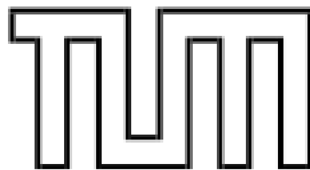
Human Pose Estimation with CNNs and LSTMs

Huseyin Coskun

Master's Thesis

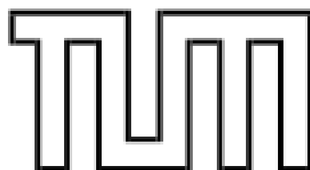


Computer Aided Medical Procedures & Augmented Reality
Department of Informatics
Technische Universität München

Human Pose Estimation with CNNs and LSTMs

Automatische Schätzung der Körperpose mit CNNs und LSTMs

Author :	Huseyin Coskun
Advisor :	Prof. Dr. Nassir Navab
Supervisors:	M.Sc. Felix Achilles
	Asst. Prof. Dr. Federico Tombari
Submission Date :	August 15th, 2016



Computer Aided Medical Procedures & Augmented Reality
Department of Informatics
Technische Universität München
Germany

Declaration

The work in this thesis is based on research carried out at the Chair of Computer Aided Medical Procedures & Augmented Reality, the Department of Informatics, Technische Universität München. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text. **Copyright © 2016 by HUSEYIN COSKUN**.

"The copyright of this thesis rests with the author. No quotations from it should be published without the author's prior written consent and information derived from it should be acknowledged".

Dedicated to

Eylul Gur

Abstract

Human pose estimation from images and videos has been a very important research field in computer vision. In this thesis, we present an end-to-end approach to human pose estimation task that based on a deep hybrid architecture that combines convolutional neural network (CNNs) and recurrent neural networks (RNNs). CNNs used to map the input image to feature space (fixed dimensionality), and then deep RNNs to decode the target sequence pose from the feature space. We experimented different RNNs architectures and we found out that deep bidirectional LSTM outperformed other architectures. Additionally, we tested different training strategies and influence of temporal information. Our final model is trained to minimize the average Euclidean distance between the ground-truth 3D joint coordinates and those predicted by our method.

To validate our approaches, we experimented on several datasets. Our experiments on Patient MoCap dataset outperformed algorithm, which is deemed state-of-art. We also evaluated on Human3.6M dataset, we achieved 19 cm. These results show the accuracy of the model and the integrity of the estimated pose that learned from the training data. Our quantitative and qualitative observations verify that our method makes significantly accurate. To the best of our knowledge, we are the first to show that CNNs+RNNs models can make accurate 3D joint coordinates estimation from depth images.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisors M.Sc. Felix Achilles and Asst. Prof. Dr. Federico Tombari for the continuous support of my master thesis and related research, for their patience, motivation, and immense knowledge. I could not have imagined having better supervisors and mentors for my master thesis research.

Besides my supervisors, I would like to thank my tutor, Dr. Llus A. Belanche from Universitat Politcnica de Catalunya for his insightful encouragement to do research.

My sincere thanks also go to Prof. Dr. Nassir Navab, who provided me an opportunity to join his team, and who gave access to the laboratory and research facilities. Without his precious support, it would not be possible to conduct this research.

I would also like to thank PhD students M.Sc. Leslie Casas and M.Sc. David Tan from Technische Universität München for their support.

Last but not the least, I would like to thank my family: my parents, my sisters, and my girlfriend for supporting me spiritually throughout writing this thesis and my life in general.

Contents

Declaration **iii**

Acknowledgements **vi**

1 Introduction **1**

1.1 Motivation 2

1.2 Problem Statement 3

1.3 Related Work 4

1.4 Overview of Thesis 5

2 Methods **6**

2.1 Basic Concepts of Neural Networks 6

2.1.1 Activation Functions 6

2.1.2 Objective Functions 9

2.1.3 Regularizations 10

2.1.4 Optimization Strategies 11

2.2 Convolutional Neural Networks 12

2.2.1 Building Blocks of the CNN 13

2.2.2 Training CNN 17

2.3 Recurrent Neural Networks 18

2.3.1 Vanilla RNNs 19

2.3.2 Long Short Term Memory (LSTM) 22

2.3.3 Gated Recurrent Units (GRU) 27

3 CNN+RNN Architectures for Human Pose Estimation 29
3.1 Motivation . . . 29
3.2 CNN+RNN Architectures . . . 30
3.3 CNN+BiRNN Architectures . . . 32
3.3.1 Bidirectional RNNs . . . 32
3.4 Forward and Backward Pass . . . 34
3.5 Computational Complexity . . . 37
3.6 Output Layer . . . 37
3.7 Training Strategies . . . 38
3.7.1 Persistent and Non-persistent Training . . . 40
3.7.2 Training with Sliding Window . . . 40

4 Experiments 42
4.1 Implementation and Hardware . . . 42
4.2 Tuning of Hyperparameters . . . 42
4.3 Data Sets . . . 43
4.3.1 Patient MoCap Dataset . . . 43
4.3.2 Human3.6M Dataset . . . 47

5 Conclusions 53
5.1 Summary . . . 53
5.2 Future Work . . . 54

Bibliography 55

List of Figures

1 Single neural network node . . . 7
2 Activation functions . . . 7
3 LeNet . . . 13
4 Pooling . . . 15
5 AlexNet . . . 15
6 AlexNet learned filters . . . 16
7 Recurrent neural network . . . 20
8 Unrollded RNN . . . 21
9 Vanishing gradient . . . 23
10 Working memory model (WMM) . . . 23
11 LSTM cell . . . 24
12 Unrolled Deep LSTM . . . 26
13 Gated Recurrent Units (GRU) . . . 28
14 CNN+LSTM architecture . . . 31
15 Bidirectional RNN . . . 33
16 CNN+LSTM workflow . . . 38
17 Blanket occlusion . . . 45
18 Average per joint error on the on Patient MoCap dataset . . . 47
19 Patient joint estimation . . . 48
20 Human3.6M dataset recording laboratory setup . . . 49
21 Sample depth from Human3.6M . . . 49
22 Human3.6M dataset joint estimation . . . 51
23 Human3.6M best model accuracy . . . 52

List of Tables

4.1 Patient MoCap dataset results 46
4.2 Patient MoCap dataset persistent state comparison 46
4.3 Patient MoCap dataset training strategy comparison 46
4.4 Human3.6M dataset results 50

Chapter 1

Introduction

General environment reliable human pose estimation from images and videos has always been a very important research field in computer vision. It has a wide range of applications from gaming over human-computer interaction, to security, and even health-care, therefore human pose estimation is one the most fascinating research fields. Over the last couple of decades, computer vision has made significant progress in pose estimation, which is fuelled by new optimization and feature engineering strategies. In the early period of human pose estimation research, pictorial structure (PS) models [36, 37] were the state of art. PS models estimate the human pose by building the human body with collections of parts arranged in a deformable setting. The launch of Microsoft's Kinect depth sensor cameras in 2010 made dramatic changes in this field. It opened a new door to human pose estimation researchers by providing depth data. Plenty of recent academic works use depth data for pose estimation. Girshick et al. [30] used depth information and made a big step towards a reliable solution with a regression forest algorithm. Even though the depth stream is quite useful for this task, there are still only a handful of large depth datasets, therefore monocular approaches (from RGB to 3D or 2D) still dominate at the pose estimation task.

Pose estimation is still a largely unfinished task. There are various problems that are either not solved or partially solved. The most important problems are: (a) clothing, subject size, and gender differences, (b) partial occlusions by body parts or other objects in the scene (c) variability of human pose, (d) loss of depth

information. Until now, there has not been a single approach that provides solutions for a general environment human pose estimating task.

In this thesis, we thus address the problems (a,b,c,d) for general environment pose estimation. Similar to [30], we used depth data in this research, however, we took advantage of recent developments in deep learning. Hence, this thesis proposes a novel method based on CNNs and recurrent neural networks (RNNs). Our approach is different from state-of-the-art methods [33–35, 38] in many ways. Even though were inspired by these approaches, our idea relies on a completely new hybrid model(CNNs+RNNs).

To validate our approaches, we evaluated our models on two different datasets, containing challenging poses from different subjects. We examined the influence of different training strategies and temporal information. Furthermore, we show the accurate and stable results on both datasets. Preliminary results on the Patient MoCap dataset [31] suggest applicability of our approach in heavily occluded scenarios. The outcome of our approach is a compact CNNs+RNNs model that reaches comparable performance on standard benchmark dataset [32] and the best result on the Patient MoCap dataset, but does not employ a complicated network architecture [33] nor pre-training on different task [34].

1.1 Motivation

Recently, CNNs have achieved great success in many computer vision applications, and CNNs have been shown to be good at disentangling factors in data. They are currently the most popular architectures for vision problems, since they learn features automatically from datasets, which makes them appealing for a variety of tasks. In addition, while the convolutional and max-pooling structure reduces the number of parameters, it also enables the network to extract translation and illumination invariant features.

We have seen similar success with RNNs. They have achieved unprecedented performance in tasks on sequential data. RNNs have several properties that make them an attractive choice for those tasks. They are able to incorporate contextual

information from past frames (and future frames too, in the case of bidirectional RNNs), which allows them to map a wide range of frames to a sequence of estimated joints. Furthermore, they are robust to temporal "noise", i.e. disturbances of the scene along the time axis. This allows them to handle heavily occluded scenarios. Vanilla RNNs are bad at leveraging long term temporal dependencies. This is why in our models, we choose Long Short-Term Memory (LSTM) RNNs, which are designed to solve this problem. It is proven that LSTMs have the power to map very long sequences between relevant input and target events for various real world and synthetic tasks [15].

In this thesis, we combine the power of CNNs and RNNs to learn human pose estimation. To the best of our knowledge, we are the first to show that CNN+RNN models can be applied to 3D human pose estimation from depth images.

1.2 Problem Statement

Human pose estimation task can be defined as the problem of localizing human joint coordinates in either 2D or 3D. This task includes subtasks as well, which are detecting the subject(s) in the image, and the segmentation of human body parts. Recent deep learning models can handle these tasks with a single procedure. We are interested in finding a posterior distribution of the pose parameters that explains the current pose, given previous and current frames.
Pose parameters in 3D:

$$y_t^i \in R^3, i = 1 : K \qquad \text{K is the number of joints.} \tag{1.2.1}$$

Current depth frame:

$$x_t \in R^{N \times M} \qquad \text{Note that a depth frame has single channel.} \tag{1.2.2}$$

We are interested in finding posterior distribution that explains the current pose parameters.

$$\prod_{i=1}^{K} p(y_t^i \mid x_{1:t}) \qquad \text{This formulation can change in case RNNs.} \tag{1.2.3}$$

y_t is an N-dimensional vector that hold each body part's position, orientation and scale. It represents an articulated 3D human body that is observed in the input frame x_t.

1.3 Related Work

There is plenty of research in human pose estimation. In the early stage of research, most of the works used pictorial structures [39, 52]. These approaches aim at detecting the subject and then determine its upper body pose. In these algorithms, the human pose estimation task is modelled as a graph labelling problem. They suffered problem complexity since inference on graph labelling problem was intractable: with n labels and h relevant discrete locations for each part, its complexity is $O(h^n)$. Felzenszwalb and Huttenlocher [39] bring the solution with assuming the model as a tree and using distance transformation. As an outcome, a great amount of PS based algorithms were developed [40–42] and proved to be a powerful solution for 2D pose estimation. Belagiannis et al. [44] proposed a PS based method for 3D pose estimation using multiple views. Despite the polynomial complexity and great successes, tree-structured models suffer from the problem of confusing left and right joints, which often happens to limbs. This problem became more dramatic at 3D joint coordinates pose estimation.

To overcome these problems of PS based models, there has been a lot of effort that focused on making more powerful models, particularly by solving this left-right confusion [45, 46] by modelling the problem as a non-tree structure. These methods achieved better results by using Branch and Bound (BB) [47] to search for the globally optimal solution. Despite their success, they only brought an approximate solution to the problem. Moreover, such approaches were based on hand-crafted features (e.g., HOG and SIFT), and they were hence limited by the representation ability of these features.

Recently, CNNs have made great success in computer vision tasks, hence many researchers applied CNNs to the human pose estimation task. Toshev et al. [38] used CNNs to estimate the direct human joint coordinates. Ouyang et al. [48]

have proposed a multi-source (visual appearance score, deformation and appearance mixture type) deep model for pose estimation. However, these approaches suffered from inaccuracy especially for the limbs. Tompson et al. [49] overcame this problem by using translation invariant CNNs. Tekin et al. [37] used structured prediction models that combine auto-encoders and CNNs yielding state-of-art results on the Human3.6M dataset [32]. The closest work to ours uses CNNs together with RNNs to regress joint coordinates [50]. However, this work does not employ bidirectional RNNs, infers 2D pose only, and uses traditional RGB frames. Furthermore, the authors formulate the 2D pose estimation as a classification task, which only allows a coarse estimation of each joint position on a predefined grid. By contrast, here we exploit bidirectional RNNs to learn the temporal patterns from depth data for real valued regression task.

1.4 Overview of Thesis

In the rest of this thesis, we present the details of our approach to human pose estimation. **Chapter 2** reviews basic concepts used in neural networks, focused on their use in our models. It also provides a formal CNNs and RNNs definition, and presents equations for the CNN and RNN error gradients. **Chapter 3** describes the CNN+RNN architectures, and intuition behind the hybrid models. It also presents learning strategies and computational complexity for these models. **Chapter 4** gives the implementation details and hardware. Moreover, it presents the publicly available datasets, namely Patient MoCap and Human3.6M. It also shows our results on these datasets and discusses them. **Chapter 5** provides a conclusion and presents possible future directions.

Chapter 2

Methods

In this chapter, we will explain the methods used in our model. Section 1 focuses on basic concepts of neural networks, section 2 explains CNNs, and finally section 3 describes RNNs.

2.1 Basic Concepts of Neural Networks

In this part, we will explain basic concepts of neural networks, namely, activation functions, objective functions, regularizations, and optimization strategies. We will focus on the concepts which are used in our model. The aim of this section is to give the reader a sufficient understanding of the matter, so that the Methods chapter is relatively self-contained. For a more complete introduction, the works by LeCun et al. [6] and Bengio et al. [12] are recommended.

2.1.1 Activation Functions

In neural networks, the activation function refers to a nonlinear function that converts the input value to the node's output value (see figure 1). During training we need to compute the gradient of output w.r.t weights and input. Because, gradient of the function becomes important. For this reason, the logistic type functions are the most used functions as an activation function, since it is easy to calculate its gradient. However, theoretically any differentiable function can be used. Most commonly used functions are: sigmoid, tanh, and rectified linear units [1]. We will

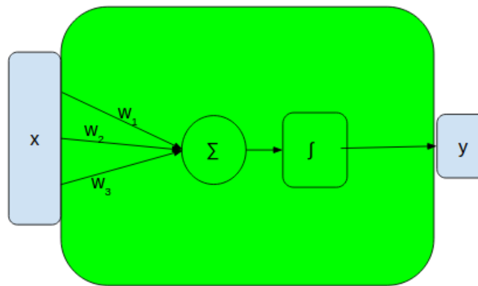


Figure 1: Figure illustrates the single neural network **node**. While x represents the **input**, w_i represents the weights. **Output** (y) is activation function applied dot product of x and w. Biases are omitted for simplicity.

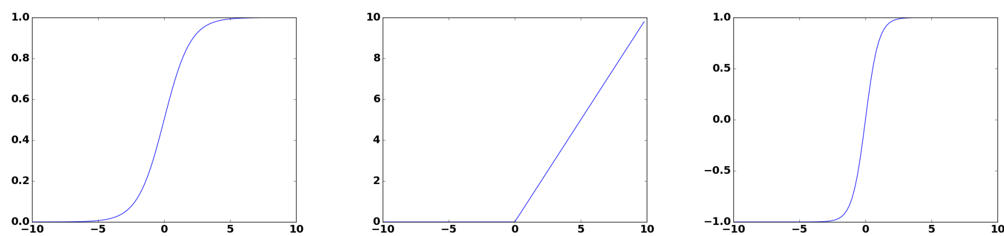


Figure 2: **Left**: Sigmoid squashes input to range between [0,1], **Middle**: ReLu function, which is zero when x <0 and then linear with slope 1 when x >0. **Right**: The tanh squashes input to range between [-1,1].

briefly explain these functions here.

Sigmoid

Sigmoid function mathematical form:

$$sigmoid(x) = 1/(1 - e^x) \tag{2.1.1}$$

Properties of sigmoid function:

- High negative values converge 0 and high positive values converge 1.
- Sigmoid is a strictly increasing function between 0 and 1.

+ Sigmoid function squashes the input into range between 0 and 1.
+ It exhibits a balance between nonlinear and linear behavior.

− The popularity of sigmoid function decreased with development of deep models, since it saturates and kills gradients. When neurons output converges

either 0 or 1, the local gradient at these regions become almost zero and this will prevent the higher level layers training, hence local gradient is very crucial for back propagation.

– Neurons at the higher layers of the neural network will receive the data that is not zero-centered, since sigmoid function output is not zero-centered which is very harmful for the performance of network. These neurons will get the data that always positive, then the gradient on that neurons will be either all positive or all negative at backward pass which can cause jittering for updates of weights during training with small batch sizes.

Tangent hyperbolic

The mathematical formulation of a tangent hyperbolic activation function is:

$$\tanh(x) = sin(x)/cos(x) = (e^x - e^{-x})/(e^x + e^{-x}) \tag{2.1.2}$$

Properties of tanh function:

- Tanh function squashes input to between -1 and 1.
- Tanh function is a strictly increasing function between 0 and 1.

+ Tanh function out is zero-centered, because of this, tanh performs better than sigmoid in practice.

– Similar to the sigmoid functions, it also has the output saturation problem.

ReLu

The mathematical formulation of a rectifier linear unit activation function is:

$$ReLu(x) = max(0, x) \tag{2.1.3}$$

Properties of tanh function:

- ReLu function output can range [0,∞).

+ Not facing gradient vanishing problem is the main reason of ReLus popularity increase together with the rise of deep networks.

+ It can greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid/tanh functions.
+ ReLu can be computed efficiently. Because, unlike sigmoid and tanh functions, ReLu does not require any exponential computation.
− ReLu can produce the same output, and when turns out to be in this state, it is unlikely rescued from this state.

2.1.2 Objective Functions

Human pose estimation is a task of estimating joint locations. For this task, it is common to compute the loss between the predicted joint locations and the ground truth and then measure the L2 norm or L1 norm of the difference. During training, we need to compute the gradient of objective function w.r.t the network parameters. There are many different objective functions in literature, but in this thesis, we will be using the L2 and L1 objective functions.

L2 loss

L2 loss can be formulated with this form:

$$L2_{loss} = \sum_{i=0}^{n} (y_i - h(x_i))^2 \tag{2.1.4}$$

$$\nabla L2_{loss} = 2 * \sum_{i=0}^{n} (y_i - h(x_i)) \nabla h(x_i) \tag{2.1.5}$$

While x represents the input, y represents the output of the network which is represented with h. L2 loss sums the squared difference between target value and prediction for all outputs. As it can be seen from the equation 2.1.5, L2 loss has a simple gradient. Because of this, it is widely used. Unlike L1 loss, L2 loss has a single solution, since taking square does not change the optimal parameters.

L1 loss

We can formulate L1 loss with that form

$$L1_{loss} = \sum_{i=0}^{n} (|y_i - h(x_i)|) \tag{2.1.6}$$

$$\nabla L1_{loss} = \sum_{i=0}^{n} (\frac{y_i - h(x_i)}{|y_i - h(x_i)|}) \nabla h(x_i) \tag{2.1.7}$$

Here we sum the absolute value of the difference between target value and our prediction. As can be seen from equation 2.1.4, the L2 error will be much larger for outliers compared to L1, as the difference between a wrong prediction and an original target value will already be quite big and squaring it will make it even bigger. As a result, L1 loss is more robust to outliers.

2.1.3 Regularizations

Regularization plays an important role to increase test time accuracy. Most of the time, the models are designed very complex and nonlinear, therefore we need to decrease the capacity of the model so that while it can learn patterns in the data, we can prevent the model from overfitting on training data. There are many different ways of doing regularization, but in our model we used 3 types of regularization strategies, namely L2 and L1 parameter regularizations and dropout.

L2 parameter regularization

L2 parameter regularization is a very simple form of regularization.

$$L2_{reg} = \sum_{i=0}^{n} w^2 \tag{2.1.8}$$

$$regularizedL2_loss = \sum_{i=0}^{n} (y_i - h(x_i))^2 + \sum_{i=0}^{n} w^2 \tag{2.1.9}$$

L2 regularization is adding equation 2.1.8 to the objective function, for instance equation 2.1.9 shows the regularized L2 loss. Adding this term enforces the parameters to be close to the origin. Another consequence of L2 regularization is that the model perceives the input as having a higher variance, hence the model parameters will shrink on features whose covariance with the output target is lower compared to this added variance.

L1 parameter regularization

Formally, L1 regularization is defined as:

$$L1_{reg} = \sum_{i=0}^{n} |w| \tag{2.1.10}$$

L1 regularization is adding equation 2.1.10 to the objective function. L1 regularization enforces the model to be sparse which is very important for network performance, by means of its role for simplifying the problem via feature selection mechanism.

Dropout

Dropout [3] is another way of doing regularization. Dropout is different than L2 and L1. In the case of L2 and L1, we modify the objective function, but here we modify the model and train the modified network over different batches. The idea of training different models on different data comes from bagging, which means training multiple models, and evaluating multiple models on each test example. This is impractical when each model is a large neural network, since training and evaluating such networks is costly in terms of runtime and memory. Dropout provides us with a way to do an inexpensive approximation to training and evaluating a bagged ensemble of exponentially many neural networks.
Let l represents the layer index of the network. y^l is the l-th layer input and p is the probability. We can define dropout such a way:

$$r_j^l \sim Bernoulli(p)$$
$$\hat{y^l} = r^l * y^l$$

2.1.4 Optimization Strategies

Optimization is the main task in dealing with machine learning problems, hence a good optimization strategy yields better results. While working on the human pose estimation problem, even though we tried different types optimization methods,

because of space constraints I'll mention ADAM [4] that we used in our final model and stochastic gradient descent which is the basis of ADAM.

Stochastic gradient descent (SGD)

Stochastic gradient descent is a gradient based learning algorithm. SGD is very commonly used an algorithm in machine learning. With SGD, we can obtain an unbiased estimate of the gradient by taking the average gradient on a mini-batch of m examples which is independent and identically distributed with the data generating distribution (See algorithm 2 for more details).

Adaptive moment estimation (ADAM)

Adaptive moment estimation (ADAM) [4] is a gradient based learning method that computes adaptive learning rates for each parameter. During learning, exponentially decaying average of past gradients and its square are stored. This helps to better converge to the minimum point. Even though we explain the ADAM algorithm more detailed at section 3.7, the ADAM update rule is shown below:

$$m_{t+1} = \gamma_1 m_t + (1 - \gamma_1)\nabla\mathcal{L}(W_t) \tag{2.1.11}$$

$$g_{t+1} = \gamma_2 g_t + (1 - \gamma_2)\nabla\mathcal{L}(W_t)^2 \tag{2.1.12}$$

$$\hat{m}_{t+1} = \frac{m_{t+1}}{1 - \gamma_1^{t+1}} \tag{2.1.13}$$

$$\hat{g}_{t+1} = \frac{g_{t+1}}{1 - \gamma_2^{t+1}} \tag{2.1.14}$$

$$w_{t+1} = w_t - \frac{\eta \hat{m}_{t+1}}{\sqrt{\hat{g}_{t+1}} + \epsilon} \tag{2.1.15}$$

$+\gamma_1$, γ_2, and η are hyperparameters that can be changed for different models. We initialize $m_0 = 0$ and $m_1 = 0$.

2.2 Convolutional Neural Networks

CNNs were first introduced by LeCun et al. in [6]. We can consider CNNs as biologically-inspired versions of multilayer perceptrons (MLPs) for vision tasks. Researchers revealed that the visual cortex contains a complex arrangement of receptive

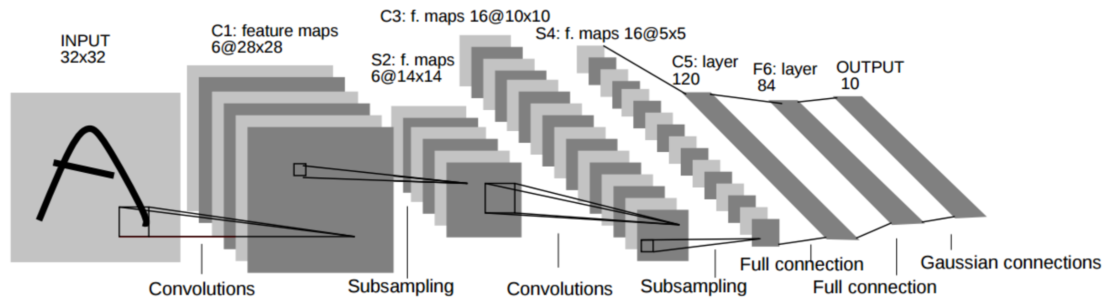


Figure 3: Convolutional Neural Networks (CNNs). The LeNet has a convolution layer followed by a pooling layer, and then another convolution followed by a pooling layer. After that, two densely connected layers are added. Taken from [5]

neurons and each of those neurons are sensitive to a specific part of the visual field which is called receptive field. Those neurons act like local filters over the input space and are well-suited to exploit the strong spatially local correlation that exists in the information coming from eyes. There are two distinct cell types. These are simple neurons that focus on edge-like patterns within their receptive area and complex neurons that focus on patterns that are locally invariant to the exact position of the information.

2.2.1 Building Blocks of the CNN

Even though there are many different architectures for CNNs in the literature, the majority of them can be built by stacking three main type of layers in different combinations. Namely, the convolutional layer, the pooling layer and the fully connected layer. In this section, we will explain those layers.

Convolutional layer

The convolutional layer aims to learn a feature representations of the inputs. As shown in Figure 3, a convolutional layer is composed of several feature maps. All neurons in the feature map are connected to the neurons in the previous layer. Such a connection is referred to as the neurons receptive field in the previous layer. A forward pass through a convolutional layer means to compute feature maps from

the input. This process can be done in such a way: Initially, we convolve input feature maps with learned filters and then we apply the activation function to these outputs. Usually, filters are spatially small, but they have the same depth as the input. Such a filter on a first layer of a CNN may have size $9 \times 9 \times 1$ (i.e. 9 pixels width and height, and 1, because depth images have one channel) for depth images. We compute the feature maps by sliding the filters over input data (or map). This process will produce an activation map that gives the responses of those filters at any spatial position. Ideally, the network will learn filters that fire when they see specific visual patterns such as a blotch of some color on the first layer or an edge of some orientation, or wheel similar shapes on last layers of the network. A converged network will have a set of filters in each layer (e.g. 128 filters), and every filter will produce an activation map.

Pooling layer

Pooling is an important concept of CNNs. As it can be seen from figure 3, it lowers the computational burden by reducing the number of connections between convolutional layers. The pooling layer aims to achieve spatial invariance by reducing the resolution of the feature maps. It is usually placed between two convolutional layers. Each feature map of the pooling layer is connected to its corresponding feature map of the preceding convolutional layer. Thus, they have the same number of feature maps. The typical pooling operations are average pooling and max pooling. Max pooling is the most commonly used type of pooling. It controls the image weather given feature exist or not. If exist, it then returns the precise position. The intuition is that when a feature has been detected, its exact location isn't as critical as, since its approximate position relative to other features. Since there are many fewer pooled features, required number of parameters will decrease in the lower layers.

Fully connected layer

Fully connected layers are responsible for high level reasoning in the neural networks.Fully connected layers are usually last layer of the neural networks. A fully

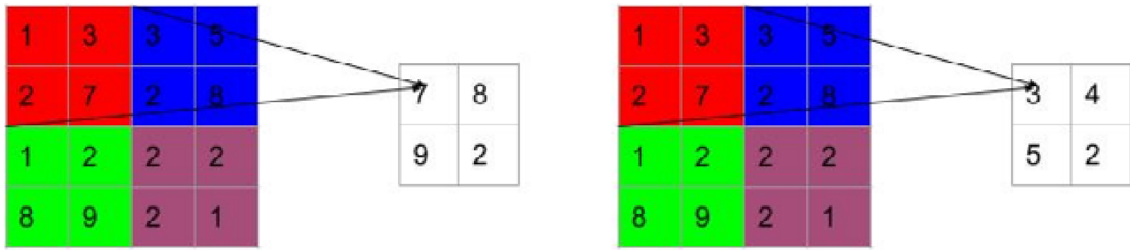


Figure 4: Pooling operation down samples the volume spatially, independently in each depth slice of the input volume. Left: Max poling applied to the input volume of size [4 × 4] with filter size 2, stride 2 and output volume of size [2 × 2]. Right: Average pooling applied to the input volume of size [4 × 4] with filter size 2, stride 2 and output volume of size [2 × 2]. Best viewed in color.

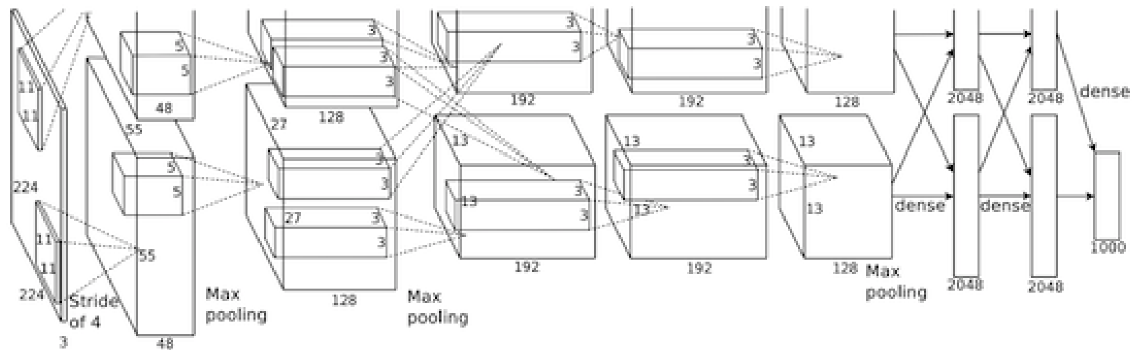


Figure 5: AlexNet is a revolutionary model for image classification and it is considered first successful deep CNNs. AlexNet that trained on ImageNet dataset achieved the best results. It dropped the state-of-the-art error percentage from 47.1% to 37.5% (for top-1 classification). Taken from [2]

connected layer takes all nodes in the previous layer (can be pooling or convolutional) and connects it to every single neuron it has. After fully connected layers, network looses the spatial information in the data(you can visualize them as one dimensional), hence there can not be the convolutional layers after a fully connected layer. Output of the last fully-connected layer will be fed to loss layer. For classification tasks, softmax is commonly used as it generates a well-formed probability distribution of the outputs, but for regression tasks, regression layer is used.

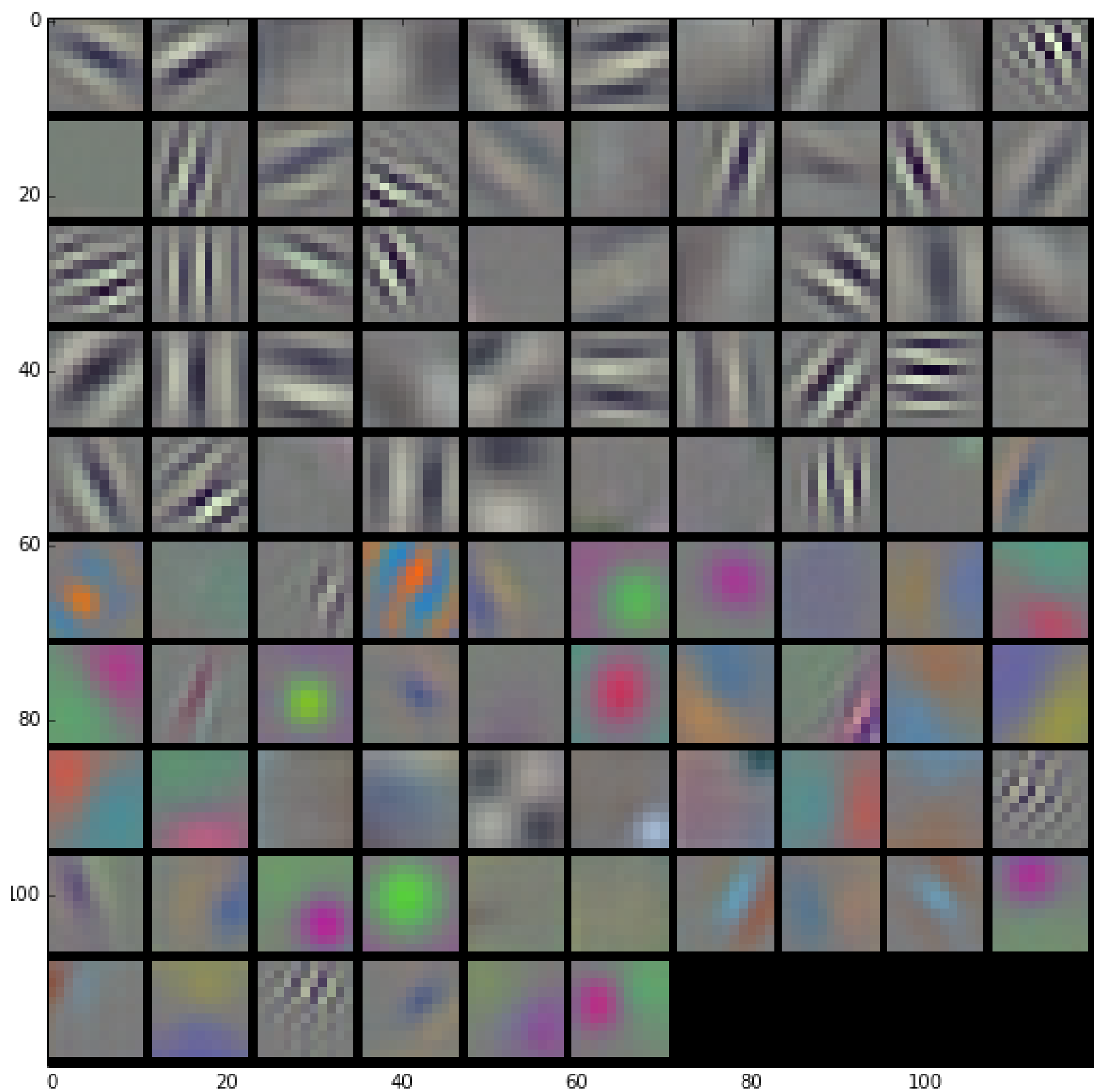


Figure 6: Figure illustrates the AlexNet filters at the first layer. There are 96 filters with $11 \times 11 \times 3$ size. They are learned at the first convolutional layer on the $224 \times 224 \times 3$ input images with using two different GPUs. From figure it can be seen that there are many different kernels which focus on frequency, edges of different orientations, phase, and colors. First 48 filters learned on GPU 1 and the last 48 filters learned on GPU 2. Filters that learned on GPU 1 are largely color-agnostic and others are largely color-specific. Best viewed in color. Taken from [25]

2.2.2 Training CNN

Training the CNNs is quite similar to the ordinary neural networks. We train the CNNs with forward passing the input and then backward passing the gradient.

Forward propagation

During training CNNs, we should pass input data in the forward direction. Let us assume that we have input size $N \times N$. Our CNN has an $m \times m$ filter with depth d, convolutional layer output will be of size $[(N - m + 1), (N - m + 1), d]$. Unit input pre-nonlinearity computed, after all the contributions that weighted by the filter components aggregated.
Applying convolution operation:

$$X_{ij}^{l} = \sum_{a=0}^{m} \sum_{b=0}^{m} W_{ab} \gamma_{(i+a)(j+b)}^{l-1} \tag{2.2.16}$$

After we perform the convolution operation, we can apply the activation function:

$$\gamma_{(i)(j)}^{l} = f(X_{ij}^{l}) \tag{2.2.17}$$

Most times, convolution layer is followed by pooling layer. In case of max pooling, our output dimensionality will be like that: $[(N - m + 1)/2, (N - m + 1)/2, d]$. Max pooling does not change the depth of an input. Each 2×2 block is reduced to just a single value via the max function (see figure 4).

Backward propagation

We will use backpropagation algorithm for backward pass. Backpropagation [6] is simply a layer-wise application of chain rule for partial gradients. The backpropagation algorithm tries to find the best parameters, which yields minimum error, using the method of gradient descent. The combination of parameters, which gives the minimum error, is accepted as solution of the learning problem. We must use a continuous and differentiable the error function for gradient descent method. First step of the backpropagation is calculating the gradient of the objective function with respect to the output layer parameters. Let us assume that we have an error function, and we know the error values at our convolutional layer. What, then, are the

error values at the previous layer of it, and what is the gradient for each parameter in that layer?

Let δ^{l+1} be the error term for the $(l+1)$-st layer in the network with an objective function J.

$$\delta_k^l = upsample((W_k^l)^T \delta_k^{l+1}).f'(z_k^l) \tag{2.2.18}$$

Where k is the index for the filters and $f'(z_k^l)$ is the gradient of the activation function. The upsample operation has to send error back through the layers by computing the error with respect to each item incoming to that layer. For instance, in the case of the max pooling the weights, which were selected as the max receives all the error because any changes in input would change the output only from that unit. For the mean pooling then upsample equally distributes the error for a that pooling unit to the units which feed into it in the previous layer.

Finally, we compute the gradient w.r.t to the filter maps by flipping the error matrix δ_k^l similar to the filters in the convolutional layer. Let assume that ai^l is the i th input for l th layer.

$$\Delta_{W_k^l} J = \sum_{i=0}^{m} a_i^l * \delta_k^{l+1} \tag{2.2.19}$$

$$\Delta_{b_k^l} J = \sum_{a,b=0}^{m} (a_i^l)_{a,b} \tag{2.2.20}$$

After we compute the gradient for each weight, we can update weights with equation 2.2.19 and 2.2.20. We used ADAM [4] optimisation strategy for updating weights. Details of this method can be seen in section 2.1.4

2.3 Recurrent Neural Networks

In computational neuroscience, there are different methods, which attempt to imitate human intelligence and connectionism is one of the most successful approaches. Connectionism is modelling the cognitive processes using interconnected networks of simplified neuronlike computational units, they used the recursively connected models. This idea has been applied to a diverse range of cognitive abilities, including

models of memory, attention, perception, action, language, concept formation, and reasoning. After seeing the importance of memory, particularly in working memory, many computational scientists tried to imitate it. First attempt to model memory was made by Geoff Hinton and Jim Anderson in 1986. Hinton and Anderson used parallel associative models to model memory, which presented an alternative computational framework for understanding cognitive processes [16]. In 1997, Hochreiter & Schmidhuber proposed a different method by modeling the working memory with a dynamic gating architecture [17], which adjusted the influence of the input on the system. In 2001,Frank, Michael J., B. Loughry, and Randall proposed a more biologically plausible model [18]. Their work relied on the selective gating architecture, but they did not use back propagation for the learning procedure. In 2006, Hazy, Thomas E., Michael J. Frank, and Randall C proposed a method in which basal ganglia dynamically update the representations in the prefrontal cortex, this method alike the Long ShortTerm Memory architecture [20]. In 2011, Oberauer, Klaus, and Stephan Lewandowsky tried to model working memory with using time based on resource sharing theory [19]. In other works [16–20] models were built on the idea that dopamine act like a gating mechanism, which controls influence of the other cortical input impact on frontal cortex. Even though methods [16,18–20] were biologically more plausible, unfortunately they were not successful on benchmarks. On the other hand, LSTM networks are state of art at many sequential tasks with some variants.

2.3.1 Vanilla RNNs

Recurrent neural network is a network that has at least one feed back connection. RNNs differ from feed forward neural networks (FFNs) with cyclic connections. These cyclic connections enable the activations flow round in a loop, therefore networks can do temporal processing and learn sequences, e.g., perform sequence recognition/reproduction or temporal association/prediction. This makes RNNs more powerful than FFNs for sequential tasks. They have been successfully used for sequence labeling and sequence prediction tasks, such as handwriting recognition, language modeling, and phonetic labeling of acoustic frames. First RNN model pro-

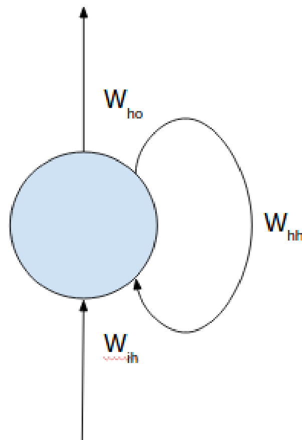


Figure 7: A recurrent neural network with a single unit. W indicates the network parameters. Best viewed in color.

posed in the 80s [7–9] were for modeling sequential data. In this part, we focus on a vanilla RNN containing a single self connected hidden layer. Figure 7 illustrates the single cell RNN.

As it can be seen from Figure 7, there are many differences between RNN and multilayer perceptron. The most striking difference is: An MLP can only use limited number of input during prediction, whereas an RNN can in principle use the entire history of previous inputs to each prediction.

Let inputs and outputs are vectors are x^t and y^t, the three connections weight matrices are W_{ih}, W_{hh}, and W_{ho} and the hidden and output unit activation functions are f_h and f_o. We can formulate RNN such a way that:

$$a_h^t = W_{ih}x^t + W_{hh}h^{t-1} \tag{2.3.21}$$

$$h^t = f_h(a_h^t) \tag{2.3.22}$$

$$y^t = f_o(W_{ho}h^t) \tag{2.3.23}$$

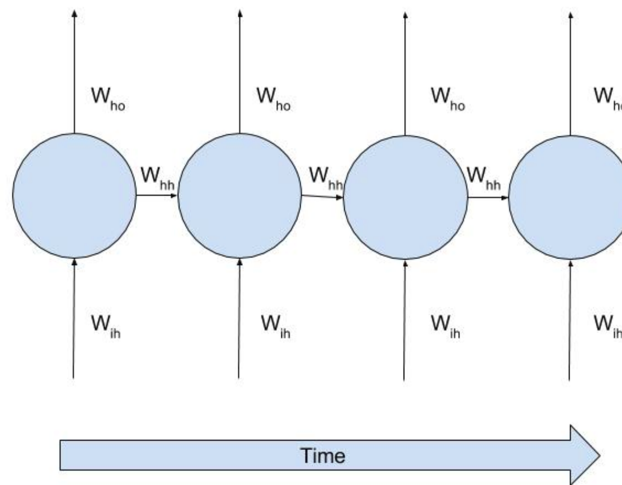


Figure 8: Unrolled version of RNN. Each cell indicates one step in time. The w_{ih}, w_{hh}, and w_{ho} are representing the gates from the input to hidden and hidden to hidden, and hidden to output respectively. Note, same weights are used for each time step. Best viewed in color.

Forward pass

We can see from Figure 8, unrolled version of RNN is same as the MLP, since forward pass of an RNN is quite similar to the multilayer perceptron. The only difference is that activations arrive at the hidden layer from both the current external input and the hidden layer activations from the previous time step.

Matrix form of the equation 2.3.21 will be like that:

$$a_h^t = \sum_{i=1}^{I} W_{ih} x_i^t + \sum_{h'=1}^{H} W_{h'h} h_{h'}^{t-1} \tag{2.3.24}$$

H denotes the number of units, while I is referring the number of input units. Equation 2.3.22 will be same and we can write equation 2.3.23 similarly in matrix form:

$$y^t = f_o(\sum_{h=1}^{H} W_{ho} h_h^t) \tag{2.3.25}$$

Equation 2.3.25 shows the output calculation of the network. After calculation network output we must compute the error.

Backward pass

Let us assume that we have an objective function J which is continuous and differentiable. We can not apply ordinary backpropagation algorithm to backward pass, since network uses same weights again and again for recursive connections. There are two different algorithms that we can use to train RNNs: real time recurrent learning [10], and backpropagation through time (BPTT) [9, 11]. We used BPTT algorithm to train RNNs.

BPTT algorithm is also recursive application of chain rule, similar to the backpropagation algorithm. The difference is that, for RNNs, the objective function depends on the activation of the hidden layer for both through its influence on the next layer and same hidden layer at the next time step.

$$\delta_h^t = W'(a_h^t)(\sum_{k=0}^{K} \delta_k^t w_{hk} + \sum_{h'=0}^{H} \delta_{h'}^{t+1} w_{h'h}) \tag{2.3.26}$$

Therefore, we can compute the δ for full sequence just repeatedly applying the equation 2.3.26. Finally, since we used the same weights at each time step we should sum all the sequence of δ.

Exploding and vanishing gradients problem

BPTT algorithm comes with the problems. We need to repeatedly apply equation 2.3.26 to compute gradient. This causes problems called exploding and vanishing gradients. Exploding gradients problem introduced in [12], it refers to the exponential increase of the gradient during training. Vanishing gradients problem considered to be opposite of the exploding gradients problem, which occurs when gradients go to zero during training.

2.3.2 Long Short Term Memory (LSTM)

LSTM is a type of RNN architecture which was proposed as a solution for the vanishing and exploding gradient problems of RNNs. A vanilla LSTM network consists of input units, output units, and a hidden unit which has a set of memory blocks, each of which includes one or more memory cells. These blocks are connected to

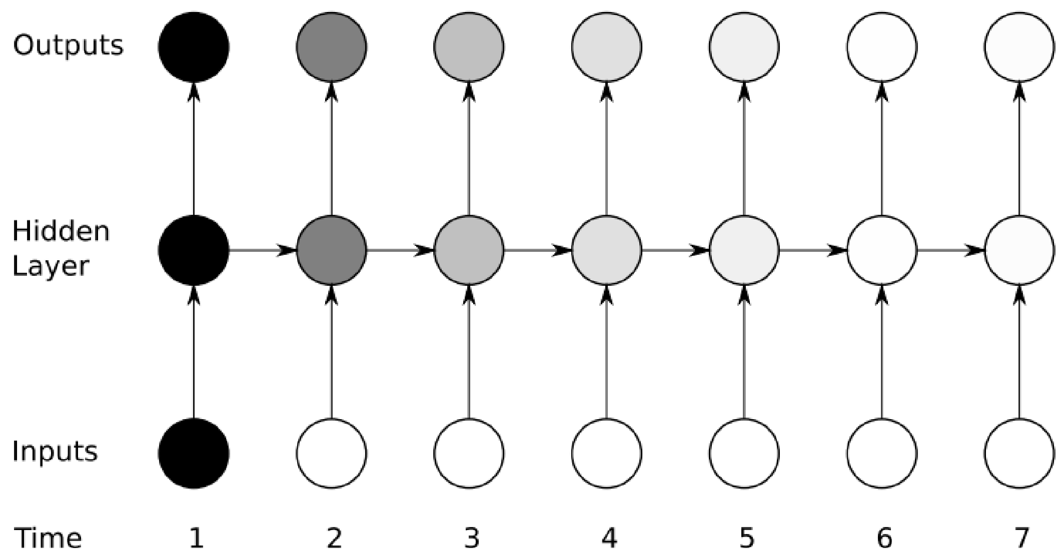


Figure 9: Vanishing gradient problem. Sensitivity to the inputs illustrated with different gray scales, darker shade means greater sensitivity. As it can be seen from the figure, sensitivity decays with the time, when new inputs come to the activations of the hidden layer, and the network *forgets* the earlier inputs. Taken from [15].

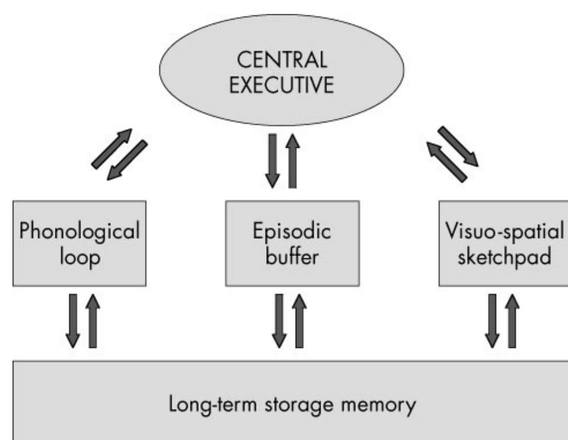


Figure 10: Working Memory Model (WMM) proposed by Baddeley In 2003 [21]. **Central executive** is assumed to be an attentional-controlling system, which is important in skills such as chess playing. **Phonological loop**, stores and rehearses speech based information and is necessary for the acquisition of language vocabulary. **Episodic buffer** acts as a mental workspace for conscious awareness. **Visuospatial sketchpad** manipulates the visual images. Taken from [53]

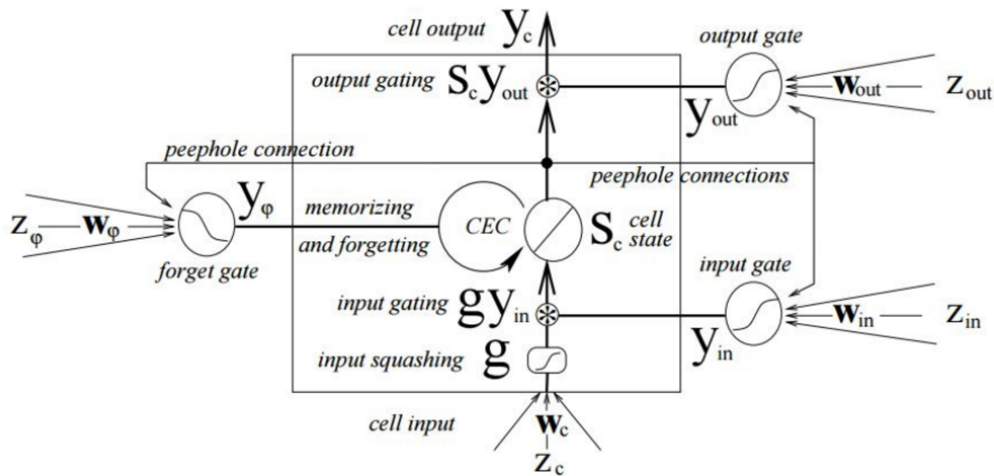


Figure 11: LSTM Memory block with one cell. There are three main gates which get activation from inside and outside the block. These gates control the activation of the cell via dot products. Forget gates control the previous cell state value while input and output gates control the input and output of the cell. There is no activation function within the cell. The line over the cell illustrates the *peephole* (weighted) connections from the cell to the gates. *Input*, *forget*, and *output* gate activation functions are usually sigmoid. The cell input activation functions are usually tanh. Taken from [14].

each other recurrently. The basic unit in the hidden layer of an LSTM network is the memory block, which contains one or more memory cells and a pair of adaptive, multiplicative gating units. The input and output gates multiply the input and the output of the cell while the forget gate multiplies the cells previous state. The gate activation function is usually the logistic sigmoid, so that the gate activations are between 0 and 1. No activation function is applied within the cell, and 'peephole' connections from the cell to the gates. Forget gates [13] and peephole [14] connections are more recent contributions to the structure, and the initial form of LSTM only contains input and output gates. The purpose of the forget gates is resetting memory cells. Peephole connections are introduced to increase LSTM accuracy for tasks that require the accurate measurements.

Biological motivation of LSTM

Gating mechanism in the brain is managed by basal ganglia as Cohen and his team research revealed how gating mechanism works in [22]. As they stated basal ganglia does this while mediating dopamine release. Basal ganglia can be considered as a dynamic gating mechanism at LSTM networks. Another similarity with gating mechanism is that LSTM does not instantly forget information, but the information is forgotten within time. This is quite similar to working memory forgetting strategy, since in human brain release of dopamine does not directly lead to forgetting. Information is lost within time at prefrontal cortex (PFC). Another similarity is that forgetting accomplished by a combination of effects on local inhibitory neurons, similarly at LSTM forgetting occurs through close cells. We want to mention that idea of learning to forget is similar at both structures, with LSTM locally learns when to open or close, this is very similar to gating mechanism of basal ganglia where neurons act locally. Another similarity is the connection structure, as growing evidence suggests that the PFC is organized layer by layer and it has recursive connections. This organization supports cognitive control and the rapid discovery of abstract action rules. LSTM also has recursive connections and can be considered layered. Lastly, we want to mention similarities between information encoding strategies. As we know from the working memory model, information is encoded semantically, similarly, LSTM network also learn internal structure of a dynamic input.

LSTM Cell

Figure 11 illustrates the single LSTM cell. We can formulate LSTM with composition of functions:

input gate

$$i_t = f_i(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \tag{2.3.27}$$

forget gate

$$f_t = f_f(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \tag{2.3.28}$$

cell state

$$c_t = f_t c_{t-1} + tanh(W_{hc}x_t + W_{hc}h_{t-1} + b_c) \tag{2.3.29}$$

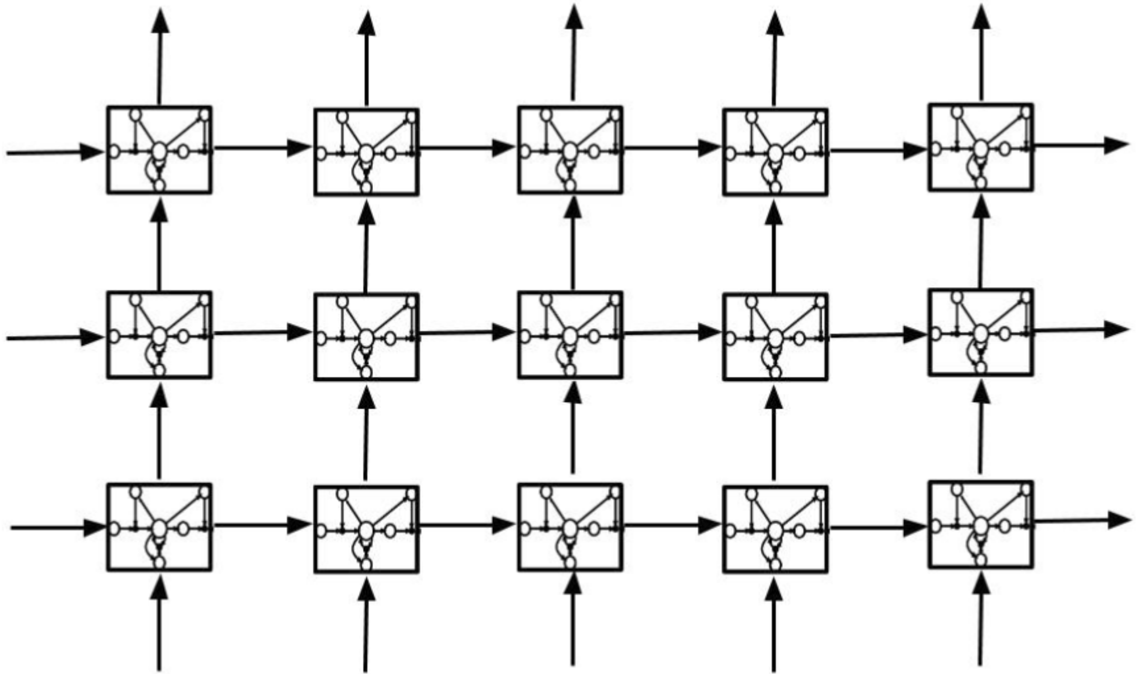


Figure 12: Unrolled Deep LSTM Network for 5 time steps. Three layer LSTM stacked each other, all the LSTM cells have same functions but different weights for each layer.

output gate

$$o_t = f_i(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o) \tag{2.3.30}$$

cell output(prediction)

$$h_t = o_t tanh(c_t) \tag{2.3.31}$$

Function: f_i, f_f, and f_t are the sigmoid functions. Gates: i, f, o and c are respectively the input gate, forget gate, output gate and cell activation vectors, all of which are the same size as the hidden vector h. Elements in each gate vector should take input from the same location of cell vector, this can be done by making weight matrices diagonal for the cell gates.

Forget gates are crucial for learning. Original form of LSTM suffers from automatically resetting itself to a neutral state once a new training sequence starts, even though in theory network should use other cells to reset, but it is not the case. Consequently, Gers(2000) proposed the reset cell. Another important property of forget gates is their capability to adapt changes, as they reset memory blocks once

their contents are out of date and hence useless.

Stack of LSTM cell layers

Deep architectures play an important role in the success of recent models, it enables higher level representations of sequential data. A deep LSTM architectures can be designed by stacking multiple LSTM cell layers. As shown in figure 12, we feed previous layer output sequence to next layer input. Model can use same functions with in LSTM cell, but different weights. Similar to the LeNet model we can use dropout between layers of LSTM.

2.3.3 Gated Recurrent Units (GRU)

RNNs can theoretically capture long-term relations within data, but they are very hard to train to perform this task. GRU modelled to tackle this problem by having more persistent memory by Cho et al. in [23]. Similar to the LSTM, the GRU also control the information flow with gates, GRU does not use memory cell.

GRU cell

Figure 13 illustrates the single GRU cell. We can formulate GRU with a composition of functions:
Memory transferring from previous state controlled by update gate. As it can be seen from the equation 2.3.32, z_t controls how much information will flow from h'_t to h_t. For example, when $z_t = 1$, then h_{t-1} will be copied to h_t. Conversely, when $z_t = 0$, h_t will be initialized newly. Formula of update gate:

$$z_t = f_z(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \tag{2.3.32}$$

The r_t represents the reset gate and it is responsible for measuring the importance of the h_{t-1} for the summarization of h'_t. The reset gate can completely erase the previous hidden state, when $h_{t-1}$is not relevant to the calculation memory. Formula of the reset gate:

$$r_t = f_r(W_{xr}x_t + W_{hr}h_{t-1}b_r) \tag{2.3.33}$$

The h'_t represent the cell state and it is responsible for merging input data x_t with the previous cell state h_{t-1}. Theoretically, this gate merges the input and cell state with using past information from h'. Cell state formula:

$$h'_t = tanh(r_t \odot W_{hh}h_{t-1} + W_{xh'}x_t + b_{h'}) \quad (2.3.34)$$

The h_t represents the hidden state. This state is generated with using update gate, cell state, and previous hidden state. Hidden state formula:

$$h_t = (1 - z_t) \odot h'_t + z_t \odot h_{t-1} \quad (2.3.35)$$

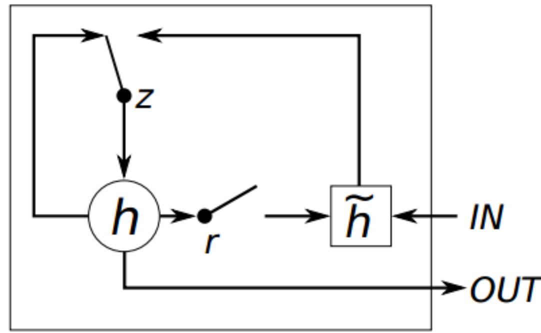


Figure 13: Gated Recurrent Units (GRU). r and z are the reset and the update gates, and h and h' are the activation and the candidate activation. It can be seen the input and forget gates combined to update gate, and also the cell state and hidden state combined to hidden state. Taken from [24]

Chapter 3

CNN+RNN Architectures for Human Pose Estimation

The main aim of this chapter is to give information about the network architectures, training strategies of models, and computational cost of algorithms. Section 3.1 gives information about motivation of combined models, section 3.2 explains network architectures and shows computational costs, and section 3.3 describes how they are trained.

3.1 Motivation

In 2012, AlexNet [25] success on ImageNet competition made revolutionary changes at computer vision research. The idea of learning a representation from very complex data has its basis in CNNs. With the rise of computational power and GPU based implementation of algorithms, the research on that direction accelerated. We can see similar advances in RNNs, especially LSTM [13]. Recurrent models became state-of-art at sequential tasks. Other advances on neural networks such as weight initialisation strategies [27, 28] and new optimisation [4] algorithms helped these models to get the better results. At our implementation, we try to use state-of-art strategies in our networks. Combining CNN and RNN enables us to use the CNN's power to learn compact representations of image data and the RNN's power to learn temporal patterns for the human pose estimation task.

3.2 CNN+RNN Architectures

We found the model architectures layer count, activation function and some other properties with doing grid search on a sub-sampled dataset.

CNN+LSTM

Figure 2 shows the CNN+LSTM architecture for predicting 42 real target data. We found the architecture of the CNN by doing grid search on the task of direct joint estimation. In a nutshell, CNN network consists of 3 convolutional layers, while each convolutional layer is followed by a pooling layer. After these, 2 fully connected layers used. An LSTM network is then attached as a single layer. Only convolutional layers, fully connected layers, and LSTM network contain learnable parameters, while the rest of them are parameter free. Both convolutional layers and fully connected layers consist of a linear transformation followed by a rectified linear unit. In the case of LSTM, we follow section 2.3.2 for implementation. For CNN, we can show the size of the each layer with this notation: width $\times$ height $\times$ depth, where the first two values show the spatial dimensions while depth refers to the channels count (or number of filters). $Conv1(9\times9\times64) - Pool1(2\times2) - Drop - Conv2(3\times3\times128) - Pool2(2\times2) - Conv3(3\times3\times128) - Pool3(2\times2) - FC(1024)$. The total number of the CNN parameters in the above model is about 8M. For LSTM, we use 128 units. The number of parameters in the model roughly are 0.5M.

CNN+GRU

CNN+GRU model is quite similar to the model which was described in the section above. We used the same CNN architecture with the section above mode, but in this model instead of LSTM, we used GRU (See Section 2.3.3). CNN have about 8M parameters and GRU network have roughly 0.2M.

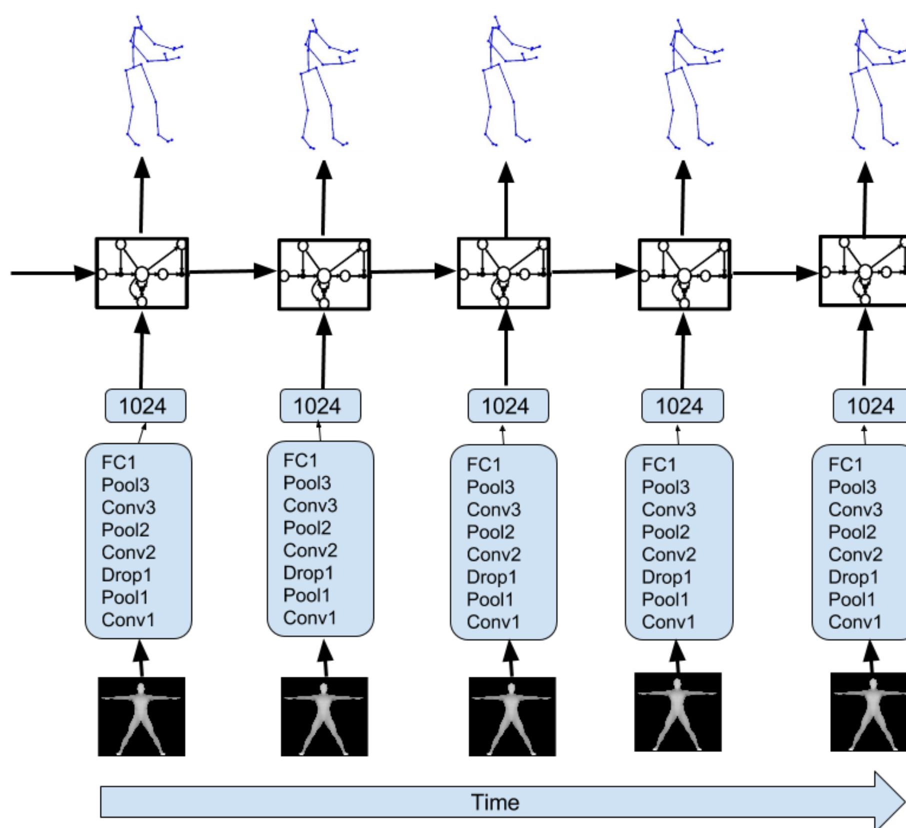


Figure 14: CNN+LSTM architecture illustration. We show the network layers with their name abbreviation. **Conv**: Convolutional layer, **Pool**: Pooling layer, **Drop**: Dropout operation, **FC**: Fully connected layer. Best viewed in color.

3.3 CNN+BiRNN Architectures

3.3.1 Bidirectional RNNs

Before describing bidirectional hybrid models, we will explain the bidirectional RNNs. Bidirectional RNNs rely on a simple idea. It uses recurrent model in backward and forward directions in time. First bidirectional RNNs were introduced in [26]. In this work, LSTM architectures were used for predicting a subcellular localization of eukaryotic proteins.

Bidirectional Vanilla RNN

This part provides the formulation for the bidirectional Vanilla RNN. As is clear from figure 15, the model makes the prediction using bidirectional way. Forward and backward prediction can be combined just simply by taking the average or in with trainable parameters. There are different combining approaches in the literature. We can formulate figure 15 in vector multiplication. Let us assume that we have T time step, then forward calculation will be the forward layer from $t = 1$ to T and backward from $t = T$ to 1, and then we will average both prediction or we can use other weights to merge these results.

Forward hidden state activation computation:

$$\overrightarrow{a}_h^t = \overrightarrow{W}_{ih} x^t + \overrightarrow{W}_{hh} \overrightarrow{h}^{t-1} \tag{3.3.1}$$

Backward hidden state activation computation:

$$\overleftarrow{a}_h^t = \overleftarrow{W}_{ih} x^t + \overleftarrow{W}_{hh} \overleftarrow{h}^{t-1} \tag{3.3.2}$$

Forward hidden state computation:

$$\overrightarrow{h}^t = f_h(\overrightarrow{a}_h^t) \tag{3.3.3}$$

Backward hidden state computation:

$$\overleftarrow{h}^t = f_h(\overleftarrow{a}_h^t) \tag{3.3.4}$$

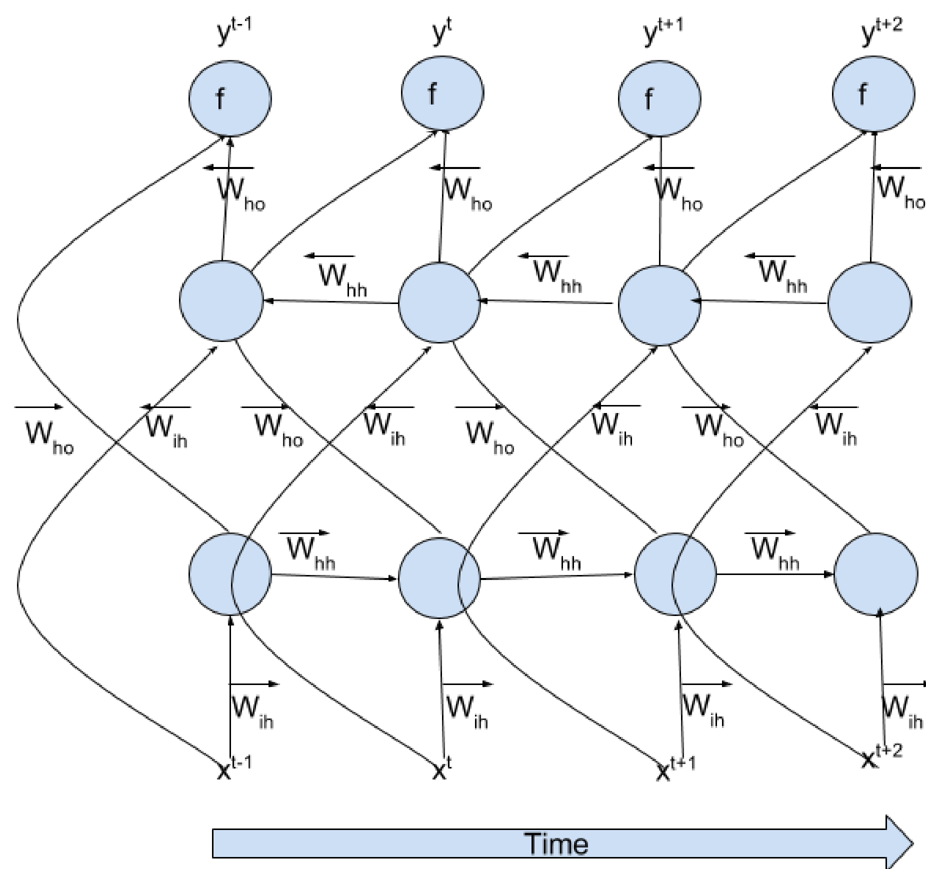


Figure 15: Bidirectional RNN with a single cell. It can be seen that the prediction is done using information from the past and future data. Best viewed in color.

We can combine forward and backward predictions in many different ways, now we are only taking average.

$$y^t = (f_o(\overrightarrow{W}_{ho}\overrightarrow{h}^t) + f_o(\overleftarrow{W}_{ho}\overleftarrow{h}^t))/2 \tag{3.3.5}$$

CNN+BiLSTM

CNN+BiLSTM model is same with CNN+LSTM except that we used bidirectional LSTM, instead of ordinary LSTM. CNN parameters stay the same, but bidirectional LSTM has two times more parameters than ordinary LSTM, meaning roughly 1M.

CNN+DeepBiLSTM

CNN+DeepBiLSTM model has the same CNN architecture as above and has three bidirectional LSTM layers. After first bidirectional LSTM layer, we use a dropout layer during training, which randomly drops features with a probability of 0.5.

3.4 Forward and Backward Pass

In this section, we will show the forward and backward pass equations for our CNN+RNN model. For simplicity, we will show the CNN equations for only the l-th layer, which consists of convolutional, max-pooling, and dropout layers. Since CNN is stacked of these three main layers, it is straightforward to build more advanced architectures similar to the one used in this thesis.

Let us assume that we have an input: $X = \{x_{(1)(1)}, ..., x_{(20)(1)}\}$, which consists of 20 consecutive depth frame. Let $x^{l-1}_{(i)(j)}$ represent the j-th channel of the i-th input ($i \in \{1, ..., 20\}$) for l-th layer of the network. The l-th layer has a $m \times m$ sized filter with depth d.

Forward pass

Applying convolution operation:

$$z^l_{(i)(j)} = \sum_{a=0}^{m-1}\sum_{b=0}^{m-1} w_{ab}x^{l-1}_{(i+a)(j+b)} \tag{3.4.6}$$

Applying activation function:

$$a^l_{(i)(j)} = f(z^l_{(i)(j)}) \tag{3.4.7}$$

Applying max pooling to non-overlapping regions S:

$$\begin{aligned} \hat{s}_k &= \max_{s \in S}(s) \\ S &= \bigcup_k^K s_k = a^l_{(i)(j)} \\ x^l_{ij} &= \bigcup_k^K \hat{s_k} \end{aligned} \tag{3.4.8}$$

Applying Dropout with probability p:

$$\begin{aligned} r^l_j &\sim Bernoulli(p) \\ \hat{x^l_{ij}} &= r^l * x^l_{(i)(j)} \end{aligned} \tag{3.4.9}$$

We repeat the 3.4.6, 3.4.7, and 3.4.8 equations as the number of layers. After several convolutional and pooling layers, we apply the RNN. As can be seen from equation 3.4.10, we need to initialise the hidden state b^0. In this thesis, we initialise b^0 to zero. w_ih, w_hh, and w_ho refer to the input to hidden state weights, hidden state to hidden state weights, and hidden state to output weights respectively. v^l_i denotes the input vector for RNN, which is equal to *flatten*(x^l_{ij}), where *flatten*() denotes the function that reorders an N-D image to a row vector.

$$\begin{aligned} a^t_h &= w_{ih}v^t + w_{hh}b^{t-1} \\ b^t &= f_h(a^t_h) \\ a^t_o &= w_{ho}b^t \\ y^t &= a^t_o \end{aligned} \tag{3.4.10}$$

Backward pass

Firstly, we need to compute L2 loss. $\hat{y^t_d}$ denotes the ground truth pose in D which a vector with $joints \times 3$ elements. Note that we omitted L2 and L1 regularizers for simplicity.

$$E^t = \sum_{d=0}^{D} (\hat{y^t_d} - y^t_d)^2, \quad \hat{y^t_d} = net_w(X) \tag{3.4.11}$$

Our backward pass will update the parameters (w) of the network with the opposite direction of the gradient of the loss function w.r.t network parameters.

$$
\begin{aligned}
\frac{\partial}{\partial w}E(w) &= \frac{\partial}{\partial w}(\hat{y_d^t} - y_d^t)^2 \\
&= 2(\hat{y_d^t} - y_d^t)\frac{\partial}{\partial w}(\hat{y_d^t} - y_d^t) \\
&= 2(\hat{y_d^t} - y_d^t)\frac{\partial}{\partial w}(net_w(x_t)) \\
net_w(x_t) &= \hat{y_d^t}
\end{aligned}
\tag{3.4.12}
$$

Equation 3.4.12 shows the compact form of gradient computing for each parameter. We will show the gradient computing for each unit of RNN with slightly different notations. δ^t defined as the error at the time step $t, t = [1, ..., T]$:

$$\delta^t = f_h'(a_h^t)\left(\delta^t w_{ho} + \delta^{t+1} w_{hh}\right) \tag{3.4.13}$$

Note that we set the $\delta^{T+1} = 0$, since we don't have an error at the beginning of the sequence.

Update rule for w_{hh}:

$$\frac{\partial E(w)}{\partial w_{hh}} = \sum_{t=1}^{T} \frac{\partial E(w)}{\partial a_h}\frac{\partial a_h}{\partial w_{hh}} = \sum_{t=1}^{T} \delta^t b_h^t \tag{3.4.14}$$

Update rule for w_{ih}:

$$\frac{\partial E(w)}{\partial w_{ih}} = \sum_{t=1}^{T} \frac{\partial E(w)}{\partial a_h}\frac{\partial a_h}{\partial w_{ih}} = \sum_{t=1}^{T} \delta^t x_h^t \tag{3.4.15}$$

Update rule for w_{ho}:

$$\frac{\partial E(w)}{\partial w_{ho}} = \sum_{t=1}^{T} \frac{\partial E(w)}{\partial a_h}\frac{\partial a_h}{\partial w_{ho}} = 2\sum_{t=1}^{T}(y^t - \hat{y}^t)b_h^t \tag{3.4.16}$$

Note that we sum the gradient in 3.4.14, 3.4.15, and 3.4.16, since we use the same weights at all the steps.

After recurrent network weights updates, we need to send gradient to the upper layers. As we saw at equation 3.4.10, RNN get the input from CNN with w_{hh}, hence gradient will flow from this unit. Let δ_{ih}^t be value of the error at the input to hidden at time t. We must sum the gradient for all the sequence.

$$\delta_{ih} = \sum_{t=1}^{T} \delta_{ih}^t \tag{3.4.17}$$

Gradient pass from Dropout: We will apply the same mask that we used during forward pass, to the error term. $\bigodot$ denotes the element wise product.

$$\hat{\delta_{ih}}^{l} = r_j^l \bigodot \delta_{ih} \quad (3.4.18)$$

Gradient pass from max-pooling: We will not update the weights respect to non maximum values, because any change in these units will not affect the output.

$$\delta^{l-1} = i^* \bigodot \delta^l \quad (3.4.19)$$

where

$$i_d^* = \begin{cases} 1, & \text{if } x[d] > x[:] \\ 0, & \text{otherwise} \end{cases} \quad (3.4.20)$$

Update rule for w^l: a_i^{l-2} denotes the input to $(l-2)$-th layer of the network.

$$\frac{\partial E(w)}{\partial w^{l-2}} = \sum_{i=0}^{m} a_i^{l-2} * \delta_k^{l-1} \quad (3.4.21)$$

Given these equations, the use of CNN+RNN for pose estimation is straightforward. We just need to feed a frame to hybrid model and do forward and backward pass.

3.5 Computational Complexity

Computational complexity of models are for all models above. In case of, bidirectional models computational complexity will change linearly by the number of parameters, therefore still stays same.

3.6 Output Layer

We used L2 loss (See at 2.1.4) for the objective function and the linear activation function at the last layer. The linear activation function gives the network flexibility to produce every real number. We decided to use L2 loss since its advantages which we mention in section 2.1.2, however, there are many error functions in literature that can be used for this task.

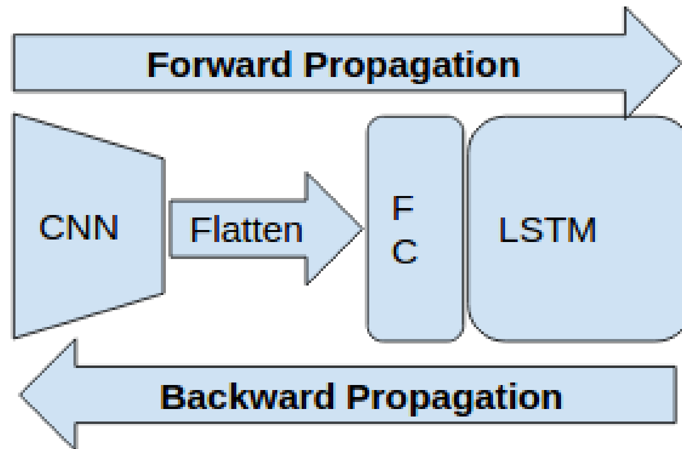


Figure 16: CNN+LSTM workflow visualisation. After LSTM, we compute the cost and send back the gradient. Best viewed in color.

3.7 Training Strategies

Training the combined architectures is the hardest part of the task, since vanishing gradient problem prevents the gradient flow from output layer to upper layers(especially CNN layers). We tested different training strategies to tackle this problem. We observed that training combined models (CNNs + RNNs) with pre-trained CNN yields better results than training from the zero. At joint training similarly, we can use the backpropagation and BPTT algorithm as we describe in section 2.2 and 2.3.2.

Gradient clipping

To tackle LSTM gradient exploding problem during training, we clip the gradient. ADAM [4] algorithm with gradient clipping can be seen at algorithm 1.

Algorithm 2 shows our final training algorithm with modified ADAM updating rule.

Algorithm 1 ADAM with gradient clipping

Require: a : Stepsize

Require: $\beta_1, \beta_2 \in [0, 1)$ Exponential decay rates for the moment estimates. Ideally, we should set 0.1 and 0.001 respectively.

Require: $J(w)$: Objective function.

Require: w : Initialise parameters.

Require: lr : Learning rate.

Require: ϵ : Epsilon value to numerical consistency

Require: m_0, v_0 : Moment vectors. Initial values must be 0.

Require: t : Timestep. Initial value must be 0,

1: **procedure** GetUpdates($J, w, lr, \beta_1, \beta_2, m_0, v_0, t, \epsilon$)

2: $t \leftarrow t + 1$

3: $g_t \leftarrow \nabla_w f_t w_{t-1}$ ▷ Gradient values of parameters.

4: **if** $\|g_t\| \geq threshold$ **then**

5: $g_t \leftarrow \frac{threshold}{(\|g_t\|)} g_t$ ▷ We shouldn't update CNN parameters, since CNN don't have exploding gradient problem.

6: $m_t \leftarrow \beta_1.m_{t-1} + (1 - \beta_1) * g_t$

7: $v_t \leftarrow \beta_2.v_{t-1} + (1 - \beta_2) * g_t$

8: $\hat{m}_t \leftarrow m_t/(1 - \beta_1^t))$

9: $\hat{v}_t \leftarrow v_t/(1 - \beta_2^t))$

10: $w_t \leftarrow w_{t-1} - lr * \hat{m}_t \div \left(\sqrt{\hat{v}_t} + \epsilon\right)$

11: **return** w_t ▷ Theano [25] can handle parameters updating.

Algorithm 2 Gradient Descent

1: **Initialize:**

$w^0_{CNN} \leftarrow load,$

Require: n_in_l, n_out_l : l-th layer weight matrix shape sizes

$s_l \leftarrow \sqrt{\frac{2}{n_in_l + n_out_l}}$

$w^0_{RNN_l} \leftarrow Gauss(s_l)$ ▷ Gaussian distribution with variance s_l

Require: max_epoch : Maximum number of epoch

Require: $batchset$: $load_data$ Batches. Preparing batches

2: $max_batch \leftarrow size(batchset)$ ▷ Number of batches

3: **for** $epoch = 0$ **to** max_epoch **do**

4: **for** $batch_index = 0$ **to** max_batch **do**

5: $batch \leftarrow get_batch(batch_index)$

6: $ForwardPropagate$

7: $w \leftarrow GetUpdates$ ▷ Details of method can be seen at Algorithm 1

3.7.1 Persistent and Non-persistent Training

We consider each human action as a sequence and one sequence can contains long or short range regularities, such as walking, turning around, or drinking water which can span many thousand frames. We tested different strategies to reset the network internal state. We try to balance network ability to capture long range dependencies and error propagation because of state caring during batches. Since we carry network internal state, we did not shuffle the order of the sequences, as it is general practice for neural networks.

3.7.2 Training with Sliding Window

Training data consists of batches and each batch consists of sequences, these sequences are can be loaded differently. For instance, let us assume that we have an action which consists of 2000 frames. We can not feed LSTM with 2000 length sequence, since gradient vanishing and exploding problems. Because of this, we divide each action to sequences with 20 frames and feed LSTM with these 20 frames length sequences. In normal setting, we divide an action from starting zero and split it to

20 frames parts and use data set for full training. For the sliding window training, we split the an action with overlapping.

Chapter 4

Experiments

In this chapter we test our CNNs+RNNs architectures on human pose estimation task. We used depth data, but model can be applied to the RGB data as well.

4.1 Implementation and Hardware

We implemented algorithm with theano [29], which is a Python library that enables programmer to define, optimize, and efficiently compute mathematical expressions including $N \times D$ matrices. Code and pre-trained models parameters will be available at GitHub. Our the biggest model roughly has 11M parameters, hence it is very hard to train on CPU based machines, therefore we used an ubuntu OS machine that has 16 cores, 60gb RAM and NVIDIA Tesla K40 Graphic Card - 1 GPUs - 745 MHz Core - 12 GB GDDR5 SDRAM. Single frame pass took roughly 12ms at testing phase. Regression forest algorithm implemented in C++ and we train this algorithm on a machine that has 8 core and 60gb ram on Windows OS.

4.2 Tuning of Hyperparameters

Models have many hyper parameters, and these parameters optimized to minimize our mean average joins error on a sub set that randomly sampled from the training data. We used with grid and manual search to optimize parameters. Parameters that optimized with grid search include learning rate(between 10^{-2} and 10^{-6}) and

momentum coefficient(between 10^{-1} and 10^{-2}). Parameters that optimized with manual search include the dropout rate, number of CNN layer, number of features at each CNN layers, number of hidden units at RNNs. Regression forest parameters found by manual search, we selected values that close to the given values at original paper [30].

4.3 Data Sets

We used two different datasets to test validity of our approaches, namely Patient MoCap [32] and Human3.6m [32]. In this section, we will briefly explain these datasets and show our results.

4.3.1 Patient MoCap Dataset

This dataset focused on patients in the hospital bed. It includes 10 subjects(5 female, 5 male), each subject performed 10 different actions and each action roughly 1 minute. Subjects are chosen from different body mass index, therefore dataset contains plenty amount of shape variations. During recording, subjects wear daily clothes. Actions are selected specifically, we consider actual sick person actions that can happen in the hospital bed. Actions include *getting out / in the bed, sleeping on a horizontal / elevated bed, eating with / without clutter, using objects, reading, clonic movement and a calibration sequence* . Each action is a single video sequence consisting of multiple depth frames. Depth frames recorded with Kinect V2 cameras, since it has much more higher resolution capability, comparing to the Kinect V1. Human joints locations are recorded with marker based motion capture system which is consist of five calibrated motion capture cameras and fusing software. We recorded 14 3D joint coordinates (*head*, *neck*, *shoulders*, *elbows*, *wrists*, *hips*, *knees* and *ankles*), though system able to capture more. Frame and joint coordinates acquisition times recorded, to maintain synchronization. Total number of video frames for the entire dataset is roughly 180,000. During training, we fed the model with the images which are cropped with a bounding box that around the bed and resized to 120×60. We rendered all images from same camera viewpoint which

is 2 meters away from the bed center with an angle of 70 degrees.

Blanket occlusion

Motion capture systems use marker based tracking to make accurate tracking, hence it is impossible to capture human joint coordinates in case of occlusion. Therefore, we simulated the blanket occlusion for the subjects that laying in the bed. This enables us to train our models on frames that subject under occlusion.

Regression forest (RF)

We compare our models with the regression forest method that introduced by Girshick et al. [30]. As described in [30], regression forests work as follows: At the training phase, tree structure and a set of relative votes to each joint at each leaf are estimated. While tree structure is estimated by using a standard greedy decision tree training algorithm, leaf votes are estimated at four steps:

1 Compute each input pixel offset to all joints.

2 Descend computed joints until leafs at the tree.

3 Cluster offsets with mean shift and select K centroids.

4 Store selected centroids as votes.

At test time inference, the location of the body joints is inferred through a local mode-finding approach based on mean shift.

Evaluation settings

We use 8 subjects (4 female and 4 male) for training, and 2 subjects (1 female and 1 male) for testing. Hyperparameters of the model found, as described in section 4.2. We found out that for CNN best parameters are: learning rate= $3 * 10^2$, batch size: = 50, for RNN learning rate =10^4, sequence length=20. We train the regression forest with a sub-sampled dataset. We used 10.000 images per tree, we did not observe noticeable improvement after training 5 trees with maximum depth of 15. Training regression forest took roughly 2 days. Table 4.1 shows the average joint

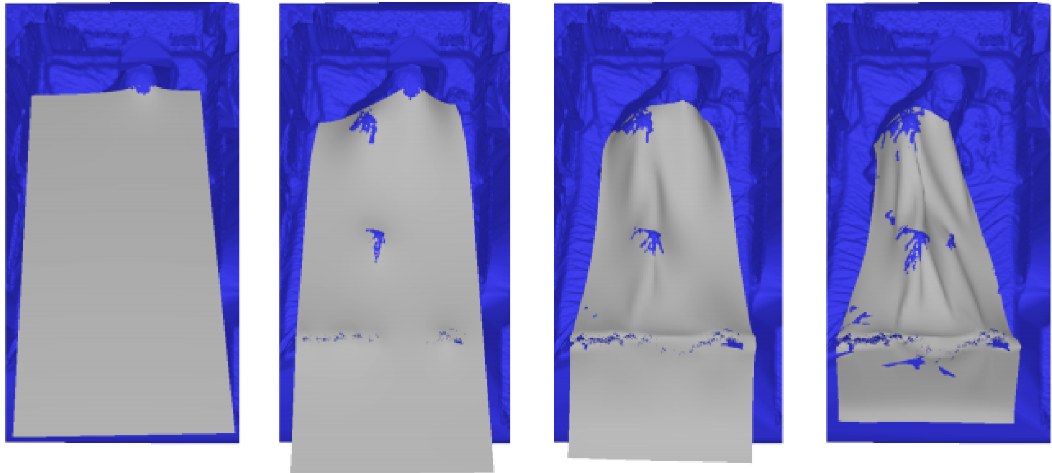


Figure 17: Blanket occluded example frames. We can see that library formed realistic bending and folding of the simulated blanket. Best viewed in color. Taken from [31]

error in CM. We observe that we get the best accuracy with CNN + DeepBiLSTM architecture, while the RF get the worst error. We also compare different training strategies for our best model. Table 4.2 shows the comparison for persistent and non persistent training. As it can be seen, persistent state training improves results. We also compare CNN+RNN joint training with other strategies. Table 4.3 shows the different training strategies and their influence on the results. These results show us that, model perform better with joint training from pre-trained CNN.

Results

Table 4.1 shows that CNN + DeepBiLSTM got the highest accuracy. It also shows that deep models (CNN + DeepBiLSTM and CNN + DeepLSTM), outperformed single layer models with the same CNN architecture, similarly we can see the improvement of bidirectional models. Overall, it can be seen that the test error decrease with deeper architecture and bidirectional models.

As is presented in the table 4.2, training the recurrent model with persistent state improves the results. This implies that our model manages to learn long term dependencies from the data.

Patient MoCap dataset comparisons	
Model Name	Test accuracy
RF	28.0
CNN	12.6
CNN + LSTM	11.36
CNN + GRU	11.46
CNN + BiLSTM	11.12
CNN + DeepLSTM	10.08
CNN + DeepBiLSTM	**10.01**

Table 4.1: Average Euclidean distance in cm between the ground-truth 3D joint locations and those predicted using RF, CNN(no RNN at all), or CNN + RNNs models. CNN + DeepBiLSTM approach yields the most accurate predictions

Patient MoCap dataset comparisons for CNN + DeepBiLSTM	
Training Strategy	Test Accuracy
Non Persistent State	10.06
Persistent State	**10.01**

Table 4.2: CNN + DeepBiLSTM persistent state yields the most accurate results.

Patient MoCap dataset comparisons for CNN + DeepBiLSTM	
Training Strategy	Test accuracy
Joint training from scratch	16.2
Separate training	12.18
Training CNN and LSTM jointly from pretrained CNN	**10.01**

Table 4.3: Jointly training CNN + DeepBiLSTM yields the most accurate results.

Table 4.3 gives information about the influence of joint training. As is observed, joint training from scratch performs the worst, while joint training from pre-trained CNN performing the best. This indicates us, we couldn't train the CNN to learn the features during join training from scratch. One reason can be that we still have

gradient vanishing problem.

The bar chart (see figure 18) illustrates the average per joint error for CNN, CNN+RNN, and RF. We can see that RF fall behind CNN and CNN+RNN model for estimation of limps joint coordinates, on the other hand, CNN+RNN got the best results. Models considerably perform better estimation for the right part of the body than left part of body joints.

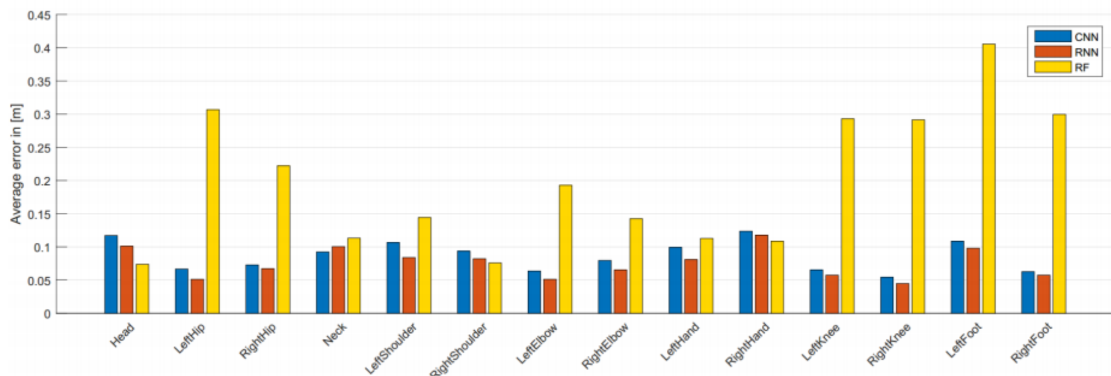


Figure 18: Average per joint error on the on Patient MoCap dataset. Taken from [31]

4.3.2 Human3.6M Dataset

Human3.6M dataset consists of 3.6 million different human poses collected with 4 digital cameras. 11 professional subjects (5 female and 6 male) were selected. Subjects wore daily clothes during performing actions. Subjects body mass index varies from 17 to 29, hence it provides great amount of body shape variability. Subjects were performed 15 actions which are *Directions, Discussion, Eating, Activities while seated, Greeting, Taking photo, Posing, Making purchases, Smoking, Waiting, Walking, Sitting on chair, Talking on the phone, Walking dog, Walking together* . Since subjects were professional, actions were realistic. Depth sequences were acquired with MESA Imaging SR4000 from SwissRanger camera at 25Hz frame(640480 resolution) rate. Pose data acquired with 10 motion capture cameras. 32 joint coordinates are recorded and we use only 17 joints location.

Figure 19: Examples of pose estimation on Patient MoCap dataset. The joints for ground truth are represented by blue lines, while the estimations are represented by yellow lines. Best viewed in color.

Evaluation settings

We used the same partition protocol that used in earlier works [33–35] for training and test splits. The data from 2 subjects (S9, S11) was used for testing and the other 5 subjects (S1, S5, S6, S7, S8) was used for training. We compute the 3D human pose estimation error in terms of average Euclidean distance between the predicted and ground-truth 3D joint positions as at Eqn. 4.3.1. We used only depth frames, however, the earlier works used [33–35].

Mean per joint position error. N denotes the number of joints in the skeleton

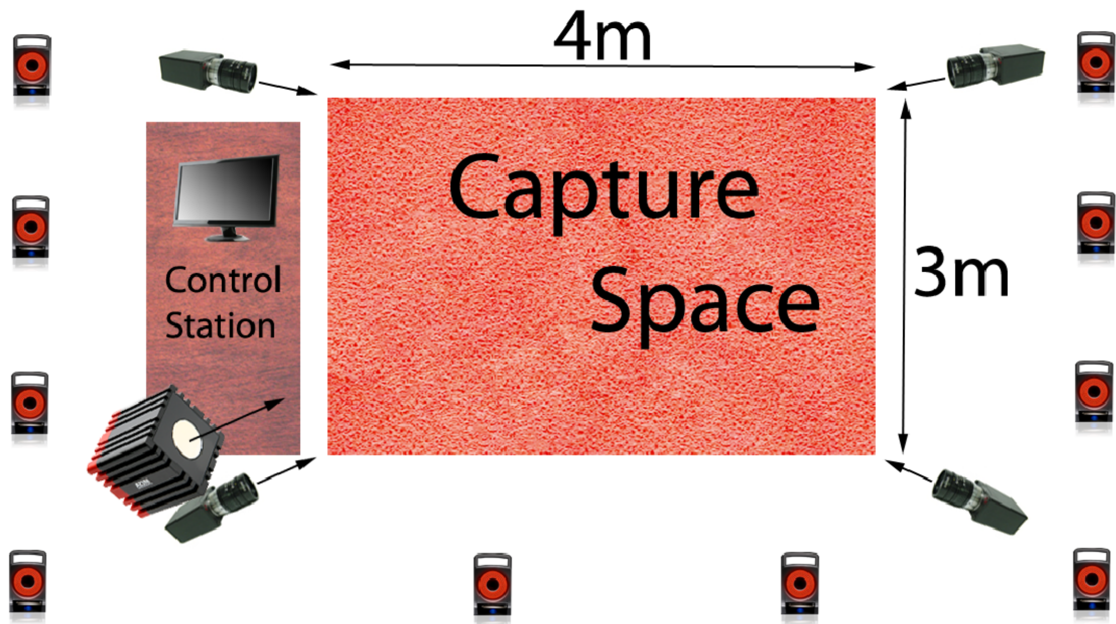


Figure 20: Human3.6M dataset recording laboratory is around $6m \times 5m$, and about $4m \times 3m$ part of the laboratory can be seen in all cameras, therefore subjects performed actions in this part of the laboratory. While 10 motion capture cameras are tracking markers, a time-of-flight sensor (TOF) recorded depth data. Best viewed in color. Taken from [32]

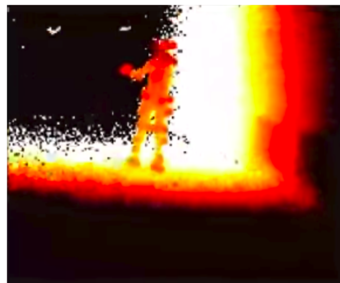


Figure 21: A sample frame from depth recordings. Best viewed in color.

(N=17).

$$Error = \frac{1}{N}\sum_{i=1}^{N} \|f(x)_i - y_i\|_2 \tag{4.3.1}$$

Results

Table 4.4 shows that our model fall behind the other models, however, we should consider that we used only depth frames which are roughly 250K, while the other

model used RGB frames which are 3.6M. If we compare only our models, namely CNN and CNN + DeepBiLSTM, we can see that RNN considerably drops the error for repetitive tasks such as walking, eating etc.

Human3.6M results							
Action	Walking	Discussion	Eating	Taking Photo	Walking Dog	Greeting	All
StructNet-Max [33]	69.97	136.88	96.94	168.68	132.17	124.74	122.9
Auto-Encoder CNN [34]	**65.75**	**129.06**	**91.43**	**162.17**	**130.53**	**121.68**	**116.8**
Deep CNN [35]	77.60	148.79	104.01	189.08	146.59	127.17	132.2
CNN (Ours)	120.2	166.12	145.22	193.7	166.34	167.27	213.9
CNN + DeepBiLSTM (Ours)	100.8	158.23	138.13	188.01	148.54	158.63	190.5

Table 4.4: Average Euclidean distance in mm between the ground-truth 3D joint locations and those predicted by competing methods [33–35](Note that RGB frames are used) and ours(Note that depth frames are used).

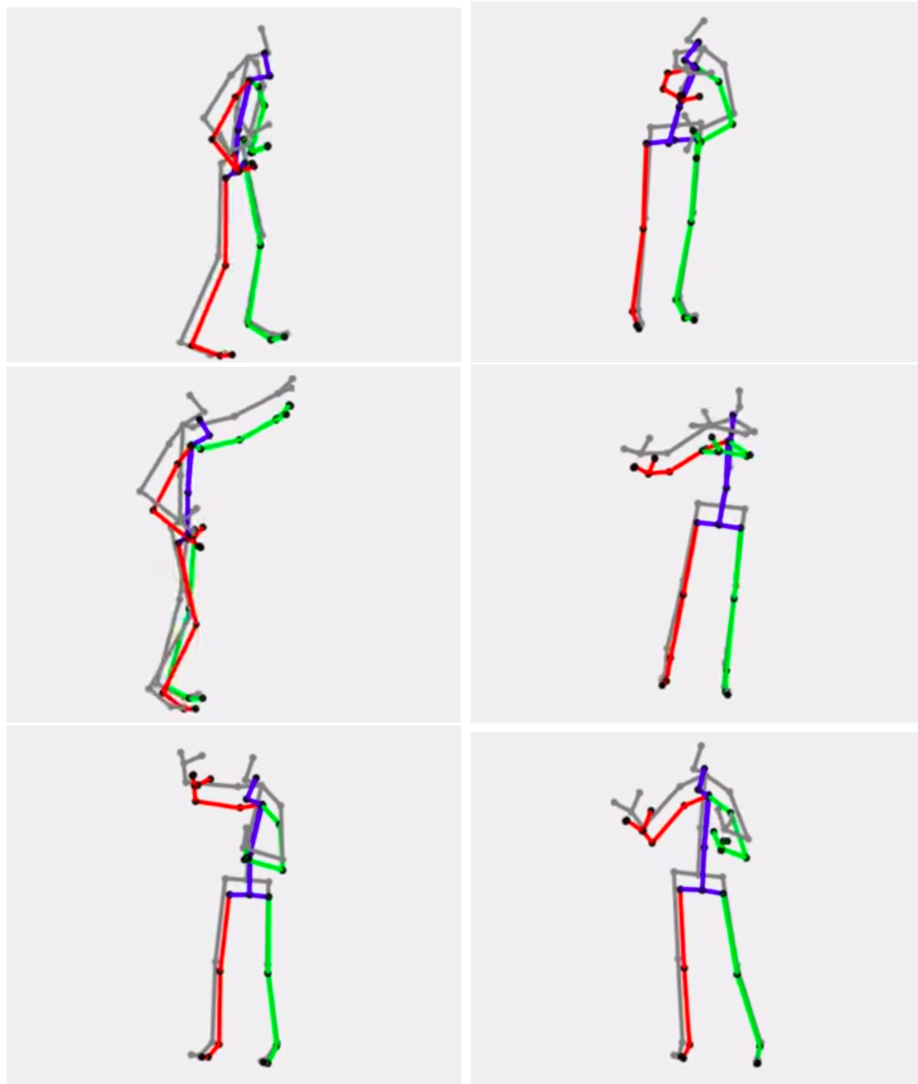


Figure 22: Examples of pose estimation on Human3.6m dataset from depth stream. Estimated joints on the left-side are represented by green lines, estimated joints on the right-side represented by red, and other estimated joints are represented by blue lines. Gray lines represent the ground truth. Best viewed in color.

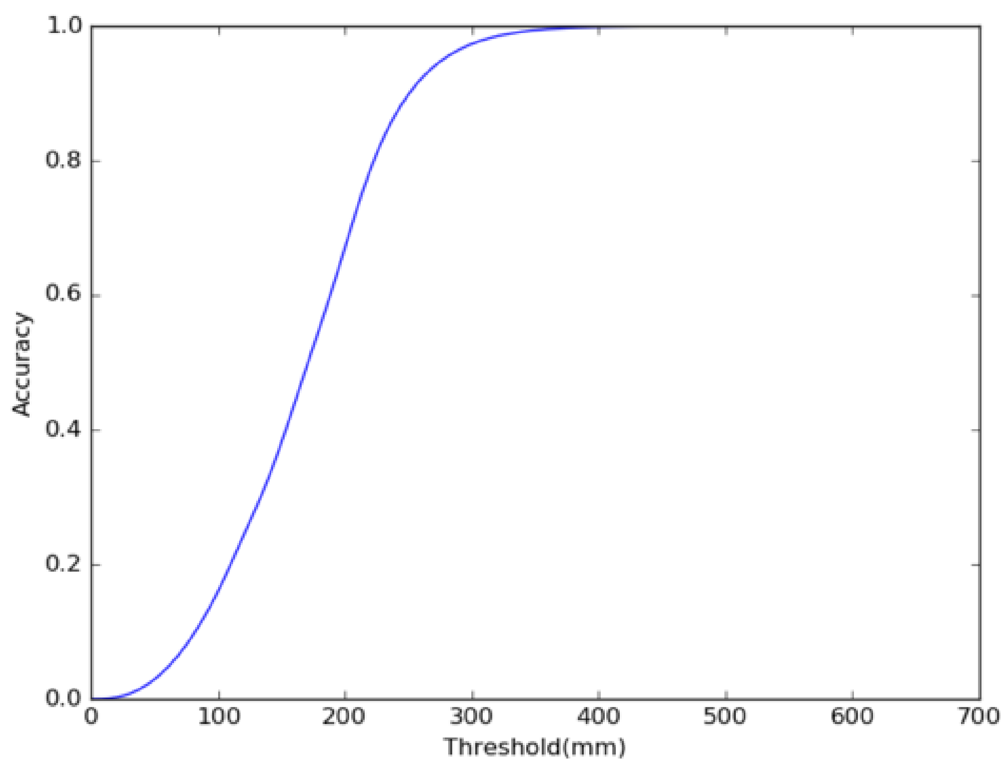


Figure 23: Figure illustrates the CNN + DeepBiLSTM model accuracy change in terms of error threshold (mm). We can see that model hardly gives error more that 300 (mm). This shows us that model capture the human skeleton structure, hence there is integrity at skeleton.

Chapter 5

Conclusions

5.1 Summary

The aim of this thesis is to bring new approaches to 3D human pose estimation from depth stream. In this direction, we proposed a novel CNNs+RNNs architecture and evaluated our approaches by using different datasets. The depth frame datasets are composed of Human3.6M and Patient MoCap databases. This research was inspired by the problems and limitations of the current models, such as failing to predict under occlusion and not considering temporal information. An ideal human pose estimation method need to satisfy the following criteria:

1 It should be robust to occlusion.

2 It should make a reliable estimation in general environment.

3 It should make the estimation while considering human skeleton integrity.

In this thesis we focus on providing solutions for to the first and second problems. The main novelty of the CNNs+RNNs models reside in consolidating the pose estimation alongside bidirectional temporal information. Although it might seems hard to train such a model, using best practices such as ADAM, gradient clipping, and pre-training to train the joint model results in a model that robust to occlusion and environmental changes. In our mode, while CNNs help us to deal with environmental changes, RNNs achieve pose estimation under occlusion with using temporal information. Furthermore, the combination of CNNs with RNNs resulted in a more

realistic estimation, but still this is an open problem. Additionally, the method is able to estimate pose in a great diversity. The lastly and most importantly, we demonstrated that a simple, straightforward and a relatively easy to train method can outperform a mature RF [30] algorithm, so further work will likely lead to even better results. These results suggest that our model will likely perform well on other challenging human pose datasets.

5.2 Future Work

Even though we considered various approaches to human pose estimation, there are many problems to be solved. A major limitation of CNN+RNN model is not considering structural information of the pose. In future work, we plan to incorporate structured prediction to improve the structure preservation and accuracy of the estimated pose. Tompson et al. [49] used CNN+MRF approach to learn structural domain dynamics such as geometric relationships between body joint coordinates. Similar approach can be applied to our model as well. Alex et al. [49] showed that RNNs can perform better at real-valued data through the use of a mixture density output layer, since our prediction also real valued data this approach can improve our results. Finally, we would like to explore the unsupervised training of RNNs. One straightforward strategy would be to use RNNs to estimate future pose for videos.

Bibliography

[1] Nair, Vinod, and Geoffrey E. Hinton. "Rectified linear units improve restricted boltzmann machines.", *Proceedings of the 27th International Conference on Machine Learning (ICML-10).* 2010.

[2] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems.* 2012.

[3] Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." *Journal of Machine Learning Research* 15.1 (2014): 1929-1958.

[4] Kingma, Diederik, and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

[5] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, Gradient-based learning applied to document recognition, *Proceedings of the IEEE* , vol. 86, no. 11, pp. 22782324, 1998.

[6] LeCun, Yann, et al. "Backpropagation applied to handwritten zip code recognition." *Neural computation* 1.4 (1989): 541-551.

[7] Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. *Learning internal representations by error propagation.* No. ICS-8506. CALIFORNIA UNIV SAN DIEGO LA JOLLA INST FOR COGNITIVE SCIENCE, 1985.

[8] Elman, Jeffrey L. "Finding structure in time." *Cognitive science* 14.2 (1990): 179-211.

[9] Werbos, Paul J. "Generalization of backpropagation with application to a recurrent gas market model." *Neural Networks* 1.4 (1988): 339-356.

[10] Robinson, A. J., and Frank Fallside. *The utility driven dynamic error propagation network*. University of Cambridge Department of Engineering, 1987.

[11] Williams, Ronald J., and David Zipser. "Gradient-based learning algorithms for recurrent networks and their computational complexity." *Backpropagation: Theory, architectures and applications* (1995): 433-486.

[12] Bengio, Yoshua, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult." *IEEE transactions on neural networks* 5.2 (1994): 157-166.

[13] Gers, Felix A., Jrgen Schmidhuber, and Fred Cummins. "Learning to forget: Continual prediction with LSTM." *Neural computation* 12.10 (2000): 2451-2471.

[14] Gers, Felix A., Nicol N. Schraudolph, and Jrgen Schmidhuber. "Learning precise timing with LSTM recurrent networks." *Journal of machine learning research* 3.Aug (2002): 115-143.

[15] Graves, Alex. "Neural Networks." *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer Berlin Heidelberg, 2012. 15-35.

[16] Hinton, Geoffrey E., and James A. Anderson. *Parallel models of associative memory: updated edition*. Psychology press, 2014.

[17] Hochreiter, Sepp, and Jrgen Schmidhuber. "Long short-term memory." *Neural computation* 9.8 (1997): 1735-1780.

[18] Frank, Michael J., Bryan Loughry, and Randall C. O'Reilly. "Interactions between frontal cortex and basal ganglia in working memory: a computational model." *Cognitive, Affective, & Behavioral Neuroscience*1.2 (2001): 137-160.

[19] Oberauer, Klaus, and Stephan Lewandowsky. "Modeling working memory: A computational implementation of the Time-Based Resource-Sharing theory." *Psychonomic Bulletin & Review* 18.1 (2011): 10-45.

[20] Hazy, Thomas E., Michael J. Frank, and Randall C. O'Reilly. "Banishing the homunculus: making working memory work." *Neuroscience* 139.1 (2006): 105-118.

[21] Baddeley, Alan. "Working memory: looking back and looking forward." *Nature reviews neuroscience* 4.10 (2003): 829-839.

[22] Baddeley, Alan. "Working memory: looking back and looking forward." *Nature reviews neuroscience* 4.10 (2003): 829-839. Cohen, Jonathan D., Todd S. Braver, and Randall C. O'Reilly. "A computational approach to prefrontal cortex, cognitive control and schizophrenia: recent developments and current challenges." *Philosophical Transactions of the Royal Society of London B: Biological Sciences* 351.1346 (1996): 1515-1527.

[23] Cho, Kyunghyun, et al. "On the properties of neural machine translation: Encoder-decoder approaches." *arXiv preprint arXiv:1409.1259* (2014).

[24] Chung, Junyoung, et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling." *arXiv preprint arXiv:1412.3555* (2014).

[25] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems.* 2012.

[26] Thireou, Trias, and Martin Reczko. "Bidirectional Long Short-Term Memory Networks for predicting the subcellular localization of eukaryotic proteins."*IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)* 4.3 (2007): 441-446.

[27] Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Aistats.* Vol. 9. 2010.

[28] He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." *Proceedings of the IEEE International Conference on Computer Vision.* 2015.

[29] Bergstra, James, et al. "Theano: A CPU and GPU math compiler in Python." *Proc. 9th Python in Science Conf.* 2010.

[30] Girshick, Ross, et al. "Efficient regression of general-activity human poses from depth images." *2011 International Conference on Computer Vision.* IEEE, 2011.

[31] Achilles, Felix, et al. "Patient MoCap: Human Pose Estimation under Blanket Occlusion for Hospital Monitoring Applications."

[32] Ionescu, Catalin, et al. "Human3. 6m: Large scale datasets and predictive methods for 3d human sensing in natural environments." *IEEE transactions on pattern analysis and machine intelligence* 36.7 (2014): 1325-1339.

[33] Li, Sijin, Weichen Zhang, and Antoni B. Chan. "Maximum-margin structured learning with deep networks for 3d human pose estimation." *Proceedings of the IEEE International Conference on Computer Vision.* 2015.

[34] Tekin, Bugra, et al. "Structured Prediction of 3D Human Pose with Deep Neural Networks." *arXiv preprint arXiv:1605.05180* (2016).

[35] Li, Sijin, and Antoni B. Chan. "3d human pose estimation from monocular images with deep convolutional neural network." *Asian Conference on Computer Vision.* Springer International Publishing, 2014.

[36] Buehler, Patrick, et al. "Upper body detection and tracking in extended signing sequences." *International journal of computer vision* 95.2 (2011): 180-197.

[37] Sapp, Benjamin, David Weiss, and Ben Taskar. "Parsing human motion with stretchable models." *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on. IEEE*, 2011.

[38] Toshev, Alexander, and Christian Szegedy. "Deeppose: Human pose estimation via deep neural networks." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2014.

[39] Felzenszwalb, Pedro F., and Daniel P. Huttenlocher. "Pictorial structures for object recognition." *International Journal of Computer Vision* 61.1 (2005): 55-79.

[40] Buehler, Patrick, et al. "Upper body detection and tracking in extended signing sequences." *International journal of computer vision* 95.2 (2011): 180-197.

[41] Eichner, Marcin, et al. "2d articulated human pose estimation and retrieval in (almost) unconstrained still images." *International journal of computer vision* 99.2 (2012): 190-214.

[42] Sapp, Benjamin, Alexander Toshev, and Ben Taskar. "Cascaded models for articulated pose estimation." *European conference on computer vision. Springer Berlin Heidelberg*, 2010.

[43] Tian, Tai-Peng, and Stan Sclaroff. "Fast globally optimal 2d human detection with loopy graph models." *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*. IEEE, 2010.

[44] Belagiannis, Vasileios, et al. "Multiple human pose estimation with temporally consistent 3D pictorial structures." *Workshop at the European Conference on Computer Vision*. Springer International Publishing, 2014.

[45] Jiang, Hao, and David R. Martin. "Global pose estimation using non-tree models." *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*. IEEE, 2008.

[46] Cherian, Anoop, et al. "Mixing body-part sequences for human pose estimation." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2014.

[47] Marinescu, Radu, and Rina Dechter. "AND/OR branch-and-bound search for combinatorial optimization in graphical models." *Artificial Intelligence* 173.16 (2009): 1457-1491.

[48] Ouyang, Wanli, Xiao Chu, and Xiaogang Wang. "Multi-source deep learning for human pose estimation." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2014.

[49] Tompson, Jonathan J., et al. "Joint training of a convolutional network and a graphical model for human pose estimation." *Advances in neural information processing systems.* 2014.

[50] Fragkiadaki, Katerina, et al. "Recurrent network models for human dynamics." *Proceedings of the IEEE International Conference on Computer Vision.* 2015.

[51] Graves, Alex. "Generating sequences with recurrent neural networks." *arXiv preprint arXiv:1308.0850* (2013).

[52] Andriluka, Mykhaylo, Stefan Roth, and Bernt Schiele. "Pictorial structures revisited: People detection and articulated pose estimation." *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on.* IEEE, 2009.

[53] Woodall, G. (2016). Working Memory Model (WMM). [online] Cognitive Psychology. Available at: http://cognitivepsychologygrace.weebly.com/blog/working-memory-model-wmm [Accessed 1 Aug. 2016].