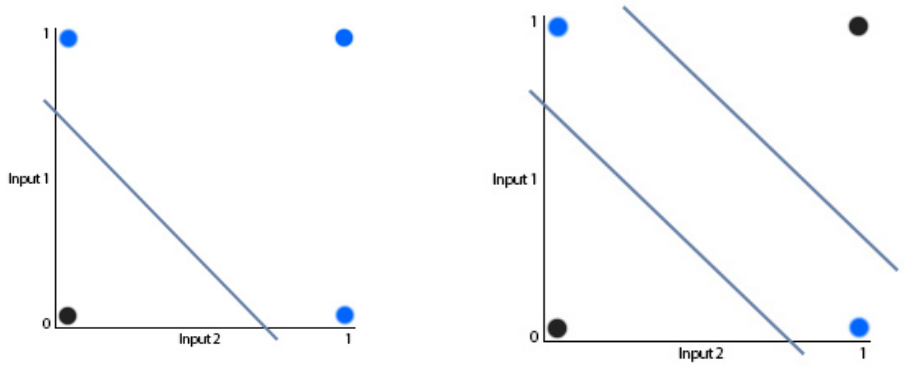
3.2.1 Perceptron

Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts. It is the basic artificial neuron. A perceptron, as shown in Figure 3.2, takes several binary inputs $x_1, x_2, \dots x_n$ and produces a single binary output. To compute the output of the perceptron we use *weights*, which are real numbers, $w_1, w_2, \dots w_n$, weighing importance of respective inputs to the output. The neurons output, which can be 0 or 1 is then determined whether the weighted sum $\sum_j w_j x_j$ is greater or lower then a given threshold.

$$output = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases} \tag{3.1}$$

Single perceptron can be thought of as a device that makes decision based on weighing up different inputs to the simple problem or also logical gate - a device that can compute elementary logical functions, `AND, OR, NAND,` etc. In the modern sense a perceptron is a binary classifier that maps its input $\vec{x}$ (a real-valued vector) to a single binary value. We can obtain the most common form of perceptron occurring in modern neural network literature by rearranging the equation shown in Figure 3.1 as follows

[2]In deep neural networks it is common practice to set number of output neurons to be equal to number of classes (for classification tasks) and then use softmax nonlinearity on top of these neurons. Each neuron then predicts a probability of individual class assigned to it with values ranging from 0 to 1.



(a) OR

(b) XOR

Figure 3.3: Simple boolean function separated by a one and two lines representing a decision boundaries separating classes.

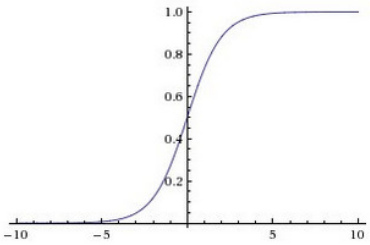
$$output = \begin{cases} 1 & \text{if } \sum_j w_j x_j + b > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

where w are weights, x is the input and b is called intercept term or more commonly know as **bias**. This term acts as offset that shifts the decision boundary and its independent from the input. We can view a single perceptron as the simplest feedforward network. It can correctly classify linearly separable problems. By stacking multiple perceptrons we get *multi-layer perceptron* and thus extend set of problems it is capable of solving. For instance, the single perceptron is capable of solving simple boolean expression separable by a single line as shown in Figure 3.3a. More complicated boolean expressions where individual classes can not be separated by a single linear line, such as, `XOR` are solvable by stacking two or more perceptrons on top of each other, as shown in Figure 3.3b.

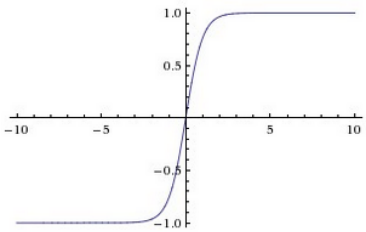
In the context of neural network we stack many of these or similar neurons often accompanied by non-linear activation functions into layers to solve complex problems. One of the main reasons networks are organized into layers and main advantage is that we can then compute these calculation in matrix vector operations. Individual weights of neurons are organized into matrices and bias into vector of real numbers. Also the input can now be a vector that holds multiple training examples and thus speeding up the gradient optimization. The forward pass of a fully-connected layer corresponds to one matrix multiplication followed by a bias offset and an activation function.

$$\vec{y} = \hat{f}(W^T x + \vec{b})$$

where W are layer weights, f is an activation function, b represents the bias vector and x is input to the layer.



(a) Sigmoid function



(b) hyperbolic tangent function

3.2.2 Sigmoid units

Sigmoid unit is a standard neuron similar to perceptron with sigmoidal activation function. Instead of being just 0 or 1 as perceptron, its outputs can take on any values between 0 and 1. Sigmoid function is used to represent the probability distribution over binary variable. We can think of sigmoid neuron as consisting out of two parts. First the standard linear computation

$$z = w^T x + b$$

where w is a weight of the unit b is bias and x is input to the unit. Second, it uses sigmoid activation function to convert z into probability by computing sigmoid (logistic function) defined as follows

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}$$

Sigmoidal units - saturate when z is very positive - saturate when z is very negative (Figure 3.4a). They are only very sensitive when z is close to 0. This widespread saturation has proven difficult in terms of gradient based optimization. In larger network this leads to phenomenon called *vanishing gradient*, where a gradient flowing through the network becomes very small and learning effectively stops. For this reason their use in more modern neural networks is discouraged and sigmoidal units are often replaced by hyperbolic tangent function, since $tanh(z) = 2\sigma(2z) - 1$.

In practice it was shown that hyperbolic tangent performs better. The function resembles more identity function near 0 (Figure 3.4b). Therefore, in certain cases it resembles more training a linear model which makes gradient optimization easier. The sigmoid function has seen frequent use historically since it has a nice interpretation as the firing rate of a neuron: from not firing at all (0) to fully-saturated firing at an assumed maximum frequency (1).

3.2.3 Softmax units

These neurons use as their activation function **softmax function**. Softmax units naturally represent a probability distribution over a discrete variable with n possible values. So this type of units acts as some sort of a switch. Their are most commonly

used in the output layer but can be in certain cases used inside the model itself as units of hidden layers. For instance, if we wish the model to choose from different classes.

Another way to look at the softmax is as a generalization of sigmoid function which represents a probability of a distribution over a binary variable defined as $\hat{y} = P(y = 1|x)$. Softmax then can be defined as $\hat{y}_i = P(y = i|x)$. The softmax function also consists out of two components. First, a linear layer predicts unnormalized log probabilities for each class by computing $z = W^T h + b$. The softmax function then exponentiates this logarithm and normalizes the output to obtain desired $\hat{y}$. Softmax is formally defined as

$$softmax(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \tag{3.3}$$

where z_i is the output of linear layer for i^{th} class while j goes through all the classes. Softmax units still can saturate and therefore it is numerically unstable. But the softmax function responses to difference between its outputs not on the output itself

$$softmax(z) = softmax(z + c)$$

where c is a constant. Using this property one can derive a numerically stable variant of the softmax

$$softmax(z) = softmax(z - \max_i z_i)$$

Even though the unit can became saturated we can show that it does not suffer from *vanishing gradient* problem, that was present in sigmoid unit, by using *log* to undo the *exp* function which results in the following

$$log\ softmax(z)_i = z_i - \log\big(\sum_j \exp(z_j)\big) \tag{3.4}$$

as we can see the first term of the input z_i will always have a direct affect on the input and therefore this term can not became saturated and learning can proceed even if the contribution of the second term becomes very small. From neurological point of view we can think of softmax as a competition between nearby neurons. When output of one neuron increases the output of other must decrease since softmax always sums up to 1. This kind of *winner-take-all* completion between nearby neurons is believed to exists in the cortex of the brain and is wired into many more biologically plausible models, such as *Self Organizing Maps* [21].

3.2.4 ReLU units

The Rectified Linear Unit is another type of artificial neuron that has become *very* popular in the last few years, mainly in image processing community. They were first

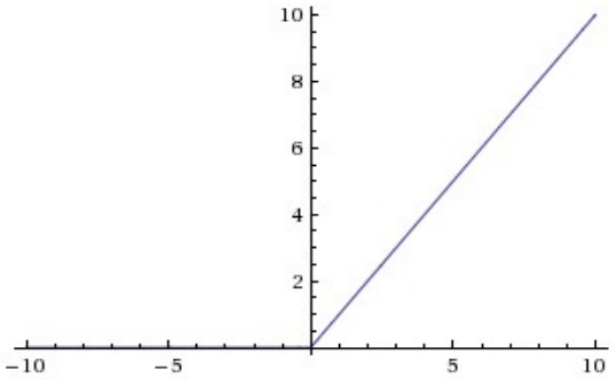


Figure 3.5: The Rectified Linear function

introduced in deep learning context in [9]. Now, the ReLU units or their variants are almost exclusively used in most of the convolutional neural networks, generative models and inside many more. The output is computed by function

$$f(z) = \max(0, z) \tag{3.5}$$

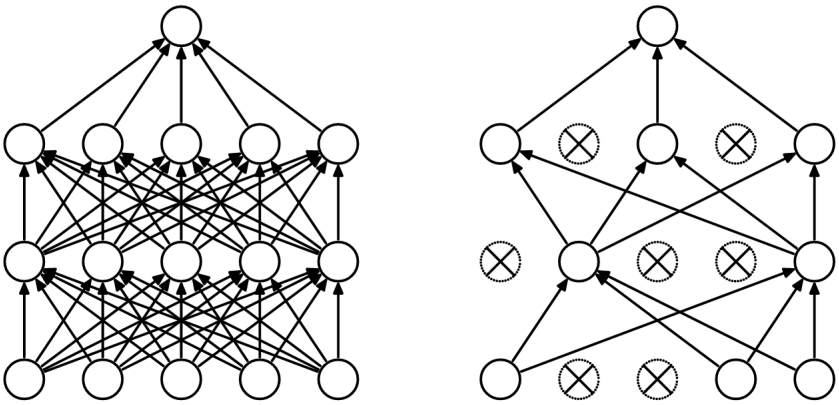
where z is linear activation computed as

$$z = w^T x + b$$

where w is a weight of the unit b is bias and x is input to the unit. To put it simply, the neurons linear activations are simply thresholded at zero (Figure 3.5). Units similarity to basic linear units makes it very easy to optimize. Gradients flowing through the units are large and steady throughout. It has been found that ReLU units can greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid functions. One major drawback of ReLU units is that they can "die" during training, meaning that a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, then the gradient flowing through the unit will forever be zero. Another drawback, is the ReLU units can not learn via gradient base optimization, for training examples whose activations are zero. Several variations of ReLU exist and try to solve these issues, e.g., leaky ReLU [22] or parametric ReLU [23]. Most of them performer comparably, but their computational cost is higher.

3.3 Dropout

One of the most used regularization techniques in modern neural networks is method called **Dropout** [6]. We can think about dropout as a very computationally efficient



(a) Standard Neural Net (b) After applying dropout.

Figure 3.6: An example of thinned net after applying dropout [6]

method of *bagging* (Bootstrap Aggregation) ensemble of even very large models. With bagging we train multiple models and evaluate these models on each test example. Bagging is so called "*meta-algorithm*" that reduces variance of the models. It is a technique that combines predictions from multiple machine learning algorithms together to make more accurate predictions than any individual model. Concretely, we train k models and construct k different dataset that are sample from original dataset with replacement. Then the model i is trained on dataset i. For example, bagging is used together with decision trees to reduce their inherently high variance.

This by the current definitions of bagging seems impossible with deep learning models that can consist out of tens of layers. To train and evaluate tens of models would require extensive computational power, time and memory. Dropout aims to approximate this process. The main difference between dropout and bagging is that different models share their parameters which is not the case with bagged ensemble models. It is also worth noting that even though ensembles are often used in in conjunction with neural networks (usually five to ten, as used in [24]) it is not on the same scale as with usual bagging where it is common to see hundreds of models.

Dropout provides cheap approximation of training and evaluating exponentially many bagged ensembles of neural networks. It is also an inexpensive regularization for a wide family of models. Specifically, dropout trains ensemble of subnetworks, that can be constructed from underlying base (*parent*) network, by removing some of its non-output units, as illustrated in Figure 3.6. In current architectures of neural networks this can be achieved by simple multiplication of the units output by zero. This requires slight modifications to the training algorithm, but has been empirically shown to increase the performance. At the train time for every loaded example we randomly sample a binary mask for all input and hidden units. This mask is sampled independently for each unit. The probability of the unit being activated (sampling a

mask value equal to 1) is a hyper-parameter of the network and it is chosen prior to the training. Typical setting of probability - for input unit to be included is 0.8 - for hidden unit to be included is 0.5.

Forward propagation step remains the same. We have to keep track of all the units and their mask because at the backpropagation, gradient cannot flow through the drop units. This might take $O(n)$ memory, where n is the number of units, while we get to the backpropagation stage. Main advantage of dropout is that it is also very computational cheap. It also takes $O(n)$ computations per example per update, to generate n random samples and multiply activations of the units. Other significant advantage of dropout is that it can be applied to a wide variety of models without any modifications to the algorithm or any significant modification to the model itself.

At the test time we do not want to generate the mask and keep dropping the neurons. So at the test time every neuron sees every input. This presents a problem in term of strength of activations. For example, if we dropped the units during training with probability $p = 0.5$ then the strength of the signal coming into the units is multiplied by factor of 2 during test time. Therefore, we have to scale the activations by the probability p during testing. But performance during testing is crucial. Optimally we would want to perform the scaling during the training and leave the testing part untouched. This is called **inverted dropout** and it multiplies each active unit by $1/p$ at the training to boost the strength of the activation. Then at the test time each unit is getting approximately the same signal strength.

Since the first introduction of the dropout there has been a lot of research devoted to understanding its properties and its similarity to regularization [25]. Dropout falls into a more general category of algorithms. The main theme of these algorithms is to introduce stochastic behaviour to forward pass of the network during training. One example of research devoted to these algorithms is a algorithm similar to dropout called **Drop connect**, where a random set of weights is instead dropped during the forward pass [26].

Dropout can be described not just as an approximation to bagging in models with distributed representations. Dropout does not train just bagged ensembles of models but ensembles of models that share hidden units. This means, together with the fact that we are dropping random units that models learn stronger features, because, each unit has to be able to correctly perform its function without relying on any other unit. To put it in other words, to be able to be swapped for any other unit in the network and perform the same function. This leads to units not only having good features but rather features that perform well in many contexts. In [27] they compare features learned through dropout and features learn with ensemble of independent models. Dropout offers additional improvement over individually learned features. The power of dropout comes from this fact. The features learned through this method have to

be universal. For example, if we train a model to recognize a face, it might do so by training some unit to recognize nose. While using dropout this unit might be dropped and the model has to train another unit to replicate this dropped unit or to learn other features that help it correctly identify face in other units, such as, mouth or eyes. This makes the models more robust and perform better. "This can be seen as form of highly intelligent, adaptive destruction of the information content of the input rather than destruction of the raw values of the input" [7].

It is important to note that this is different from just adding noise to the input. By simply adding noise we can not erase nose representation in the model and force it to leverage all accumulated information so far to make correct prediction. Lastly, the noise added through dropout is multiplicative not additive. If the noise were additive with fixed scale than units activation might just become large enough to make noise insignificant. This is not the case with multiplicative noise and dropout does not suffer from such a problem.

3.4 Learning algorithms

In this section we will discuss basic gradient based optimization techniques used for training of neural networks.

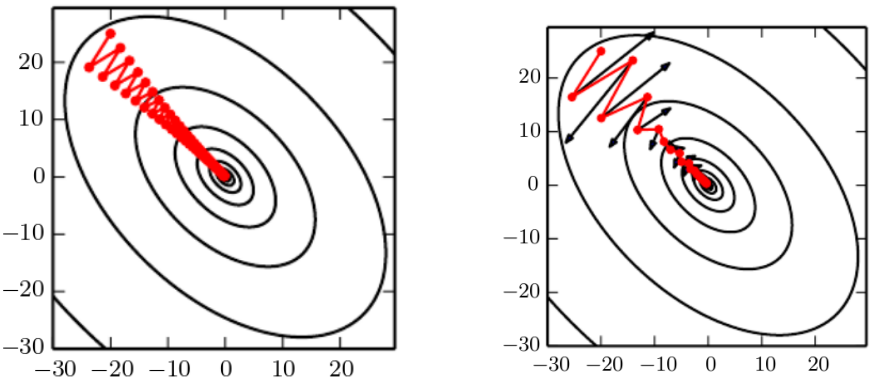
3.4.1 Gradient descent

In standard machine learning gradient optimization we continuously evaluate a *loss* (cost) function, which is a measure how "wrong" is our prediction. We then compute a gradient of this loss function and perform a parameter update. This is called **gradient descent**, because we are moving opposite of the gradient and therefore descending to local optima. In case of neural network update is performed by propagating a gradient throughout the network in back-to-front fashion and updating weights of each layer. As previously mentioned this is called backpropagation. A basic version of gradient descent algorithm is sometimes also called *vanilla* gradient descent. Each step, or in other words parameter update, is performed after evaluating entire dataset and then computing gradient on loss function.

$$\theta = \theta - \alpha \nabla_\theta J(\theta)$$

where θ is parameter vector, $J(\theta)$ is a cost function, $\nabla_\theta J(\theta)$ is the gradient of the cost function w.r.t. the parameters θ and α is a the learning rate.

In large scale machine learning problems, where datasets can have millions of training examples, evaluating a loss function and then computing gradient to perform only a single parameter update becomes unfeasible. We can therefore modify this algorithm



(a) Optimization with standard gradient descent

(b) Optimization with momentum.

Figure 3.7: Optimization with SGD and Momentum SGD. The contour line depicts a quadratic function. Arrow in (b) at each step indicate the size and direction a standard SGD would take [7].

to perform parameter update after each training example. This is called **stochastic gradient descent** (SGD). This modification causes gradient to "jump around" in parameter space since every example has direct influence on the size and direction of the gradient. We can mitigate this by computing an unbiased estimate of the gradient, by taking the average gradient on a minibatch of m examples randomly drawn from the data distribution. This is called **minibatch stochastic gradient descent** and is most used optimization algorithm in today's neural networks. The m becomes a hyper-parameter of neural network, fewer examples cause gradient to "jump around" more. With an increasing number of training examples in a minibatch this effect is weakened but each step takes longer to evaluate. In this stochastic setting choosing a good *learning rate* is crucial, since SGD introduces a source of noise (random sampling) that does not vanish even when we arrive at the local optima. Therefore it is common in practice with each iteration to decrease a learning rate.

The gradient from a minibatch is a good approximation of the gradient of the full objective. Therefore, much faster convergence can be achieved in practice by evaluating the minibatch gradients to perform more frequent parameter updates. Often times the *minibatch* prefix is omitted in practice and is referred to *minibatch stochastic gradient descent* as simply *stochastic gradient descent* or *SGD*. In next subsections we will look at some improvements on this algorithm that help even more with faster convergence and problems often encountered in gradient descent optimization, such as saddle points or flat surfaces.

3.4.2 Momentum

Momentum is a modification of standard SGD algorithm that almost always enjoys better convergence. The method of momentum [28] was designed to help learning in high curvature space, or small but static or noisy gradients. The momentum gradient descent accumulates gradients and moves along in their direction, as shown in Figure 3.7b. This can be motivated from physics point of view. Where loss function represents a hilly terrain. A random initialization can then be seen as placing a particle somewhere in this terrain with zero velocity and optimization process as rolling this particle down the hill. The force felt by the particle is precisely the (negative) gradient of the loss function. More formally, $F = ma$, where m is mass of the particle in our case assumed to be 1 and a is acceleration of the particle. This view then suggests that the movement of the particle is only influenced by the velocity. Therefore, momentum algorithm does not update parameters directly. Rather it introduces a variable v which is a decaying sum of previous values and is also modified by current step in the space

$$v_t = \mu v_{t-1} - \alpha \nabla_\theta J(\theta)$$

where α is learning rate, θ parameter vector, $\nabla_\theta J(\theta)$ is the gradient of the cost function w.r.t. the parameters θ and μ is hyper-parameter specifying the amount of exponential decay. Then parameters are influenced by velocity vector $\theta = \theta + v$. This is different from standard SGD in a way that now the size of the steps depends on the size and *alignment* of the gradients. The step is going to be largest where many consecutive gradients will point in the same direction. The μ parameter can be interpreted as friction slowing down the particle. Otherwise the particle would never come to stop.

Nesterov momentum is a slight modification on momentum update [28]. It is based on *Nesterov accelerated gradient* method [29, 30]. It has stronger theoretical convergence guarantees for convex functions and also often times works slightly better in practice than basic momentum. The main difference is in the point where a gradient is evaluated. In Nesterov momentum, gradient is evaluated after current velocity is applied. This is similar as adding a correction factor to standard momentum update. Another way to look at this update is that in standard momentum update, we know that momentum term is going to be nudged by μv. So we can treat $\theta + \mu v$ as "look-ahead". We are surely going to end up in vicinity of this point therefore it makes sense to evaluate gradient at this "look-ahead" location (Figure 3.8). The full update is then

$$v_t = \gamma v_{t-1} + \alpha \nabla_\theta J(\theta - \mu v_{t-1})$$

$$\theta = \theta - v_t$$

where α is the learning rate, θ is the parameter vector, $\nabla_\theta J(\theta - \mu v_{t-1})$ is the "look-ahead gradient of the cost function w.r.t. the parameters θ and γ is the amount of decay applied.

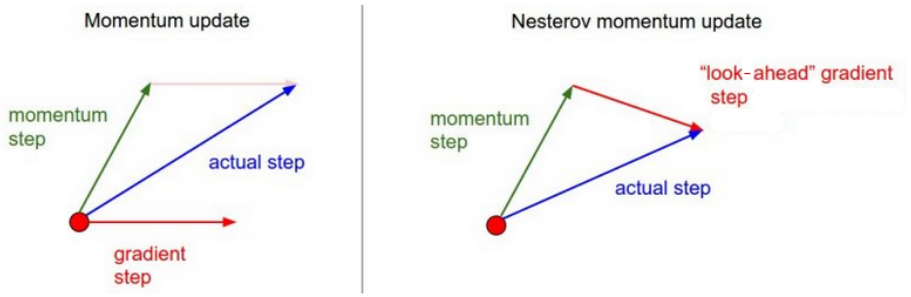


Figure 3.8: Momentum and Nesterov momentum update [8].

3.4.3 Adagrad

All previously discussed methods modified learning rate globally and with the same value for each parameter. There is quite a lot of work in current literature dedicated to per parameter update methods. Many of these methods might require more hyper-parameter optimization but they are well behaved for a broader range of hyper-parameter values than the raw learning rate.

One of these optimization methods is an adaptive learning rate method - **Adagrad** [31]. It works by adaptively scaling the learning rates of all model parameters. This scaling is done inversely proportional to the square root of the sum of all the historical squared values of the gradient.

$$\theta = \theta - \frac{\mu}{\sqrt{\mathbf{G}} - \epsilon} \odot \nabla_\theta J(\theta)$$

where $\mathbf{G}$ is diagonal matrix containing squared values of previous gradients, ϵ is smoothing factor and $\odot$ is a matrix-vector dot product. The parameters with larger partial derivatives have higher decrease in their learning rate, while parameters with small partial derivatives of the loss have a relatively small decrease. This produces effect on the network where greater progress is made in the more gently sloped directions of the parameter space. Adagrad enjoys nice theoretical guaranties. A downside of Adagrad is that in case of deep learning, the monotonic learning rate usually proves too aggressive and stops learning too early.

3.4.4 RMSProp

Modification of previously mentioned *Adagrad* is **RMSProp** [32]. It works by slightly modifying gradient accumulation and performs better in non-convex settings. Adagrad is designed to converge rapidly in convex settings. In training neural networks the parameter space is a highly non-convex surface. It might pass various structures during training and then at the end arrive to a region that is locally a convex bowl. It scales learning rate based on entire history of the squared gradient and therefore, might shrink

the learning too much before arriving at such a local convex region. RMSProp modifies this accumulation of the gradients into an exponentially weighted moving average.

$$\mathbf{G} = \gamma\mathbf{G} + (1-\gamma)\nabla_\theta J(\theta)^2$$
$$\theta = \theta - \frac{\alpha}{\sqrt{\mathbf{G}} - \epsilon} \odot \nabla_\theta J(\theta)$$

where γ is a decay rate and usually set to $\gamma = 0.9$ or $\gamma = 0.999$ and $\mathbf{G}$ is a diagonal matrix of previous gradients. This discards some of its extreme past and can converge more rapidly when in local convex bowl region. This convergence is similar to Adagrad if we initialize it somewhere in this convex bowl region. "Empirically, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks. It is currently one of the go-to optimization methods" [7].

3.4.5 Adam

In current deep neural networks the probably most used and also one of the go-to optimization method is "*adaptive movements*" algorithm called **Adam** [33]. It can be perhaps best summarized as a combination of previously mentioned *RMSProp* and *Momentum* with few important distinctions. Momentum is directly incorporated to Adam as first order moment with exponential weighting of the gradients.

$$m_t = \beta_1 m_{t-1}(1-\beta_1)\nabla_\theta J(\theta)$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla_\theta J(\theta)^2$$

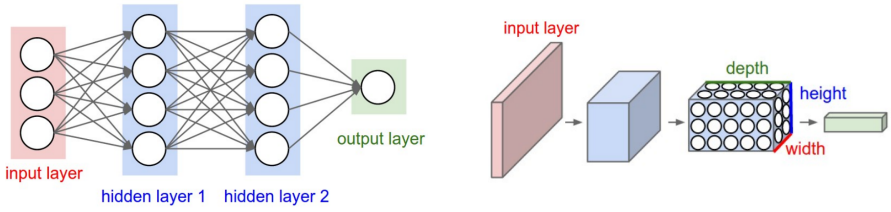
where β_1 and β_2 are decay rates which authors suggests are set to $\beta_1 = 0.9$, $\beta_2 = 0.999$ Also, Adam incorporates a bias correction of both first order moments (the movement term) and and the (uncentered) second-order moments to account for their initialization at the origin.

$$\hat{m}_t = \frac{1}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{1}{1-\beta_2^t}$$

This compensates for the fact that in the first few time steps the movement vectors are both initialized and therefore biased at zero, before they can fully "warm up". Adam is regraded to by fairly robust in terms of hyper-parameters although some slight modification to learning rate are sometimes necessary. The parameter update is as follows

$$\theta = \theta - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

where ϵ is smoothing factor that avoids division by zero and is usually $\epsilon = 10^{-8}$. As mentioned earlier Adam with its default parameters is usually go-to choice when testing deep neural networks. Note, that RMSProp has been shown to work slightly better in some cases with use of recurrent neural networks.



(a) (b)

Figure 3.9: Comparison between standard 3 layer neural network (a) and convolutional neural network (b) [8]. Note, that how hidden layers and output of CNN is a volume.

3.5 Convolutional Neural Networks

Convolutional neural networks are very similar to ordinary neural networks. They are made up of neurons that have learnable weights. Unlike normal neural networks, convolutional neural networks make the explicit assumption that the input is a signal in any form, for instance, images, speech, music, etc. This assumption that input is a signal allows us to encode certain properties into the architecture of the network.

In the high dimensional setting, such as images, the full connectivity of the regular neural network is wasteful. Also huge amount of parameters would quickly lead to overfitting. Therefore, convolutional neural networks do not connect neurons in such a fully-connected matter. The neurons in one layer are connected only to a small region of neurons the layer before it. Another main difference between normal neural and convolutional network is that later has its neurons arranged in 3D volume (Figure 3.9). A simple basic convolutional network also consists of sequences of layers. In a typical convolutional network there are three types of layers: convolutional, pooling and fully-connected.

3.5.1 Convolutional layer

Convolutional layers are the main building block of convolutional networks as the layer does most of the computations. The convolutional layer's parameters consist of a set of learnable filters. Every filter is small spatially, but extends through the full depth of the input volume. For instance, first filters in the network processing images, might have dimension of $5 \times 5 \times 3$ as they extend through 3 channels of RGB images. During the forward pass of the network, we convolve each filter (also sometimes called *kernel*) with the input volume and compute dot products between the entries of the filter and the input at all spatial location. The output is then a 2-dimensional activation map that gives the responses of that filter at every position. More formally if we assume

two dimensional input image I the discrete convolution can be defined as

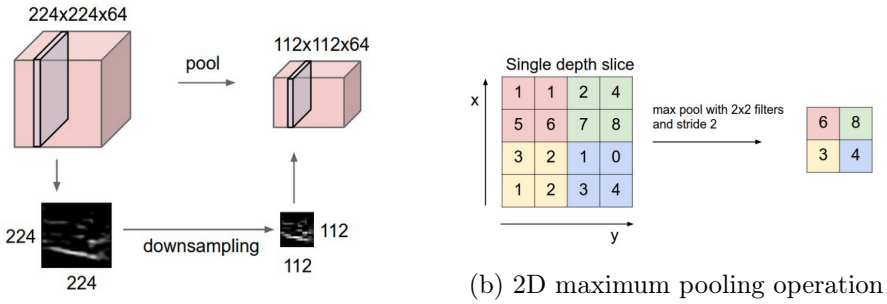
$$S(i,j) = (K \star I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

where S is output feature map, K is two dimensional kernel and I is two dimensional input. Intuitively, this map shows how much a filter is similar or in other words "likes" every position in the input. The network then learns this filters through the process of backpropagation so they extract important features for given task. For example, at the first layers, networks learn to recognize simple shapes such as edges. These simple features are in deeper layers combined to create more complicated shapes, such as honeycombs or wheel like blobs of color. Deeper the layer, more complicated filters it learns. These filters then produce high activations at similar locations in the input. Each filter produces such a activation map. We then stack these activation maps along the depth dimension to produce the output volume.

We can look at the output volume as an output of the neuron that looks only at a small region of the input and shares its weights spatially. It would be very impractical to connect all neurons to each other, specially when dealing with high-dimensional objects like images. The neurons in convolutional networks are connected to only local regions in the input. The region the neuron is connected to is called a **receptive field** and it is a hyper-parameter of the filter. The size of the output volume produced by convolutional layer is also based on other hyper-parameters, such as number of filters, size of filters, stride with which we are applying these filters and padding of the input.

3.5.2 Pooling layer

Pooling layer are usually inserted in-between successive convolutional layers. Its function is to progressively reduce the spatial size of the representation and to reduce the amount of parameters and computations in the network. It operates independently on every activation map of the input and reducing each map spatially using, most commonly, **max** operation [34]. It works in a similar way to the convolution layer. Sliding window is applied with arbitrary chosen stride to spatial location of the input computing max operation over its elements. This also helps to control overfitting of the network. As the convolutional filter slides through the input it transfer information to the next layer about every spatial location of that given input. Pooling layers reduce this information only to its most important part and therefore help the network to focus only on the important features of the signal rather then spatial locations of the filter. In other words pooling helps to make representations approximately invariant to small translations of the input. "*Invariance to local translation can be a useful property if we care more about whether some feature is present than exactly where it is*" [7]. Example of pooling layer is shown in Figure 3.10.



(a) Example of maximum pooling layer

(b) 2D maximum pooling operation

Figure 3.10: Pooling layer with example of corresponding max pooling operation [8].

Most common setup is a pooling layer with window of size 2 and stride of size 2 or 3 also called *overlap pooling* [9]. This setup achieves good performance while not discarding many features of the image. Larger receptive fields are too destructive. It is also worth noting there are other kinds of pooling operations, such as, **average pooling** or **L2-norm pooling**. There are also methods that learn pooling by clustering interesting features [35]. There was some work done analyzing different pooling methods and problems they are suitable for [36]. Historically, *average pooling* was often used, but recently the *max pooling* became more popular.

3.5.3 Fully-connected layer

Fully-connected layer is the same layer as in regular neural networks. Neurons in a fully connected layer have full pairwise connections to all activations in the previous layer, as seen in regular neural networks. Last 3D output volume of pooling or convolutional layer is *resized* to a long one dimensional vector and is treated as normal fully connected layer.

Note, that the only difference is that the convolutional layers are connected only to local region in the input and neurons share parameters. Functional form of fully-connected layers and convolutional layers is identical as they both compute the dot product over the input. It is therefore possible to convert fully-connected layers to convolutional layers and vice versa. The second conversion from fully-connected to convolutional layers is often used in practice. To convert between these layers the filter size of convolutional layer is set to full spatial dimension of the input and setting the number of filters to be equal to the number of neurons originally contained in fully-connected layer. This conversion allows us to slide the whole network over the bigger input very effectively. This is most commonly used to get better performance by upscaling the image and evaluating the network at many different spatial locations of bigger image in a single forward pass and averaging the score.

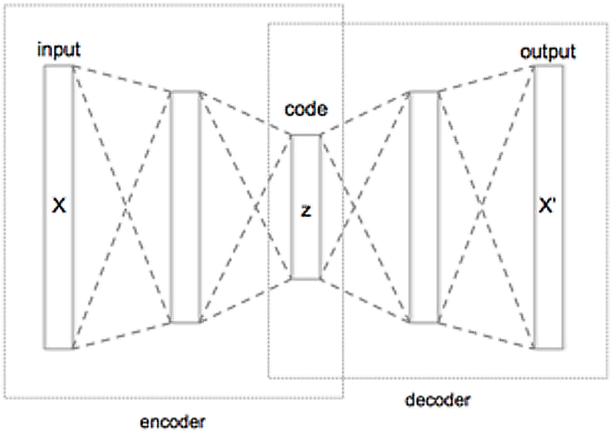


Figure 3.11: An example of autoencoder structure.

3.6 Autoencoders

In this section we will discuss models called **autoencoders**. Autoencoder is a neural network that is trained to reproduce its input as output. More precisely, autoencoder has a hidden layer h that represents or codes its input. Autoencoders consist of two parts. First, **encoder** that transforms input of the network to some hidden representation. Second, a **decoder** that tries to reconstruct input from this encoded representation, as shown in Figure 3.11. More formally, encoder learns a function $h = f(x)$ and decoder that produces a reconstruction $r = g(h)$. The obvious simple solution is to learn an identity function $g(f(x)) = x$. This is not particularly useful and therefore encoders are usually restricted in some way not to copy the input but to reconstruct an approximation of the input. These restrictions force the model to prioritized which features of the input should be reconstructed. Model is then often forced to learn useful properties of the data. The autoencoders are not a new idea and been part of active research for quite some times [37, 38]. Historically, autoencoders where used for dimensionality reduction or feature learning. In recent years a connection between autoencoders and latent variable modeling pushed these models to forefront of generative modeling. Autoencoders can be viewed as feed-forward networks and therefore trained with all the same techniques, such as minibatch gradient descent followed by gradients computed by backpropagation algorithm. But, there is also another way that autoencoders can be trained. This method is called *recirculation* [39] and is biologically more plausible than backpropagation. *Recirculation* optimizes the network by comparing activation on original input with activation on the reconstructed input.

One way to restrict the autoencoder not to learn an identity function is to pose a restriction on h to have smaller dimensions then input x. Such a autoencoder is

called **undercomplete** or *CAE*. Note, that CEA shorthand is also used in literature to denote a Convolutional Autoencoder, which uses convolution layers and is often used to reconstruct images. Learning such a *undercomplete* representation forces the autoencoder to learn only most silent features of the input. This learning process is described by simple optimization of the loss function based on difference between reconstructed output $g(f(x))$ being dissimilar from input x.

$$L(x, g(f(x)))$$

where loss function L can be, for instance, *Mean Squared Error* (MSE). Unfortunately, if the encoder and decoder are allowed too much capacity they fail to extract anything useful and rather learn to copy the image even if we pose very strict restrictions on h.

Similar problem occurs also if the dimensions of hidden representation h is allowed to be equal or in case of CAE greater then input dimension. In these cases even a linear model can learn just simply copy the input without learning anything useful about data distribution. We should be able to train any autoencoder and choose its capacity and hidden representation based on complexity of data distribution. One way how to achieve this are **Regularized Autoencoders**.

3.6.1 Regularized Autoencoders

Regularized Autoencoders do not limit model capacity by keeping it shallow or by position restriction on dimensionality of h. Rather, they use specific loss function to force the model to have different properties, other than coping input. Such properties might include a sparsity of the representation, small derivative of the representation or robustness to noise or missing inputs. Regularized autoencoders can be deep, nonlinear and undercomplete and still be capable of learning something useful about data distribution even when their capacity is large enough to learn trivial identity function. It is worth noting that any generative model with latent variables and inference procedure can be viewed as some form of autoencoder, for instance, *Variational Autoencoders* (VAE) [40] or *Generative Stochastic Networks* [41].

One class of these regularized models are **sparse encoders**. These encoders force a sparsity property on hidden representation h by simply adding a sparsity penalty $\Omega(h)$ to standard reconstruction error

$$L(x, g(f(x))) + \Omega(h)$$

where $g(h)$ is output of the decoder and $h = f(x)$ and its output of the encoder. $\Omega(h)$ can be represented as a sum of absolute value penalty

$$\Omega(h) = \lambda \sum_i |h_i|$$

where λ is treated as hyper-parameter.

Other model that utilizes regularization is **Denoising Autoencoder** (DAE) [42]. The denoising autoencoder is an autoencoder that receives a corrupted data point as input and is trained to predict the original, uncorrupted data point as its output. This model does not add a regularization term $\Omega(h)$ but rather modifies a optimization of the loss function. In standard autoencoder we use

$$L(x, g(f(x)))$$

loss function which optimizes a reconstruction error. Denoising autoencoder modifies this to compute

$$L(x, g(f(x')))$$

where x' is a modified input x with random noise. This forces model to first repair the damaged done by adding noise to the input. This shows that useful features can be learned as a byproduct of optimizing reconstruction error. Methods described in this section are only some basic regularization techniques. More regularization methods, such as *Regularizing by Penalizing Derivatives*, can be found in [7].

Chapter 4

Dataset

In this section we will analyze the dataset used in this work and describe the sampling technique to create training samples. Our models were trained and evaluated on the dataset proposed in [5][3].

This dataset contains annotated paths of 12684 pedestrians from when they entered the scene to the time they left the scene and were no longer visible. The dataset is composed of two parts. First, a set of 6001 full-HD RGB frames of surveillance video taken from public camera and saved every 20 frames (0.8 seconds). Concatenated, this dataset is 4800.8 seconds in length. Second part is a set of files for each pedestrian containing mapping to each frame he/she is visible in with exact coordinates in image space.

4.1 Error analyses

The dataset contains labeling errors and therefore needed to be cleaned for further use. Not all frames have assigned pedestrian files with coordinates annotations for pedestrians visible in the scene. This dataset contains 260 unlabeled images. These

Figure 4.1: Sequence of 5 unlabeled images from the dataset.

[3]This dataset can be found at `http://www.ee.cuhk.edu.hk/~syi/`

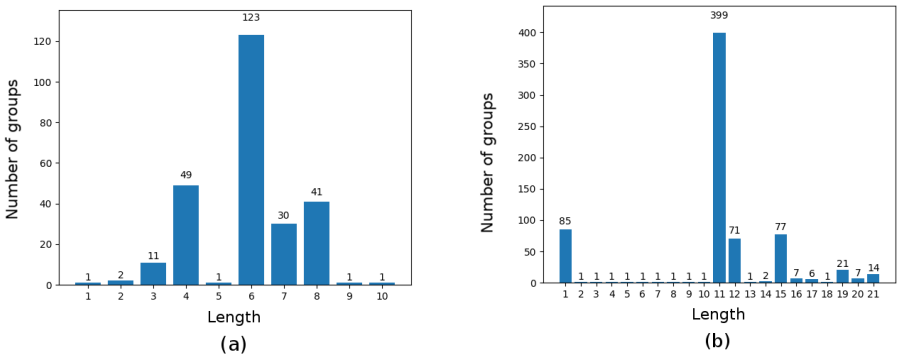


Figure 4.2: (a) Lengths histogram of missing sequences in the data. (b) Lengths histogram of sequences containing badly annotated frames in the data

images represent 10 different sequences of consecutive frames with the largest sequence spanning across 123 frames (98.4 seconds), as can be seen in Figure 4.2. Unlabeled frames are not empty in terms of people moving through the scene and in some cases contain many pedestrians, as shown in Figure 4.1. Other frames are not completely annotated and contain errors in labeling. Error frames have annotations for only a fraction of pedestrian actually visible in the scene and can be identified by setting a threshold on a number of annotations associated with individual frames. We consider every image that has a low number of labels to be marked as "badly annotated" and removed from our set. Figure 4.3 shows relationship between the number of badly annotated images and increasing pedestrian threshold. As we can see from the graph (Figure 4.3) the relationship has a logarithmic nature until it reaches the threshold value of 40. This value represents 1007 badly labeled images. After this value, graph shows linear relation of badly annotated files in respect to increasing threshold value. This suggests that after this threshold the images have more or less correct annotations and may contain less people moving through the scene. Furthermore, average number of labels per image is 79 with a maximum number of labels being 289 on a single frame.

The number of sequences (consecutive frames) these badly annotated images create grows exponentially in respect to the threshold value (Figure 4.4). Most of these sequences contain only one frame, as shown in Figure 4.4, which if removed would introduce many small one frame holes into our dataset, even with sufficiently small threshold. For instance, threshold value set to 30 removes 770 images and creates 138 holes from which 122 are only single frame long. These single frame holes might introduce "time skips" to our models. This property is undesirable because images are fed to our models in sequences. This is because our models are using 3D convolution to also capture the time dimension of the data. Therefore, one frame holes would represent impossible jumps of individual pedestrians. These jumps might be explained

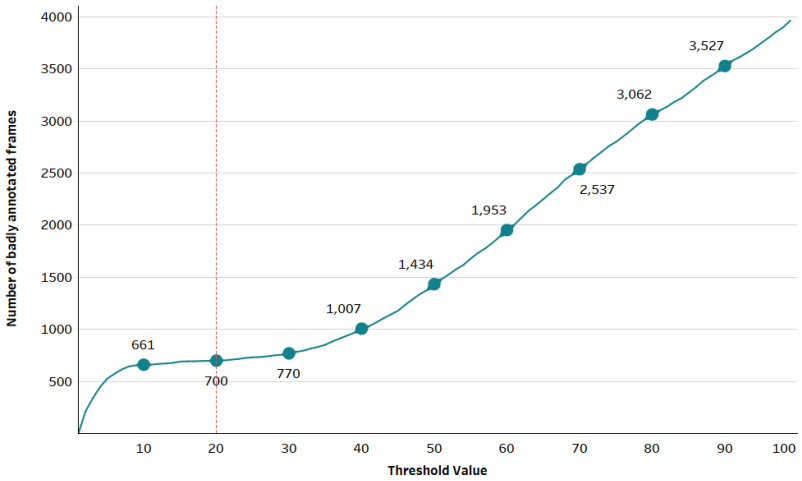


Figure 4.3: Total number of badly annotated frames in relation to increasing pedestrian threshold.

only by every short and very small 1.6 second runs and might hinder the models ability to extract time based information from sequences of frames.

It could be argued that these holes create somewhat adversarial environment. The models should learn to ignore or fix these holes and in the end this type of errors should make the models more robust. This might be true for larger models and bigger datasets but due to the fact that our dataset is not particularly large in terms of shear number of images, with higher threshold value, we would remove sizable chunk of almost correctly annotated images. Also, our models should be reasonably small so they can be actually used in production and therefore might be unable to deal with these types of errors. Based on these fact we set the pedestrian threshold to 20 labels. This is a good trade-off between removing enough badly annotated images that have almost no annotations and not creating many holes in the data. Each image that contains less than 20 labels is considered to be badly annotated and therefore removed. This threshold value removes 700 frames which are represented in 21 missing sequences with the largest one being 399 frames long (319.2 seconds), as shown in Figure 4.2. Rendering the total size of our dataset to 5041 images.

4.2 Sampling

As mentioned in the previous section, our dataset is not large in terms of shear image count. Hence, we artificially expanded the dataset with technique called data augmentation [43]. One of the most common data augmentation is flipping the image horizontally. This creates new training samples as well as helps the models to preserve spatial invariant property. This should help the model to extract relevant information from the frames rather than focusing specific part of the image. For instance, in our

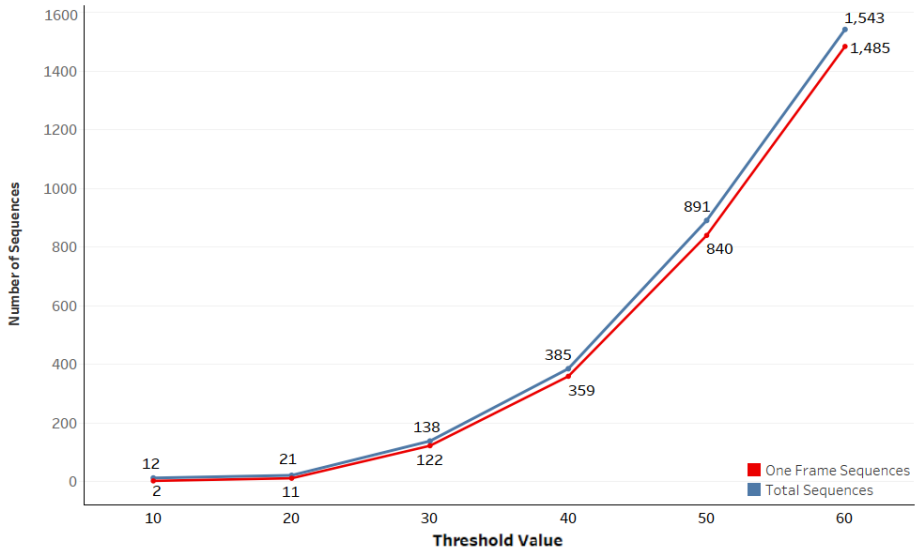


Figure 4.4: Number of sequences created by badly annotated files in relation to increasing pedestrian threshold.

case, networks should learn to identify where the individual pedestrians are on the image rather than focusing on the fact that there are never any pedestrians in top left corner.

As mentioned in Section 3.6 we are trying to predict a mask of future coordinate points where the pedestrians will be in the next time step of the video. The main problem with this approach is the sparsity of the output. Most of the pixels in the output mask are zeros. The consequence of this fact is that if we would try to predict entire mask, our weights of our models would effectively become zero. Since, for most of the sensible loss functions, predicting mask of all zeros would yield a small enough error and therefore models would stop learning after few epochs. This is due to the fact that the zero pixel predicting model would be in most of the cases correct and miss only few pixels representing pedestrians resulting in a small error rate. One way how to make the output less sparse is to enlarge the representation of the pedestrians on the mask and thus resulting in less zeros in the output. This approach might lead to unreasonably big representations and merging of pedestrians representation to big blobs. We tested various sizes of pedestrian representations. In Figure 4.5 we can see the ratio between all zero samples and samples containing parts of pedestrian in relation to increasing pedestrian representation. Results, which these representations have on performance of the model are proposed in Section 6.2. Based on this ratio, shown in Figure 4.5 we also assigned a less importance to samples that contain no pedestrians.

The other approach used in [44, 45] is to split the input and output to smaller chunks. First, we sample M consecutive images and stack them together to create

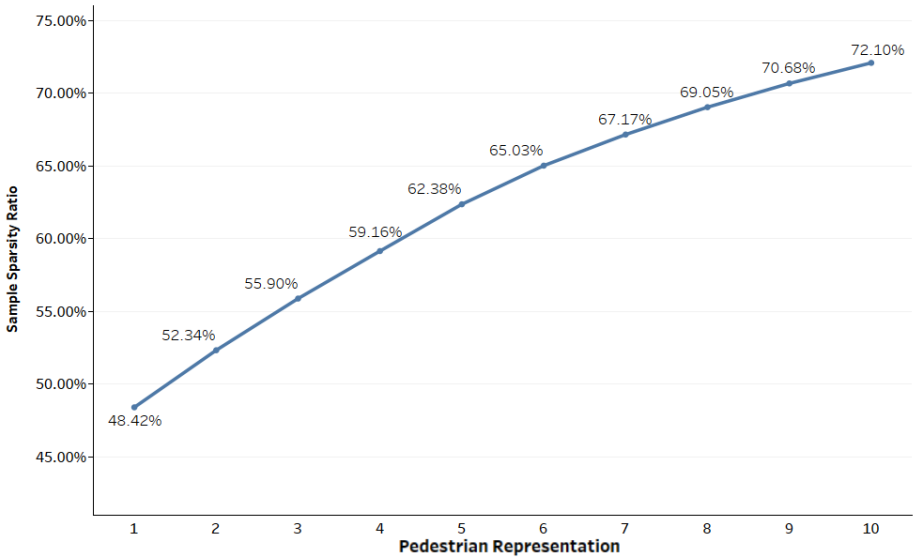


Figure 4.5: Ratio between all zero samples and samples containing at least on pixel representing pedestrian in relation to increasing size of pedestrian representation.

volume $\mathcal{D}$ of size $M \times H \times W \times 3$, where H and W are height and width of a image. Next, we split this volume into smaller equal patches each of size $M \times H_p \times W_p \times 3$, where H_p and W_p are width and height of a patch. The size of each patch was chosen arbitrary to fit the size of the image. The patches were created by sliding a window through our volume $\mathcal{D}$ and cutting out the overlapping part, as shown in Figure 4.6. Sliding window was also parametrized by striding value so we can control how much are individual patches overlapping.

There are two main approaches we are going to use to construct the final output volume from model prediction. First, our model will predict entire probability mask for each input patch. A thresholding operation will be applied on these prediction so we get $0/1$ mask (Figure 4.7). Then, we merged the patches together. There are multiple merging schemes we can use to put together the overlapping parts of the volume. For instance, if one pixels is in three different patches, the final value will be determined based on the majority value of this pixels in these overlapping patches. Second approach is to classify each pixel of the output volume based on the input patch (Figure 4.7). This approach is also used in [44]. The data for this model can be created by setting the stride of our cutting window to 1. If the we keep the spatial dimension of the input we would downscale the output volume. This is because there is no space for window to be centered on the edge pixels. For this reason we also have to pad the volume (each frame) with zeros.

This process creates a set of smaller volumes of size $N \times M \times H_p \times W_p \times 3$, where H_p and W_p are width and height of a patch and N is the number of patches that fit

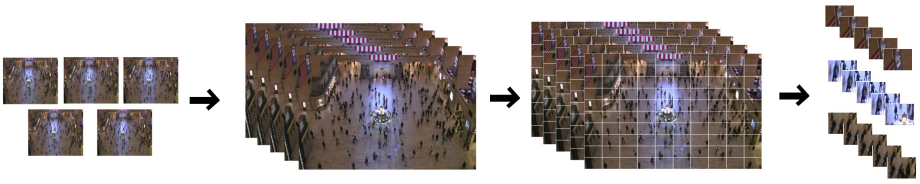


Figure 4.6: Process of segmenting input into smaller patches.

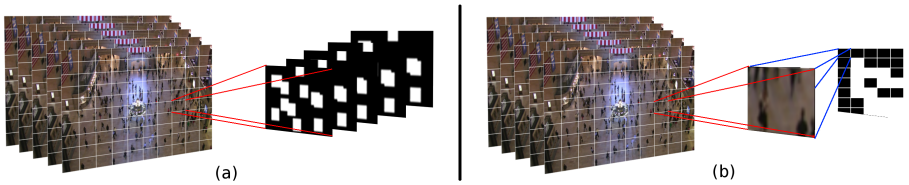


Figure 4.7: Illustration of our approaches. (a) Illustration of predicting entire volume at once. (b) Illustration of classifying task where we classify each pixel.

into the volume and can tbe calculated as follows

$$N = \left(\frac{H - H_p + 2 \times P_h}{S_h} + 1\right) \times \left(\frac{W - W_p + 2 \times P_w}{S_w} + 1\right)$$

where H, W are volume height and width respectively. P_w and P_h is the amount of padding applied in each direction. S_h, S_w are strides the cutting window is sliding in respective directions. We evaluated our models for different patch sizes. Results can be found in Section 6.2. The true labels are constructed in similar way. First, for each image in the input we create equally sized output mask where all elements are set to zero. Based on the provided annotations we mark each pedestrian with a small matrix of ones of various sizes based on experiments as mentioned before. These output masks are stacked to create a output volume $\mathcal{D}'$ of size $M' \times H \times W$, where H and W are height and width of input images and M' is the number of time steps ahead we want to predict. Note, there is no 4^{th} dimension representing RGB channels, because all elements of our masks are ones and zeros representing pedestrians and everything else respectively. These volumes are also segmented into smaller patches. Each patch representing a mask of specific input patch, therefore having same spatial dimension $M \times H_p \times W_p$. For pixel classification models, the cutting step is removed and the true labels are constructed by taking each $H \times W$ vectors of size M from output volume $\mathcal{D}'$.

In current implementation we set $M = 5$. Each input volume contains five consecutive frames taken from video. As mentioned before frames are sampled every 0.8 second therefore allowing our networks to look at 4 seconds snapshots of the video. The larger the value of M more computations and bigger models are needed. With larger values of M the performance is also increasing due to the fact that the network has more information and is looking at longer snapshots of the video. Output is created

from annotated input and therefore also sampled at the same rate of 20 frames (0.8 seconds). For instance, for M' equal to 5 we predict pedestrian locations and their path for 5 consecutive frames which is 4 seconds. We tested multiple values of M' and evaluating models ability to predict various time frames ahead. Dataset was split to test and train set with ratio of 80:20 and test results can be found in Section 6.2. We used a library called *scikit-learn* [46] to implement some of the preprocessing steps and all data were stored in *NumPy* arrays [47] .

Chapter 5

Model architectures

In this section we will discuss the architecture of proposed models. There were two main approaches as described in Chapter 4. First, we tried to predict entire volume of pixels in the output mask. Second, we approached the problem as a binary classification task over individual pixels of the output mask and tried to classify if each pixel represents a pedestrian or not across all future frames. For the first approach we tested two models. First, a simple encoder where an input volume is first encoded into some representation and then through the deconvolution operation the output mask of size $M' \times H_p \times W_p$ is constructed where M' is the number of future frames predicted and W_p, H_p are width and height of extracted patches, respectively.

Second, we used a column stack encoder which is similar to previously mentioned simple encoder but its first layers are stacked in column wise fashion as in [3], each having different size of filters. These stacked layers are separate and do not share weights with each other. For the classification task we tested only column stack convolution neural network where output of the network is a probability that a pedestrian is located in this position over M' pixels. Input to both encoders and classifier was the same - volume of raw previous frames of size $M \times H_p \times W_p$ where M is the number of previously seen frames and W_p, H_p are width and height of extracted patches, respectively, as described in Chapter 4. For implementation of our models we used a deep learning framework *Keras* [48]. Each training of the network was monitored with early stopping script that had patience set to 3. This stops the training of the network when the validation loss increases during the training 3 times. We also used 10% of the training set for validation. Each model was optimized with *binary cross-entropy* but we also ran experiments with *mean square error* or *logcosh* losses. Each loss performed similarly and there were no significant differences in the performance of the models. Due to the sparsity of the output we also tested architectures were the model was regularized on the sparsity of the hidden representations as defined in Section 3.6. By

adding a sparsity penalty $\Omega(h)$ expressed as a sum of absolute value penalty

$$\Omega(h) = \lambda \sum_i |h_i|$$

where λ is a hyper-parameter and h is the encoded hidden representation. Another regularization technique that we tested was adding a penalty $\Omega(D, D')$ for miss-placed pixels, in other words, for false positive and negatives or pixels with low probability. Since we have only binary output for each pixel this can be done by simply subtracting the predicted output mask from true mask

$$\Omega(D, D') = \lambda \frac{1}{N} \sum_i \sum_j \sum_m^{M'} (D_{ij}^m - D_{ij}'^m)^2$$

where λ is a hyper-parameter adjusting a strength of the regularization, N is the number of pixels in output volume and D is true output volume representing pedestrian mask in each time frame M', D' is the predicted volume representing pedestrian mask in each time frame M'. Results for these regularized models are provided in Section 6.2.

5.1 Simple convolutional encoder

As mentioned before first model architecture we tested was simple encoder (Table 5.1) that gets the input volume of $M = 5$ previous frames of video split into small patches and tries to construct a pedestrian mask of size $M' \times H_p \times W_p$. From Table 5.1, showing the architecture of this model, we can see that the first convolutional layer with ReLU nonlinearity consists of 16 filters each of spatial size of 5×5 spanning across all M time frames. This layers is followed by dropout layer which drops 20% of units. We also tested networks without any dropout layers but we were able to train networks with dropout layers for more epochs and therefore these networks performed better. This layer is then followed by another 3D convolutional layer consisting of 32 filters each of spatial size of 3×3 spanning 5 time frames. This layer has more filters and each filter has a smaller spatial size, which is common in convolutional neural networks where with increasing depth the number of filter increases and spatial size of these filters decreases. This block of layers is ended by max pooling layer with filter size of $2 \times 3 \times 2$. Next, another block of convolutional layers with ReLU nonlinearities follows with number of filter 64 and 128. Each filter has spatial size of 3×3 and time dimension spanning across 3 time frames. This then produces encoded representation of the input from which pedestrian mask is constructed. This is done by running two sets of convolutions and upsampling layers. Each convolution layer consists of 64 filters of spatial size 3×3 and spanning across 3 and 5 time frames respectively. After final

Layer	Type	Shape
0	Input	$M \times H_p \times W_p$
1	Convolution + ReLU	16 filters, each $M \times 5 \times 5$
2	Dropout of 20%	
3	Convolution + ReLU	32 filters, each $5 \times 3 \times 3$
4	Max Pooling	$2 \times 3 \times 2$
5	Convolution + ReLU	64 filters, each $3 \times 3 \times 3$
6	Dropout of 20%	
7	Convolution + ReLU	128 filters, each $3 \times 3 \times 3$
8	Encoded Output	
9	Convolution + ReLU	64 filters, each $3 \times 3 \times 3$
10	Upsampling	$1 \times 3 \times 2$
11	Convolution + ReLU	64 filters, each $5 \times 3 \times 3$
12	Upsampling	$M' \times 3 \times 2$
13	Convolution	1 filter, each $5 \times 3 \times 3$
14	Sigmoid Nonlinearity	output of size $M' \times H_p \times W_p$

Table 5.1: Architecture of a simple convolutional encoder.

upsampling layer a final convolutional layer is applied to produce output of desired size.

5.2 Column stack convolutional encoder

The second model architecture we tested was based on architecture proposed in [3] where first layers of the network are separate and run in parallel each with different sizes of the filters. The weights in these layers are not shared. This should help the network with perception of the scene. Since many of the scenes shown in videos from public surveillance cameras have in some form warped perspective where pedestrian in the bottom of the frame (closest to the camera) appear larger than pedestrians on the top of the frame (furthest from the camera). Column architecture of the models tries to address this discrepancy with filters of different spatial sizes. Larger filters can better detect larger pedestrians since their receptive field is larger. On the other hand, smaller filters are better at detecting pedestrians that appear smaller. Each layer then extracts these low-level features on different scales which should help the network to compensate for perspective of the scene.

From Table 5.2, showing the architecture of our model, we can see that our first column layer consists of 3 single convolutional layers each having 16 filters. For these first layers we used filter size of spatial size 7×7, 5×5 and 3×3 each spanning across all

Layer	Type	Shape
0	Input	$M \times H_p \times W_p$
1.1	Convolution + ReLU	16 filters, each $M \times 7 \times 7$
1.2	Convolution + ReLU	16 filters, each $M \times 5 \times 5$
1.3	Convolution + ReLU	16 filters, each $M \times 3 \times 3$
2	Concatenation	[layer 1.1, layer 1.2, layer 1.3]
3	Convolution + ReLU	32 filters, each $5 \times 3 \times 3$
4	Dropout of 20%	
5	Convolution + ReLU	32 filters, each $5 \times 3 \times 3$
6	Max Pooling	$2 \times 3 \times 2$
7	Convolution + ReLU	64 filters, each $3 \times 3 \times 3$
8	Dropout of 20%	
9	Convolution + ReLU	128 filters, each $3 \times 3 \times 3$
10	Encoded Output	
11	Convolution + ReLU	64 filters, each $3 \times 3 \times 3$
12	Upsampling	$1 \times 3 \times 2$
13	Convolution + ReLU	64 filters, each $5 \times 3 \times 3$
14	Upsampling	$M' \times 3 \times 2$
15	Convolution	1 filter, each $5 \times 3 \times 3$
16	Sigmoid Nonlinearity	output of size $M' \times H_p \times W_p$

Table 5.2: Architecture of column stack convolutional encoder.

depth of the given volume. The low-level features are then concatenated into one single representation followed by convolutional block with ReLU nonlinearities consisting of two convolutional layers of 32 filters of size $5 \times 3 \times 3$ and dropout layer with drop rate of 20%. We also tested different merging techniques, such as addition but this did not provide any significant performance improvement, therefore we decided for the simplest merging method. First column of layers consists of only one convolutional layer per row extracting only low-level features from the images. We also tested a model architecture with additional convolutional layers in each row. These layers where accompanied by a dropout layer with drop rate of 50% which should extract more mid-level features. This model neither showed any significant performance improvement and therefore we again decided to keep the simpler model. Merging layer, is followed with another block of convolutions with 64 and 128 filters each of size $3 \times 3 \times 3$. This produced encoded output from which a pedestrian mask was constructed by running the same convolutional and upsampling scheme as in simple encoder (Section 5.1). We tested this model architecture without any dropout layers but again we were able to train models with dropout layers for longer without any significant overfitting. We also tested an ensemble version of this model where instead of predicting entire output volume representing pedestrian mask for each of M' future time frames. We trained M' column convolutional encoders each predicting only one mask of size $1 \times H_p \times W_p$ for each future time frame. The first model predicted a mask of the first frame ahead (0.8 seconds), the second model predicted a mask of the second frame ahead (1.6 seconds) and so on.

5.3 Column stack convolutional classifier

Last but not least, the model architecture we tested was similar to that of column stack convolutional encoder. However this model architecture was optimized for binary classification objective. We tried to classify each pixel of the output mask along entire depth (time dimension) of the output volume, as described in Section 4. The model consists of the same architecture as column stack convolution encoder (Section 5.2) except after the encoding part we flatten the output of last convolution and two fully connected layers follow, as shown in Table 5.3, the first one having 128 neurons and the second one having M' number of neurons. In other words, we are trying to predict the probability of the pedestrian currently being located on this particular pixel at each time frame. We choose this architecture as it was shown before [44, 45] that similar classification models perform well on prediction of image masks. Similarly to column stack convolutional encoder, described in Section 5.2, we also tested a ensemble version of this model where rather than predicting M' pixels at the output layer we had M'

Layer	Type	Shape
0	Input	$M \times H_p \times W_p$
1.1	Convolution + ReLU	16 filters, each $M \times 7 \times 7$
1.2	Convolution + ReLU	16 filters, each $M \times 5 \times 5$
1.3	Convolution + ReLU	16 filters, each $M \times 3 \times 3$
2	Concatenation	[layer 1.1, layer 1.2, layer 1.3]
3	Convolution + ReLU	32 filters, each $5 \times 3 \times 3$
4	Dropout of 20%	
5	Convolution + ReLU	32 filters, each $5 \times 3 \times 3$
6	Max Pooling	$2 \times 3 \times 2$
7	Convolution + ReLU	64 filters, each $3 \times 3 \times 3$
8	Dropout of 20%	
9	Convolution + ReLU	128 filters, each $3 \times 3 \times 3$
10	Encoded Output	Flatten encoded output
11	Dense	128 neurons
15	Dense	M' neurons
16	Sigmoid Nonlinearity	output of size M'

Table 5.3: Architecture of column stack convolutional classifier.

models each predicting pixels for different time frame. Results for both single model and ensemble are reported in Section 6.2.

Chapter 6

Results

In this chapter we provide and analyze results for proposed models in Chapter 5 as well as provide a more detailed error analysis of the proposed approaches. Mean squared error (MSE) is adopted as evaluation metrics, similar to [5] to make results comparable. The [5] defines a mean square error as follows

$$MSE = \frac{1}{NM'} \sum_{i}^{N} \sum_{j}^{M'} ||I_i^j - I_i'^j||_2 \times 100\%$$

where N is the number of samples, M' is the number of predicted frames, I is the volume containing normalized annotated positions of each pedestrian and I' is predicted volume of normalized pedestrian locations. Since we are predicting a mask of pedestrians in future time frames and not exact locations, a postprocessing step is needed to extract pedestrian locations and construct prediction volume I'. In order to extract coordinates of each pedestrian, in the image, we thresholded a predicted probability mask with a binary threshold function together with *Otsu's Binarization* [49]. Next, we searched for contours in the prediction mask. Each found contour (patch of pixels) represents one pedestrian. From each contour we kept top and left most coordinates to represent a location of pedestrians. This process is illustrated in Figure 6.1. We implemented these steps with help of a computer vision library called *OpenCV* [50]. We performed various experiments for different cutting windows sizes in data preparation step, pedestrian representations, as mentioned in Chapter 4, and also performed a grid search for optimal hyper-parameters of the each network. In this section we present

Figure 6.1: Posprocessing step needed to extract exact pedestrian coordinates.

	9x8	18x16	9x16	18x8
simple encoder	**12.446**	14.247	14.997	13.879
column encoder	**10.348**	11.152	12.432	12.795
column classifier	**10.732**	12.678	12.485	11.485

Table 6.1: Average MSE of different window sizes in preprocessing step for each model predicting 1, 3 and 5 frames ahead.

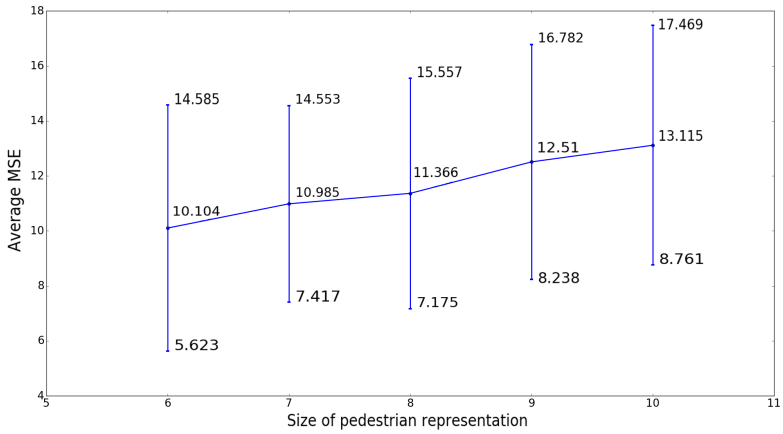


Figure 6.2: Average MSE for different sizes of pedestrian patch representations.

only the most relevant results. Lastly, we present sample predictions of our best model.

6.1 Preprocessing

In this section we provide results for hyper-parameters chosen in our preprocessing steps - the size of a cutting window and the size of pedestrian representation.

First, we needed to find the best size of cutting window in a preprocessing step. We performed grid search through various combinations of windows on various models trained with different hyper-parameters. We report average MSE for models that were predicting 1, 3 and 5 frames ahead. As we can see from Table 6.1, window size 9×8 performed the best. This fact is surprising since we would expect that large windows would provide more information for the network and therefore should perform better. But this was not the case in our setting and smaller windows generally performed better than large ones. This could be caused by sparsity of the output, because we are predicting a pedestrian mask of the same size as input patch.

Due to the sparsity of the output we also tested different sizes of the pedestrian patch representations. From values in Figure 6.2 we can see that we achieve best performance of the networks with representation of size 6×6. We were unable to train networks with smaller representations, since the sparsity of the output was too

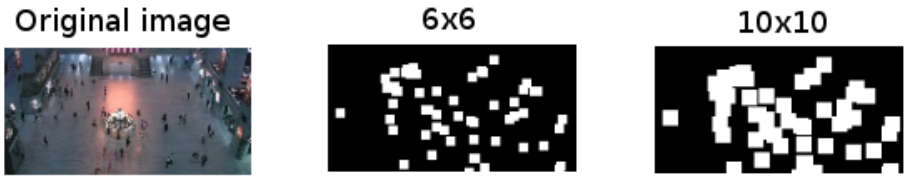


Figure 6.3: Different size of individual pedestrian representation with original image.

	1 frame	2 frame	3 frame	4 frame	5 frame
simple encoder	5.805	13.112	13.899	14.010	15.472
column encoder	3.946	11.766	12.101	12.002	12.621
simple encoder + sparsity	4.900	13.761	14.189	14.510	15.706
column encoder + sparsity	3.401	11.102	**11.336**	11.919	12.803
simple encoder + penalization	4.605	12.779	13.860	14.217	15.873
column encoder + penalization	4.070	11.690	12.093	12.370	12.110
column classifier	3.245	11.042	11.711	**11.893**	12.829
column encoder ensemble	3.103	11.983	12.622	12.891	13.209
column classifier ensemble	**3.015**	**10.814**	*11.519*	*11.978*	**12.780**

Table 6.2: Final MSE for various types of tested model architectures predicting 1 to 5 frames ahead.

large and networks in our experiments set all their weights to zero after first the few epochs and training stopped. Further analysis of this fact is provided in Section 6.3. As we can see from Figure 6.2 with increasing size of the pedestrian representation the average MSE is also increasing. This is due to the fact that in our dataset, or generally in similar public spaces, individual scenes are crowded and pedestrians are close to each other. Therefore with larger representations we often merge multiple pedestrians into one *big "blob"*, as shown in Figure 6.3. Similar blobs of pedestrians are then also predicted and we are unable to distinguish individual pedestrians present in these blobs and therefore unable to extract each pedestrian coordinates.

6.2 Model results

We tested various model architectures mentioned in Chapter 5 and in this section we provide basic analyses and results for these model architectures. As discussed in Section 6.1 we report results for models that were trained on the data that had cutting window of size set to 9×8 and patch representing each pedestrian of size 6×6.

From results shown in Table 6.2 we can see that **ensemble of column classifiers** achieved best results for prediction of 1, 2 and 5 frames ahead. For prediction of 3 and 4 frames ahead it achieved second best score where difference from best score is minimal.

	precision	recall	accuracy	F1 score
simple encoder	**75.579**	18.224	**77.215**	29.366
simple encoder + sparsity	75.153	18.557	77.147	29.764
simple encoder + penalization	74.918	**19.489**	76.809	**30.931**
column encoder	80.832	25.366	88.095	38.157
column encoder + sparsity	**81.051**	24.998	87.887	38.210
column encoder + penalization	80.601	**26.310**	**88.105**	**39.670**

Table 6.3: Precision, recall, accuracy and F1 score metrics reported for simple and column encoders. Note the minimal difference in reported metrics for models with no regularization and models with regularization.

We can see that single column classifier achieved best results from models that predict entire volume of frames and are not ensemble of individual networks. This is not a very surprising result, since in [44] and [45] similar classification models where used for solving similar problems of predicting a sparse mask of given image and show better performance than encoder models. This can be attributed to the nature of the problem where it is inherently more difficult to approach the problem from the regression stand point and predict how entire image or segment of the image will look like (predict exact value of each pixel) than a binary classification of individual pixels of the given image. We can also see that column encoder performs generally better than simple encoder. This is expected since column encoder model is larger in size and therefore has larger capacity and can extract more information. Also, by utilizing filters with receptive fields of different sizes, the features learned by each column are adaptive to variations in pedestrian body sizes due to perspective effect, as mentioned in Section 5.2.

Values in Table 6.3 also show that regularizing a sparsity of an encoded vector or penalization of false positive and negatives does not help models to make more accurate prediction which can be also seen in Table 6.2 where there is no significant improvement in error rate for models with regularization.

From Table 6.3 we can see that best precision achieved column encoder with sparsity regularization but the difference in values for other column encoders where minimal. This suggests that this type of regularization had almost no effect on performance of our models. This can be seen also from values for simple encoder that are shown in Table 6.3. Main reason for this fact can be the capacity of the models or large sparsity of the output and consequent inadequate pedestrian representation.

We can also see from Table 6.2 that the error increases roughly three times when predicting more than one frame ahead. This suggests that our models can predict one frame ahead (0.8 seconds) but are unable to extract enough positional information from raw frames to predict pedestrian coordinates more than one steps ahead. This

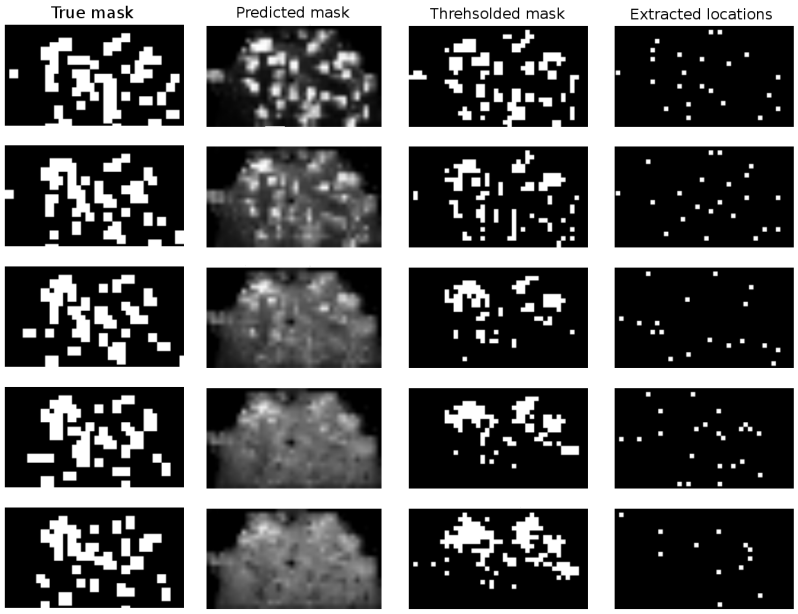


Figure 6.4: Sample prediction of our best model with extracted pedestrian locations. Each row represents one step ahead in prediction. Note, the huge cluster in later steps where models predict rough map of possible location of pedestrians.

can be attributed to many factors with probably the main one being that our models are too shallow and have not enough capacity to extract information about pedestrian positions in given frames and also predict future walking patterns of every pedestrian in the frame. This can also be seen in Figure 6.4 revealing that for frames $2 \dots 5$ the model collapses to failure state where it is uncertain where each pedestrian will be located and rather predicts very rough estimate of all possible locations. This estimate can be interpreted as a probability map of the scene where pedestrians are likely to move. For prediction of second frame ahead we can also see that this fact is not that strong and a many pedestrian locations still can be extracted (Figure 6.4). More detail analysis is provided in Section 6.3.

From Figure 6.4 we can see that our ensemble model can make prediction for one and two frames ahead (0.8 and 1.6 seconds). The model likely learned to differentiate given frames from each other and then extract some basic information about changing pixels through convolutional operation. This is also suggested by the these probability maps shown in Figure 6.4 for prediction of $3, 4$ and 5 frames ahead since difference between 5 frames will result in similar indistinguishable map of pedestrians.

6.3 Error analyses

In this section we provide analysis of our approach and reasoning behind high error rates and failure states of the models. We can see from results provided in Section 6.2

that our models are able to accurately predict individual pedestrian locations in more than two step ahead (1.6+ seconds ahead). Reported errors increased from first step (0.8 seconds) to second step (1.6 seconds) roughly three times. From Figure 6.4 we can see that in steps 3, 4 and 5 models end up in some failure state where they are not able to conclusively decide where each individual pedestrian will be in the next steps. Models prediction can be thought of as a map highlighting often taken pedestrian trajectories through the scene and how these trajectories will change in future steps. In [5] authors report best MSE of 2.421 on dataset used in this work for prediction of 5 frames ahead. Constant velocity model which predicts future pedestrian coordinates as if pedestrian will walk in the same direction with constant velocity achieves a MSE of 6.091 on prediction of 5 frames ahead. Our best model achieves a MSE of 3.015 for prediction of 1 frame ahead and 12.780 for prediction of 5 frames ahead which is almost 6 times higher.

However, this high difference between results can be expected since most of previously mentioned methods get as whole or part of the input exact coordinates of a pedestrian in the last M frames of video. These locations then form a time series and this problem can then be treated as time series prediction. This approach poses a problem during test time where we do not have labels for each pedestrian location. During test time we first in the preprocessing step need to extract individual pedestrian coordinates. This can be done for example with some head-and-shoulders tracker or as in [5] authors use a KLT tracker. Extraction of exact pedestrian locations is difficult problem on itself since there are a lot of factors which influence the success rate of these methods, for instance, a perspective effect, resolution, lighting conditions and assignment of all coordinates in every given frame to the same pedestrian. In our settings models do not get this information and are force to extract these pedestrian trajectories from *raw* frames of the video and then from this extracted representation predict future state of the scene.

Our models are therefore trying to solve two problems at once. First one being extraction of individual pedestrian in given frames and connecting these coordinate in each frame to represent pedestrians prior movement. Second, from pedestrian representation try to predict future states of the scene where each pedestrian will be located in these future frames. This is inherently a more complex problem which requires that the model is able to form a representation of the scene, parts where it is possible to walk and is capable of extracting pedestrian locations given previous frames and is also able to capture in some cases a complex walking patterns of pedestrians. This would require a very large and complex model and therefore it is not surprising that our considerably shallow models fail to capture such complex information and accurately predict pedestrian movement.

Another issue is the before mentioned *sparsity* of the output. Our representation

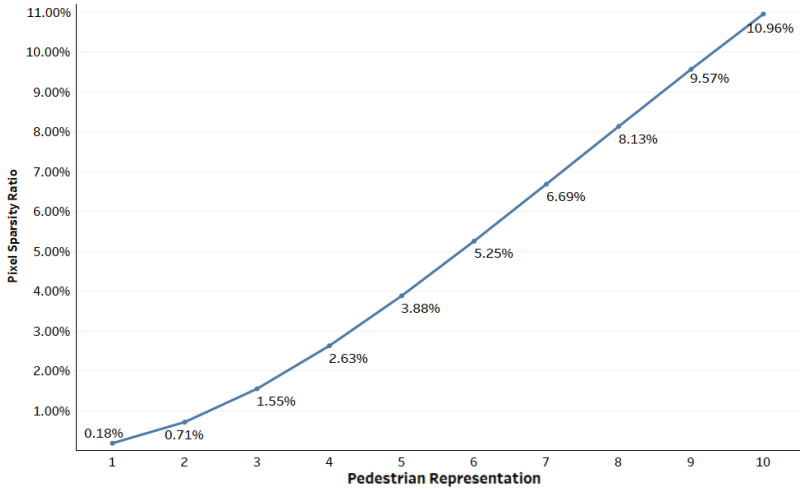


Figure 6.5: Ratio between zero pixels and pixels representing pedestrian in relation to increasing size of pedestrian representation.

of the each pedestrian by patch of white pixels of size 6×6 on average leaves more than 94.75% of pixels empty, as shown in Figure 6.5. This is not a very desired property due to the fact that model can set its weights to zero and predict zeros and will be 94% correct. As discussed in Section 6.2 we were unable to train models with smaller pedestrian representations due to this fact. As shown in Figure 6.5, one pixel representation of each pedestrian sets only 0.18% of pixels to some non-zero value therefore it is expected that our models would predict only the most common value. This fact also imposes a constraint on model architecture. A more complex model would be more susceptible to such a behaviour where it would set all the weights to zero and effectively stop training. To be able to accurately predict more frames ahead we need more complex models but due to the sparsity we are constrained by the depth of the model. This suggests that we need better pedestrian representation than a entire mask of the output where each pedestrian location is marked. This can also be seen from Table 6.3 where the models were not able to benefit from regularizing sparsity of hidden representation or by regularizing a false positive and negative which in theory should force the model to make more accurate predictions. With an increasing size of pedestrian representation, sparsity of the output becomes more manageable but merging of these individual representations into bigger blobs occur, as can be seen from Figure 6.6. To accurately predict each pedestrian location, models would have to be able to recognize a size of given blobs and infer how many pedestrians are represented by this blob and from previous information where each of them is located and where is heading. This increases complexity of the problem and would require a deeper model.

One solution to this problem is to use high dimensional *variational auto-encoder* which could be able to represent pedestrian locations and complex pattern through a probability distribution. Another, more complex solution is to use a convolutional neu-

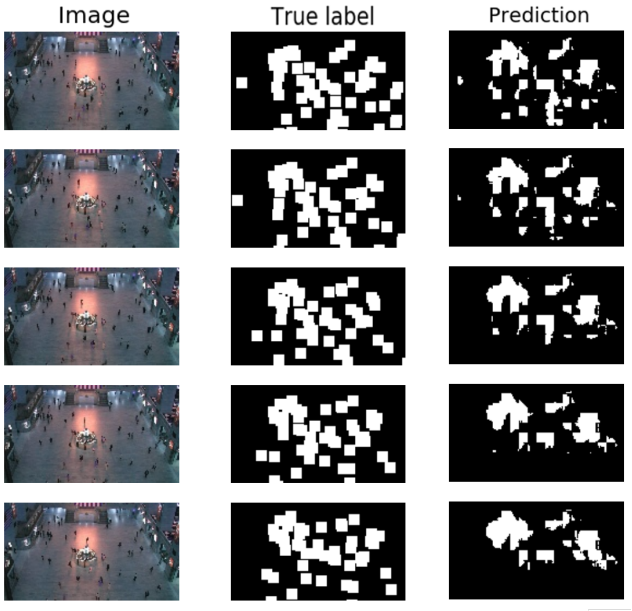


Figure 6.6: Column encoder predictions of future frames with pedestrian representation of size 10×10. Each row represents prediction of next time frame. Note, predictions images are shown after thresholding operations.

ral network to represent current scene and each pedestrian in this given scene and then use a recurrent neural network to predict pedestrian movement as a sequence. This would eliminate high sparsity of the output and could potentially yield a better prediction performance but would increase the size of the models and their computational and memory cost would also increase.

As mentioned in Chapter 5 we trained ensemble of models where each was used separate to predict each future frame. We can see from Figure 6.4 that even when we split this problem into separate models and each is given a task to predict only one frame the models that predict latter frames collapse to the same failure state. This fact alone strongly suggests that prediction of pedestrians in more than two frames ahead is very a complex problem. This can also be caused by the dynamic of the scene, i.e., in the similar public scene people do tend to walk straight from their entry point to the point of their interest. This means that in 4 seconds, which is our prediction window, a many new people will enter or leave the scene. For people entering the scene our model did not have any prior information about their movement and therefore could not predict with high certainty where their next locations are going to be. This consequently leads to predicting a highly dispersed probability map of likely pedestrian locations based on given input as can be seen in last rows of Figure 6.4.

Chapter 7

Conclusion and future work

In our work we tried to predict human behaviour in a public space, such as bus or train station from surveillance video with convolutional neural networks. Our approach is mainly based of work [5] where a deep convolutional neural network was used for prediction of human behavior on the dataset that was released as part of the work and used also in our thesis. This dataset contains 12684 annotated pedestrians in hour long video taken from Grand Central Train Station of New York. The images present in the dataset were sampled every 20 frames (0.8 seconds) and contain annotation errors. In our work, we provided detail analysis of the proposed dataset in Chapter 4. We also provide a description of our data preparation process in the same chapter. Due to our memory constrains we resized each image to size of 90×180. We used stacks of 5 frames per sample that was then split into smaller patches. These patches were then fed into our models. We tested different sizes of these patches. We achieved the best performance with a patch of size 9×8. Other results for different patch sizes are provided in Section 6.1.

The main difference and novelty of our approach to that proposed in [5] is that our models did not receive any information about previous pedestrian location, since this information is usually hard to obtain in the real world scenario. Often other technique needs to be used to extract exact pedestrian locations, such as KLT tracker or head-and-shoulders detectors. These technique usually were not designed for such a task and therefore often contain errors, for example, prediction of many more pedestrians then are currently located in the image or in case of head-and-shoulder detectors failing to predict any pedestrians in the image.

Our models were trained to extract individual pedestrian locations from raw frames of the video and then predict each pedestrian movement in the given scene. We tested several convolutional models whose detailed description can be found in Chapter 5. We achieved the best performance with our *convolution column stack classifier ensemble*, where the MSE for single frame prediction was 3.015 and for prediction of 5

frames ahead (4 seconds) 12.780. This model consisted of multiple convolution column classifiers each predicting a pedestrian mask for different time frame ahead. More detailed results for the other model with error analyses can be found in Section 6.2. We achieved satisfactory results for prediction of 1 and 2 frames ahead. For 3 to 5 frames our models failed to predict pedestrian locations to satisfactory degree. This could have been caused by many factors from which a capacity of our models and improper representation of the output are most probably the main contributors. Further analysis of these factors is provided in Section 6.3.

We also propose the improvements to our current approach, such as new model - variational encoder where each pedestrian and its trajectory would be modeled by some probability distribution. This should provide easier learning objective and therefore improve our results. Also, to better represent the final output, rather than predicting entire mask where each pedestrian is going to be located we can combine our convolutional encoders with recurrent neural networks. We could train a encoder to learn to find pedestrians in the current image. Its hidden representation can then be used as input to the recurrent neural network which should treat a pedestrian trajectory as time series prediction problem. This would eliminate the high sparsity of the output and also should provide improvement in performance.

Bibliography

[1] D. Helbing and P. Molnar, "Social force model for pedestrian dynamics," *Physical Review E*, vol. 51, no. 5, p. 4282, 1995.

[2] J. Xing, H. Ai, L. Liu, and S. Lao, "Robust crowd counting using detection flow," in *2011 18th IEEE International Conference on Image Processing*, pp. 2061–2064, IEEE, 2011.

[3] Y. Zhang, D. Zhou, S. Chen, S. Gao, and Y. Ma, "Single-image crowd counting via multi-column convolutional neural network," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 589–597, 2016.

[4] J. Liu, S. Zhang, S. Wang, and D. N. Metaxas, "Multispectral deep neural networks for pedestrian detection," *arXiv preprint arXiv:1611.02644*, 2016.

[5] S. Yi, H. Li, and X. Wang, "Pedestrian behavior understanding and prediction with deep neural networks," in *European Conference on Computer Vision*, pp. 263–279, Springer, 2016.

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

[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[8] F.-F. Li, A. Karpathy, and J. Johnson, "Cs231n: Convolutional neural networks for visual recognition 2016,"

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[10] A. Kumar, O. Irsoy, P. Ondruska, M. Iyyer, J. Bradbury, I. Gulrajani, V. Zhong, R. Paulus, and R. Socher, "Ask me anything: Dynamic memory networks for

natural language processing," in *International Conference on Machine Learning*, pp. 1378–1387, 2016.

[11] S. Zhang, R. Benenson, M. Omran, J. Hosang, and B. Schiele, "How far are we from solving pedestrian detection?," *arXiv preprint arXiv:1602.01237*, 2016.

[12] S. Yi, H. Li, and X. Wang, "Understanding pedestrian behaviors from stationary crowd groups," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3488–3496, 2015.

[13] S. Tang, M. Andriluka, A. Milan, K. Schindler, S. Roth, and B. Schiele, "Learning people detectors for tracking in crowded scenes," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1049–1056, 2013.

[14] J. Shao, K. Kang, C. C. Loy, and X. Wang, "Deeply learned attributes for crowded scene understanding," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4657–4666, IEEE, 2015.

[15] R. Mehran, A. Oyama, and M. Shah, "Abnormal crowd behavior detection using social force model," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on Computer Vision and Pattern Recognition*, pp. 935–942, IEEE, 2009.

[16] X. Wang, K. T. Ma, G.-W. Ng, and W. E. L. Grimson, "Trajectory analysis and semantic region modeling using nonparametric hierarchical bayesian models," *International Journal of Computer Vision*, vol. 95, no. 3, pp. 287–312, 2011.

[17] C. Huang, H. Ai, Y. Li, and S. Lao, "High-performance rotation invariant multiview face detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 4, pp. 671–686, 2007.

[18] J. Shi *et al.*, "Good features to track," in *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 593–600, IEEE, 1994.

[19] E. Bonabeau, "Agent-based modeling: Methods and techniques for simulating human systems," *Proceedings of the National Academy of Sciences*, vol. 99, no. suppl 3, pp. 7280–7287, 2002.

[20] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, no. 4, pp. 303–314, 1989.

[21] T. Kohonen, "The self-organizing map," *Neurocomputing*, vol. 21, no. 1-3, pp. 1–6, 1998.

[22] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. International Conference on Machine Learning*, vol. 30, p. 3, 2013.

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1026–1034, 2015.

[24] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, *et al.*, "Going deeper with convolutions," 2015.

[25] S. Wager, S. Wang, and P. S. Liang, "Dropout training as adaptive regularization," in *Advances in Neural Information Processing Systems*, pp. 351–359, 2013.

[26] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *International Conference on Machine Learning*, pp. 1058–1066, 2013.

[27] D. Warde-Farley, I. J. Goodfellow, A. Courville, and Y. Bengio, "An empirical analysis of dropout in piecewise linear networks," *arXiv preprint arXiv:1312.6197*, 2013.

[28] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International Conference on Machine Learning*, pp. 1139–1147, 2013.

[29] Y. Nesterov, "A method of solving a convex programming problem with convergence rate o (1/k2)," in *Soviet Mathematics Doklady*, vol. 27, pp. 372–376, 1983.

[30] Y. Nesterov, *Introductory lectures on convex optimization: A basic course*, vol. 87. Springer Science & Business Media, 2013.

[31] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 2011.

[32] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

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

[34] Y. Zhou and R. Chellappa, "Computation of optical flow using a neural network," in *IEEE International Conference on Neural Networks*, vol. 27, pp. 71–78, 1988.

[35] Y.-L. Boureau, N. Le Roux, F. Bach, J. Ponce, and Y. LeCun, "Ask the locals: multi-way local pooling for image recognition," in *Computer Vision (ICCV), 2011 IEEE International Conference on Computer Vision*, pp. 2651–2658, IEEE, 2011.

[36] Y.-L. Boureau, J. Ponce, and Y. LeCun, "A theoretical analysis of feature pooling in visual recognition," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 111–118, 2010.

[37] H. Bourlard and Y. Kamp, "Auto-association by multilayer perceptrons and singular value decomposition," *Biological cybernetics*, vol. 59, no. 4-5, pp. 291–294, 1988.

[38] G. E. Hinton and R. S. Zemel, "Autoencoders, minimum description length and helmholtz free energy," in *Advances in neural information processing systems*, pp. 3–10, 1994.

[39] G. E. Hinton and J. L. McClelland, "Learning representations by recirculation," in *Neural Information Processing Systems*, pp. 358–366, 1988.

[40] C. Doersch, "Tutorial on variational autoencoders," *arXiv preprint arXiv:1606.05908*, 2016.

[41] Y. Bengio, E. Laufer, G. Alain, and J. Yosinski, "Deep generative stochastic networks trainable by backprop," in *International Conference on Machine Learning*, pp. 226–234, 2014.

[42] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *Journal of Machine Learning Research*, vol. 11, pp. 3371–3408, 2010.

[43] J. Wang and L. Perez, "The effectiveness of data augmentation in image classification using deep learning," tech. rep.

[44] S. Bittel, V. Kaiser, M. Teichmann, and M. Thoma, "Pixel-wise segmentation of street with neural networks," *arXiv preprint arXiv:1511.00513*, 2015.

[45] Jariabka, Šuppa, and Rudolf, "Single camera path detection for outdoor navigation," in *Central European Seminar On Computer Graphics, 2017*, CESCG, 2017.

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

[47] T. E. Oliphant, *A guide to NumPy*, vol. 1. Trelgol Publishing USA, 2006.

[48] F. Chollet *et al.*, "Keras." `https://keras.io`, 2015.

[49] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 9, no. 1, pp. 62–66, 1979.

[50] Itseez, "Open source computer vision library." `https://github.com/itseez/opencv`, 2015.

[51] X. Wang, X. Ma, and W. E. L. Grimson, "Unsupervised activity perception in crowded and complicated scenes using hierarchical bayesian models," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 3, pp. 539–555, 2009.

Appendix A

Attached CD contains a `zip` file containing all training and testing scripts used in this thesis.

- `train.py` - A script used for model training
- `model.py` - Module containing definitions of the models
- `predict.py` - A script used for evaluation, prediction and displaying results
- `predict_ens.py` - A script used for evaluation, prediction and displaying results. This script was used for ensemble models
- `load_data.py` - Module containing helper functions for data preparation and loading
- `eval.py` - A script used for MSE evaluation of the models
- `show.py` - A script used for displaying the results
- `analyze.py` - A script used for analyzing the dataset
- `resized` - A directory containg a resized version of original dataset used in this work
- `models` - A directory containing trained ensemble model
- Original dataset can be found at `https://www.dropbox.com/s/7y90xsxq0l0yv8d/cvpr2015_pedestrianWalkingPathDataset.rar?dl=0`