Fine-Grained Semantic Segmentation of Motion Capture Data using Convolutional Neural Networks

by

Noshaba Cheema

Master Thesis

Faculty of Mathematics and Computer Science
Department of Visual Computing
Saarland University

Supervisor
Prof. Dr. Philipp Slusallek

Advisor
Somayeh Hosseini

Reviewers
Prof. Dr. Philipp Slusallek
Prof. Dr. Christian Theobalt

March 29, 2019

UNIVERSITÄT
DES
SAARLANDES

Declaration of Authorship

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Datum/Date: ____________________________

Unterschrift/Signature: ____________________________

SAARLAND UNIVERSITY
Department of Visual Computing

Abstract

Fine-Grained Semantic Segmentation of Motion Capture Data using Convolutional Neural Networks

by Noshaba Cheema

Human motion capture data has been widely used in data-driven character animation. In order to generate realistic, natural-looking motions, most data-driven approaches require considerable efforts of pre-processing, including motion segmentation, annotation, and so on. Existing (semi-) automatic solutions either require hand-crafted features for motion segmentation or do not produce the semantic annotations required for motion synthesis and building large-scale motion databases. In this thesis, an approach for a semi-automatic framework for semantic segmentation of motion capture data based on (semi-) supervised machine learning techniques is developed. The motion capture data is first transformed into a “motion image” to apply common convolutional neural networks for image segmentation. Convolutions over the time domain enable the extraction of temporal information and dilated convolutions are used to enlarge the receptive field exponentially using comparably few layers and parameters. The finally developed dilated temporal fully-convolutional model is compared against state-of-the-art models in action segmentation, as well as a popular network for sequence modeling. The models are further tested on noisy and inaccurate training labels and the developed model is found to be surprisingly robust and self-correcting.

Acknowledgements

There are many people I want to thank for their help and support they offered me throught this thesis.

First of all, I want thank Professor Philipp Slusallek for offering me this thesis, supervising me and being a great mentor overall. Due to his encouraging and optimistic nature, it has been always fun and a delight to work with him. I also want to thank him for supporting me in every other possible way outside of the thesis work. Despite his busy schedule he tried to squeeze me in wherever he could. :)

Additionally, I want to thank Professor Christian Theobalt who agreed to be my second reviewer. Unfortunately, we did not have the chance to work much together but I hope that this will change in the future.

The person who has been with me through the most and who deserves the most recognition is Somayeh Hosseini. Thank you for spending countless nights with me running experiments, writing papers and staying up until the last minute on each deadline we tried to reach ♡ Same goes for Janis Sprenger, Erik Herrmann and Han Du. I appreciate that each one of you stayed up with me until the last minute of a deadline. Further acknowledgement goes to Klaus Fischer for proof-reading and commenting our papers, and Arsène Pérard-Gayot for going over and commenting our rebuttal.

Then I want to thank my friends, specifically Somayeh Hosseini, Itrat Rubab, Soumen Ganguly, Sören Klinger, Ho Vhinh Thinh, Vedika Agarwal, and Nathaniel Borges for being there for me whenever I was in Saarbrücken and did not have any place to sleep after coming back from Finland.

Last but not least, I want to thank Tobias Schemken for being there for me in every possible way and bearing me whenever I was stressed out ♡

Content

Dedicated to my friends and family ♡

Chapter 1

Introduction

The increasing demand of computer generated natural human motion for many industrial, as well as, entertainment applications, coupled with the decreasing cost of motion capture production, are a driving force for more sophisticated tools to process and analyze motion capture data. However, many of these applications, like motion retargeting or gesture recognition, require small, semantically similar pieces of motion data in order to function properly [11]. Typical motion capture sessions, however, produce long streams of motion. Naturally, a preprocessing step is needed to break the motion stream up into short pieces, which are semantically meaningful for the application. So far, this process is often done manually, which is laborious and time consuming.

To reduce time and effort, an automatic solution is needed. Such an algorithm would segment motion capture data into a collection of smaller portions of the original data. To be able to synthesize new motions of the same motion type or to find a specific motion in a data-base one needs semantic labels for them. Thus, high-level descriptive labels are needed for such tasks. In kinematic segmentation, segment labels are akin to a low-level kinematic description of a motion, like *high velocity sagittal motion* [12]. This makes the labels harder to interpret and less intuitive to work with. Whereas semantic segmentation uses high-level descriptions, such as *carrying a package*, to label the resulting segments, which accounts for more intuitiveness. Hence, one of the main goals of automatic motion segmentation should be to produce semantically sophisticated labels of motion segments on a general set of actions.

1.1 Importance of Automatic Segmentation

The importance of effective automatic motion segmentation is defined by the multitude of applications that require semantically annotated segments and by the numerous advantages to automating the process of producing them [11].

1.1.1 Segmentation Applications

Generally speaking, applications that make use of motion capture segmentation can be grouped into two distinct categories [11] - motion synthesis and motion analysis. The first category includes animation applications, such as video games or movies, that depend on individual segments from motion capture data to drive the animation (e.g. [3, 42, 49, 50, 67, 80, 92, 93, 104, 113, 126]). Many data-driven motion synthesis techniques require fine-grained segmented motion primitives (e.g. *left step* or *right step* instead of *walking*) as a preprocessing step. A recent overview of motion synthesis approaches can by found by Guo et al. [42]. In most of these approaches the motion recordings need to be split in structurally and semantically similar segments for further statistical and/or graph-based modeling. Interactive applications, such as video games, control their animation often through motion graphs [21, 68, 92], where nodes are motion primitives and the transitions represent potential segment sequencing. Such a graph is often created by using semantic segments of motion capture data with smooth transitions to all other potential segments.

Many motion analysis tools on the other hand, focus more on tasks like action classification, where data is labeled for content (e.g. [3, 27, 95, 96, 142]). Human-computer interaction strives to increase the efficiency with which humans interface with computers. One approach towards this objective is to make human-to-computer interaction similar to human-to-human interaction [11]. To achieve this, computers need the ability to recognize multiple styles of human communication, including verbal, facial and gestural. Such gestural analysis classifiers [52, 116] often need to analyze discrete pieces of data, which have to be obtained via action segmentation.

An off-line classification application that involves motion segmentation is motion database indexing. The database storing the data becomes increasingly large and difficult to access as more motion data is stored. A solution is to label segments of motion through automated classification and when accessing the database, to perform searches on the labels [11]. However, this still needs small, logical pieces of motion data in order to perform classification.

1.1.2 Automatic Segmentation Advantages

For real-time applications of motion segmentation - not only segmentation is required, but *automated* segmentation. Manually segmenting motion primitives from motion capture sequences is tedious work. Bouchard et al. [11, 12] have shown different people produce different segmentation results, when given the same motion data. Additionally, the same person will often give differing segmentations of identical motion capture data. The median of each segment boundary was computed and then the distance of each boundary from the median was computed and graphed as a histogram. The result is a standard deviation of 15.4 frames, or about $\frac{1}{2}$ seconds. This is due to high inter-annotator and intra-annotator disagreements in manual motion capture segmentation. Automatic segmentation however, computes the same result for the same sequence, making it a more deterministic approach.

1.2 Outline and Approaches

1.2.1 Outline

Conventional action and motion primitive segmentation methods either rely on low-level hand-crafted features, such as difference between foot and floor [50, 92], or on unsupervised machine learning techniques, such as clustering [141] or Principle Component Analysis [127]. While the latter do not require hand-crafted features and thus are able to generalize to unseen motion types, they lack control and semantics over the segmentation output. For example, Zhou et al. [141] are able to segment walking motions into *left step* and *right step* when two clusters are used (Fig. 1.1 *top*) with their Hierarchical Cluster Analysis (HACA) method. However, when using four clusters their unsupervised segmentation method divides a single step, essentially segmenting a walking motion into *left forward*, *left stop*, *right forward*, *right stop* (Fig. 1.1 *middle*). While this is a valid segmentation result, it might not be what a "human expert" had hoped for when using four clusters. Many graph-based motion synthesis methods [92, 93] in fact benefit more from clusters similar to *begin left step*, *right step*, *left step* and *end right step* (Fig. 1.1 *bottom*), as they better model transitions between a standing and a walking action.

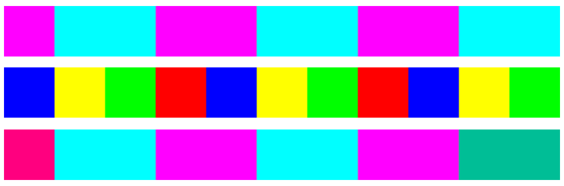


FIGURE 1.1: *Top:* HACA result on a walking sequence when two clusters are used. The method segments the motion into *left step* (🟪) and *right step* (🟦). *Middle:* HACA result on a walking sequence when four clusters are used. The method segments the motion into *right forward* (🟨), *right stop* (🟩), *left stop* (🟥) and *left forward* (🟦). *Bottom:* A desired output for [92] with *begin left step* (🟥) in the beginning and *end right step* (🟩) in the end.

In this work we therefore focus on the development of a (semi-) supervised method using convolutional neural networks to achieve such a segmentation, which then can be used for motion synthesis or classification applications as previously described. A detailed description of each chapter is included below.

Before describing our work, in Chapter 2 we give an overview of motion segmentation methods and highlight their advantages and disadvantages. In particular, we divide existing work into three major segmentation categories - *kinematic-based*, *data-analysis-based* and *recognition-based*, in addition to manual segmentation. Since the main contribution of this work is a convolutional neural network for motion segmentation, a machine learning and neural network overview is given in Chapter 3.

Chapter 4 describes the data structure of the motion capture file format of the data-set, as well as the subset of joints and sequences, and the annotations that are used for the experiments. It furthermore introduces a technique to generate a *motion image* out of a motion capture sequence.

The first technical contribution is introduced in Chapter 5. Here a simple two-layer fully-connected model similar to Holden et al. [50] is described and evaluated on the given data-set. This model takes a single frame as its input and classifies it.

Chapter 6 improves on the previous model by changing the fully-connected layers to convolutional layers - making it a fully-convolutional model. Due to "convolutionizing" the dense layers, the model is now able to take inputs of various lengths. In particular, it is now able to take a whole sequence and classify each frame of it simultaneously. Furthermore, it does not need any data balancing anymore, which is the case with the model in Chapter 5. Another benefit of using convolutions instead of dense layers is that the receptive field size of the model can be arbitrarily changed without having to change the input data. I.e. the overhead of extracting a specific window out of the sequence is taken away by the convolutions of the deep learning framework. The size of such a window can simply be changed by changing the widths of the convolutions.

Finally, Chapter 7 introduces various techniques to further increase the receptive field size of such convolutional models by increasing the kernel size, number of layers and dilating convolutions. The developed model is then compared against two state-of-the-art action segmentation methods using temporal convolutional neural networks [74, 75] and a popular sequence modeling method using a bi-directional Long-Short-Term Memory network [119]. We found our model to achieve state-of-the-art performance and to be surprisingly robust and accurate compared to the benchmark models.

Last but not least, in Chapter 8 many of the practical implications and limitations of deep learning-based segmentation techniques and the introduced models are addressed. The chapter further highlights future directions for fine-grained motion modeling and describes potential pathways for this work at scale.

1.2.2 Challenges

When it comes to supervised data-driven segmentation, the quality of the segmentation result relies on the quality of the segmentation of the training data given to the model. It requires a sophisticated classifier which is robust enough to take poorly classified data as an input. As noted by Bouchard [11]: “[...] the problem is a paradox, where recognition requires segmentation and segmentation requires recognition”.

Another challenging aspect is that two different motion sequences, which are semantically identical for human observers (e.g. *reaching for an object*), can look very different from each other from a geometric point of view. E.g. in one sequence the object is picked up from the ground and in another sequence it is picked up from a shelve above the character. In both cases an object is reached for but in one sequence the character is bending down and in another it is lifting its arm. These variations have to be taken into account when building such a classifier.

Furthermore, graph-based motion synthesis models, e.g. *Motion-Graphs++* [92] require fine-grained segmentations, e.g. differentiating between a beginning left step (*begin left step*, from a standing position) and a *left step* which is done while walking. In the former case, the step starts from both feet next to each other and in the latter case one foot is in front of the other. This is important to synthesize new actions in which the character transitions from standing to walking. However when the classifier is just given a couple of frames, these two classes can be difficult to tell apart from each other.

1.3 Thesis Statement

We posit that dilated temporal convolutional filters can efficiently capture complex time-series patterns for the use of fine-grained semantic segmentation of motion capture data. Additionally, we believe such (semi-)supervised segmentation methods can produce complex semantic labels in contrast to commonly used unsupervised methods in motion capture segmentation.

1.4 Contributions

We introduce a Dilated Termporal Fully-Convolutional (DT-FCN) Network architecture which outperforms previous state-of-the-art TCN-based models, as well as an RNN-model in fine-grained motion segmentation tasks. Additionaly, the model is surpringly robust when trained on noisy training labels.

Chapter 7 is mostly based on:

- *Fine-Grained Semantic Segmentation of Motion Capture Data using Dilated Temporal Fully-Convolutional Networks*
  Noshaba Cheema*, Somayeh Hosseini*, Janis Sprenger, Erik Herrmann, Han Du, Klaus Fischer, Philipp Slusallek
  Eurographics 2019 - Short Papers.

- *Dilated Temporal Fully-Convolutional Network for Semantic Segmentation of Motion Capture Data*
  Noshaba Cheema*, Somayeh Hosseini*, Janis Sprenger, Erik Herrmann, Han Du, Klaus Fischer, Philipp Slusallek
  ACM SIGGRAPH/Eurographics Symposium on Computer Animation - Posters 2018. **Best Poster Award**.

* denotes equal contribution

Chapter 2

Related Work

2.1 Motion Recognition Stratification

As outlined by Lea [75], naming conventions in literature concerning action recognition are often abused. For consistency, we therefore use a variation of their terminology throughout this work in regards to action and motion primitive recognition and classification. The terminology is as follows:

Motion Primitive: An *elementary action* part of a bigger action like walking. The elementary actions or motion primitives here would be *left step* or *right step.*

Action: A sequence of motion primitives, which as a whole are considered as one action, e.g. *walking*, *picking*, *carrying* etc. All these can be decomposed into further elementary actions, i.e. motion primitives.

Motion Primitive/Action Classification (trimmed): Given a video or sensor sequence that only consists of one motion primitive/action, classify that motion/action.

Motion Primitive/Action Classification (untrimmed): Given a video or sensor sequence that only consists of one dominant motion primitive/action and some background class, classify the dominant motion primitive/action.

Motion Primitive/Action Localization: Given a sequence of data that consists of one dominant motion primitive/action and some background class, classify the dominant motion primitive/action and determine the starting and ending frame.

Motion Primitive/Action Detection: Given a sequence of data with many motion primitives/actions, detect all instances of every primitive/action and the corresponding starting and stopping frame for each. Typically there are "background" segments between motions/actions which are not detected.

Semantic Motion Primitive/Action Segmentation: Given a sequence of data with many motion primitives/actions, densely label all time steps with a motion primitive/action class. This may include an explicit background class which must be detected.

Motion Primitive/Action Recognition: This is used as an umbrella term for classification, segmentation and other aforementioned tasks, in which the input is some data sequence and the output is some information related to the presence, timings, or locations of motion primitives/actions.

Motion: A general umbrella term for 3D skeletal motion.

Most of previous literature [22, 57, 71, 117, 130] focuses on action classification, rather than action segmentation - let alone segmenting these actions into further elementary primitives. Nevertheless, being able to accurately segment motions and grouping them into their corresponding equivalence classes is crucial for many motion synthesis and analysis applications [14, 21, 92, 102]. In this thesis we focus on segmenting motion primitives from long motion capture sequences. The advantage of this fine-grained segmentation method is that the granularity can easily be adjusted to a "coarser" action segmentation later on [73], without losing the applicability to certain graph-based motion synthesis [92] methods.

2.2 Motion and Action Recognition and Segmentation

This section presents an overview of main segmentation methods used in academia and industry. Similar to Bouchard [11], we divide existing algorithms into three major types of motion segmentation methods, in addition to manual segmentation. The other three being *(i) kinematic-based segmentation*, *(ii) data-analysis-based segmentation*, and *(iii) recognition-based segmentation.* To our knowledge there is not extensive work done in motion primitive segmentation, therefore most of the mentioned related work focuses on action segmentation. We include work based on 3D motion capture data, as well as, 2D video-based data.

2.2.1 Manual Segmentation

The first and most naïve type of segmentation method is based on manually segmenting motion data. While this approach provides an output, which is "natural" to a human annotator, it can vastly vary from annotator to annotator. There is not only high inter-annotator disagreement but also intra-annotator disagreement, as described in Section 1.1.2. In addition, the work is very time consuming and tedious, making it unfit for real-time applications.

2.2.2 Kinematic-based Segmentation

Kinematic segmentation is based on low-level kinematic features such as Euclidean coordinates or rotational velocity of joints. Commonly, the segmentation is accomplished by comparing handcrafted, low-level kinematic features over a time series to determine the patterns that correlate to segment boundaries. Segments for new motion data are then found by computing the same time series features and searching for the correlated segment boundary patterns. These methods especially require manually setting many parameters, e.g. to determine how similar two segments are to one another.

Min et al. [92] and Holden et al. [49] use coordinate-based contact point measures to classify motion primitives in motion sequences, e.g. distance between foot and floor. A similar position-based method is used by Jenkins and Matarić [55, 56] using an arm's centroid to segment arm motions. A threshold for a maximum centroid distance between the starting segment frame and another frame is calculated. The ending frame is classified whenever it is above the set threshold. The next segment starts where the previous one ended. Another simple and frequently used kinematic feature is velocity. Many motion or action segments begin at relatively low speed, accelerate in mid-action and finally reduce their speed again. This feature is utilized in various algorithms. Fod et al. [27] implement two different approaches for motion segmentation of an arm using angular velocity in four different degrees of freedom. The first method chooses segments such that at least two degrees of freedom have zero velocity within 3 ms of both the beginning and end of every segment. The second method tracks the sum of the four degree of freedom's angular velocities and determines segment boundaries when the value drops below an experiment-based threshold. Müller et al. [96] use boolean features per joint that indicate the spatial relationship between a joint and a plane formed by other joints and a joint angle's or velocity's relationship to a threshold. A sudden change in direction also correlates with reaching a destination and the end of a motion, which is why curvature - a measure of change in direction - is another popular kinematic feature used for segmentation. Zhao and Badler [138] calculate segment boundaries when hand linear acceleration zero-crosses and curvature is above some threshold.

Kinematic-based segmentation allows for a simple on-line motion or action segmentation method using hand-crafted features. However, these types of hand-crafted features are difficult and time-consuming to craft for many different motions and actions, as one needs to craft other features for different motion or action classes. For example, the distance between foot and floor cannot be utilized for recognizing hand-waving motions.

2.2.3 Data-Analysis-based Segmentation

The next class of segmentation methods draws from data analysis methods, such as time series analysis [44], Principle Component Analysis (PCA) [54, 103], Gaussian distribution models [84] and other *unsupervised* machine learning algorithms, such as clustering [88]. These methods are data-driven and are able to learn or correlate distinct high-level features from the data without having to label it and automatically segment motion based on finding patterns in these features. Hence, in comparison to kinematic-based segmentation, they produce more sophisticated results. While they are computationally less efficient than kinematic segmentation methods, their main advantage is that they do not require manually labeled data and hand-crafted features to achieve reasonable results. A major disadvantage of such methods is the lack of control of what is supposed to be learned for a specific motion segmentation task.

One of the earliest works on motion data segmentation by Barbič et al. [8] introduces three techniques in data-analysis-based motion segmentation. The first one being based on PCA, which decomposes human motion into distinct actions by detecting sudden changes in intrinsic dimensionality. The second method, using Probabilistic PCA (PPCA) [127], segments by detecting changes in the distance of fitting a small segment of motion capture data to a Gaussian distribution model of the segment's preceding frames. The idea is that two different behaviors will belong to separate Gaussian distributions. Another commonly used method from unsupervised machine learning is clustering. The third method introduced by Barbič et al. [8] uses a Gaussian Mixture Model (GMM) [23] to cluster motion segments. Lee and Elgammal [79] use k-means [54] to estimate the GMM that corresponds to a cluster. A more sophisticated approach by Zhou et al. [140, 141] uses the k-means algorithm with a kernel extension [114] for temporal clustering of data to segment actions. Each cluster denotes an action class. Vögele et al. [70, 131] improve on this by using a neighborhood graph to further segment these motions into primitive partitions. Another method introduced by Müller et al. [95] makes use of so-called Motion Templates (MTs) that consist of a set of predefined dynamically time-warped binary pose features. Classification and segmentation of motion data is then done via template matching. Many variants of motion features [71, 87, 132] are proposed to provide a more informative representation for motion data.

Data-analysis-based methods are able to learn more complex features from data and hence are able to produce more sophisticated segmentation results compared to their kinematic counterparts. A consequence of that is that they are slower than kinematic segmentation methods. Nevertheless, they are able to generalize to unseen motions without having to come up with new hand-crafted features or any supervisory signal. Due to that however, they are not able to make any insights on the content or semantics of the motion data and therefore do not necessarily produce semantic segmentations.

2.2.4 Recognition-based Segmentation

The fourth class of segmentation techniques is based on *supervised* machine learning techniques. Supervised learning techniques have the advantage that the created segments can be as complex as manual segmentation, due to the manual labeling of the training data. Typically, a large collection of motion capture data is manually segmented and labeled and used to train a classifier. While these methods are very difficult to implement for general or unknown motion types and the initial labeling still requires human work, they provide high flexibility and control when it comes to motion segmentation. The users themselves can decide, whether they want to segment different actions from a motion capture stream or single motion primitives, such as left or right step. This flexibility and the ability to have a controlled output, makes these techniques compelling for segmentation tasks for motion synthesis.

Kahol et al. [60] use a naïve Bayesian classifier [81, 89] to derive choreographer segmentation profiles from dance motion sequences. Bouchard and Badler [12, 13] have proposed to our knowledge the first neural network based segmentation method on motion capture data. They perform Laban Movement Analysis (LMA) [98] to obtain more meaningful features than simple kinematic features. These features are then used to train 121 perceptrons for segment boundary detection. The use of 121 networks is to minimize segment boundary inconsistencies. The results are then summarized over the networks and the local maxima determine the segment boundaries.

In recent years, deep learning methods have gained popularity over such more shallow networks. An approach by Wu et al. [132] uses a Hidden Markow Model to model the sequential dynamics of motion capture data and adopt a multi-layer neural network to train the output probabilities from hidden action states to observed pose sequences. Their approach does not require hand crafted features and can achieve competitive results on the MSR Action3D [82] and MSRC12 [29] motion data sets for high-level action recognition and segmentation. Recurrent neural network (RNN) methods [48, 111] have exclusively been developed for sequence modeling tasks like the temporal segmentation of motion or video data. The idea behind these types of networks is that the data is sequentially dependent, i.e. current and future samples are based on previous samples. Thus, the order of the data matters and is not disregarded in these networks. Well-regarded books [40] and courses [99] on sequential learning focus almost exclusively on RNNs. Fragkiadaki et al. [31] have proposed a three layered encoder-decoder RNN-model to do human pose estimation in video data based on motion data for training and synthesis. To overcome the vanishing gradient problem [47] in RNNs, Long Short-Term Memory (LSTM) networks, a special kind of RNN which uses different gates to remember or forget important and unimportant features, are used instead for "memorizing" very long sequences. Du et al. [22] propose a multi-layered LSTM network for action classification that feeds skeleton data into five sub networks for different body parts at the initial layer and hierarchically fuses the output in higher layers. With this they are able to achieve state-of-the-art results in action classification on the MSR Action3D [82], Berkeley MHAD [100] and HDM05 [97] datasets. As convincing as these applications may appear in terms of recognition and the complexity of features recurrent models are able to learn, one of their major disadvantages is that training is extremely difficult and slow [7, 47, 48]. Due to their sequential nature, one training step has to be finished before another can start as the next training step is based on the output of the previous step. This makes parallelization during training tough in comparison to other neural network models [32, 69, 110] which are easy to parallelize. Furthermore, the vanishing and exploding gradient problem [47] in recurrent architectures makes training on very long sequences a challenging task.

Recent studies [7, 74, 75] suggest that certain architectures of Convolutional Neural Networks (CNNs) [19, 36, 61, 129], which are traditionally used for image classification, can reach state-of-the-art results in typical sequence modeling tasks such as word translation or audio synthesis. Bai et al. [7] have conducted an extensive study about Temporal Convolutional Networks (TCNs) outperforming different types of RNNs, including LSTM and Gated Recurrent Units in various sequence modeling tasks. They further study long-range information propagation in convolutional and recurrent networks, and show that the "infinite memory" advantage of LSTMs is largely absent in practice. They show that TCNs exhibit longer memory than recurrent architectures with the same capacity. TCNs have also been used for action segmentation. Lea et al. [74, 106] use an Encoder-Decoder TCN and a variation of WaveNet [129] to segment different actions from the 50 Salads [124], MERL Shopping [119] and GTEA [25] video datasets. CNNs are orders of magnitude faster to train than RNNs, due to their "embarrassingly parallel" [69] nature. In order to apply CNNs, which have been successfully applied on image classification tasks [69, 118, 125], also on action recognition tasks, Laraba et al. [71] propose a conversion of skeleton sequences into RGB images, where the color of a pixel represents the normalized position of a joint at a certain frame. Similarly, Ke et al. [64] process skeleton sequences into a set of images, each representing the entire skeleton sequence with a focus on one particular spatial relationship, and apply a convolutional Multi-Task Learning Network architecture for segmentation. The idea of representing motion sequences as an image and then use image processing techniques has also been used by Aristidou et al. [5]. The same authors have recently extended their work to use an unsupervised deep learning approach for motion retrieval and segmentation [4]. We explore a similar strategy to Laraba et al. [71] and incorporate results of Bai et al. [7] on fine-grain motion segmentation tasks with motion capture data.

While recognition-based segmentation is hard to generalize for unseen motion or action classes, it is able to generate segmentations that can be as "natural" as human annotations [11]. They are also flexible as they can be trained to detect a variety of segmentation types. For example, Kahol et al. [60] demonstrate that it is possible to create classifiers that emulate the segmentation results of different people. Lea et al. [106] show how a classifier can be trained to detect various levels of granularity within action segmentations. They have a major advantage over kinematic-based segmentation as they do not require hand-crafted features. While data-analysis-based methods are able to generalize to unseen motion classes, they do not infer any semantic information from the data. Recognition-based methods overcome this problem by giving semantically labeled data at training time to the respective model. When dealing with data-driven motion synthesis [92] which requires a database of small structurally similar pieces of motion, the semantic meaning of these pieces is crucial for creating such a database.

Chapter 3

Neural Networks

Artificial neural networks (ANNs) have been the subject in the field of Computational Neuroscience for some time. Inspired by biological neuronal systems, primarily the human brain, ANNs are networks of multiple complex layers of non-linear transformations to process information [109]. Typically used for analysis tasks - such as classification or detection [69, 118] - their increasing popularity, due to the availability of huge amounts of data and accelerated hardware, has driven their development to various other tasks, such as data synthesis [49, 129], automated captioning [63] or computational creativity [35].

This chapter introduces the general architecture of artificial neural networks and how they are trained and used. It further gives an overview of optimization methods, layers and popular adaptations (such as LSTM [48] and CNN [32, 69, 118]) that are mentioned throughout this thesis and explains them in detail.

3.1 Supervised Learning vs. Unsupervised Learning

In machine learning, *supervised learning* algorithms *learn* a function h which maps an input x to an output y based on example input-output pairs called *training samples* [112]. I.e. each of these training samples x is *labeled* with the desired output value y of the function h. The output value is also called the *supervisory signal.* A supervised learning algorithm analyzes the training data and adjusts its parameters such that it produces an inferred function, which can then be used for mapping new examples. In an optimal scenario the algorithm is able to correctly determine the labels of unseen examples, which requires the learning algorithm to reasonably generalize from the training data to new instances. Artificial neural networks belong into this category.

In contrast to supervised learning, in *unsupervised learning* there is no supervisory signal given. I.e. no labeled output value y for a training sample x. In this scenario the algorithm has to find a pattern or structure from the unlabeled data itself. However, in this thesis we exclusively focus on supervised learning algorithms due to their ability to give a semantic meaning to each classified sample.

3.2 Basic Architecture

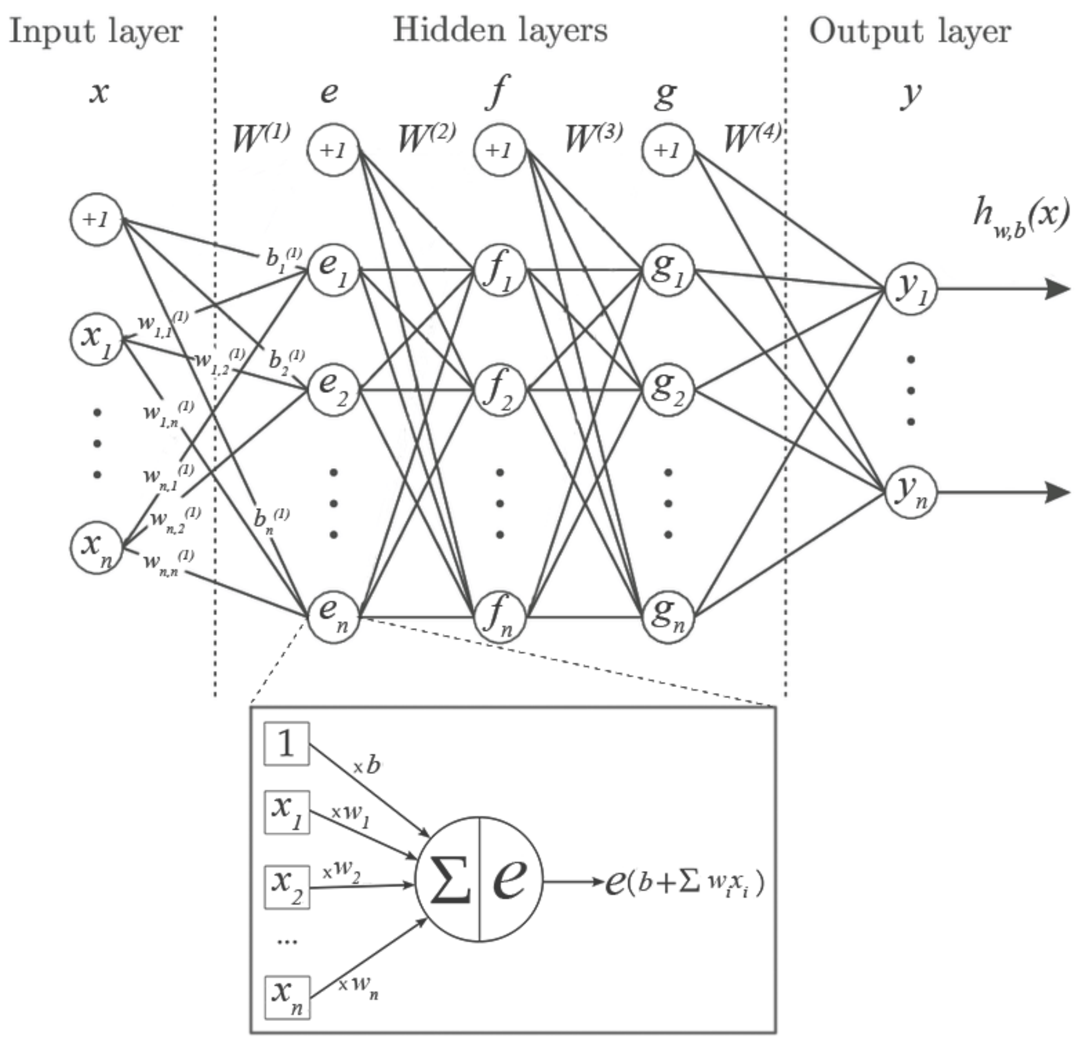


FIGURE 3.1: *Top:* A neural network architecture with three hidden layers. *Bottom:* A neuron that takes a weighted sum as input and applies an activation function e to it.

ANNs are based on collections of units or nodes called *artificial neurons* which loosely model the neurons in biological brains. Such an artificial neuron is depicted in Fig. 3.1 *bottom.* Similar to a biological neuron the artificial neuron takes one or more inputs which are biologically akin to the postsynaptic potentials [9], computes a weighted sum of them and applies an *activation function* to it. The activation function models an action potential in neurobiological terms.

In common ANN implementations such neural units are arranged in layers. Fig. 3.1 *top* illustrates an ANN with five such layers. Layers between the ANN input layer and the ANN output layer are called *hidden layers.* These neurons have an activation function which receives a weighted sum as input (Fig. 3.1 *bottom*) and outputs the "activated" *feature map* to the next layer, which can either be another hidden layer or an output layer. A feature map is the output of a neural layer. As the name suggests, it describes various *features* of the original input as the neural layer essentially filters it. Such features can be for example, edges, orientations or other various transformations of the input [120].

Forward Propagation

The process of forwarding these features to the next layer and computing more such features until we reach the output layer is called *forward propagation* or *testing.* In the network illustrated above, this is formally done by computing:

$$h_{w,b}(x) = y\left(\vec{b}^{(4)} + W^{(4)}g\left(\vec{b}^{(3)} + W^{(3)}f\left(\vec{b}^{(2)} + W^{(2)}e\left(\vec{b}^{(1)} + W^{(1)}\vec{x}\right)\right)\right)\right) \tag{3.1}$$

The "edges" which connect these neural nodes are called *weights* or *parameters.* There are two different types of parameters. The ones that weigh the input elements (in Fig. 3.1 *bottom* depicted as w_i) and the ones that shift the layer input to the left or right which are called *biases.* Such a bias is denoted as b in the illustration above. The other parameters w_i are usually just referred to as weights.

3.2.1 Training

These parameters are not predefined but have to be learned through some task - usually a classification task. E.g. the network has to classify whether some given input is a dog or a cat. Usually, such labels are represented by a vector using *one-hot encoding.* I.e. every vector value is 0, except one, where it is 1:

$$y = \begin{bmatrix} 0 & ... & 0 & 1 & 0 & ... & 0 \end{bmatrix} \tag{3.2}$$

The index i of $y_i = 1$ then determines the object class. The classification is done via the forward propagation. However, since the weights have not been learned yet and are initially set to some random values, our initial classification result might be wrong. To adjust the parameters in our network such that it performs well on the given task we need to tune the parameters by *training* the network. For that we need labeled training data. Assume we have an input x and the true label of x is $\hat{y}$. When x is given to the network some transformations are applied to x such that we get a prediction

$$y = h_{w,b}(x) \tag{3.3}$$

from the network, where $h_{w,b}$ describes the non-linear function of the network applied to x based on the parameters w and b. To see how wrong our prediction y is from the true value $\hat{y}$, we calculate some error or *loss* function $\mathcal{L}$ between these two which incorporates the difference between y and $\hat{y}$. Our goal is to minimize this error as best as possible with our network. To do so, we need to calculate the minimum of $\mathcal{L}$ by adjusting the network's parameters:

$$\min_{w,b} \mathcal{L} = \frac{1}{n}\sum_i (y_i - \hat{y}_i) \tag{3.4}$$

However, since ANNs usually have a high-dimensional parameter space, computing the minimum analytically takes a lot of computing time [34, 40]. Hence, we need to compute the minimum numerically. This is usually done with optimization methods - the most common one being *gradient descent*.

Gradient Descent

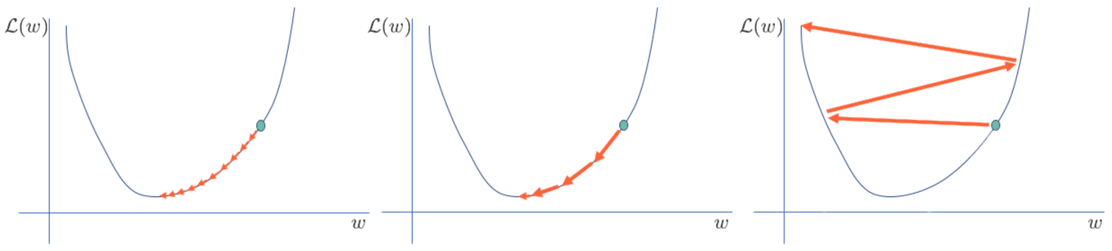


FIGURE 3.2: Gradient descent method. *Left:* The learning rate is too low, which requires many update steps before reaching an optimum. *Middle:* An optimal learning reaches the optimum with fewer update steps. *Right:* The learning rate is too large which leads to a divergent behaviour. *Image source (edited):* [59]

Fig. 3.2 illustrates how an error function could look like in an ANN. In reality the error function has a high-dimensional parameter space and is usually non-convex. The basic idea of numerically updating the parameters of the network is to compute the steepest descent such that we reach the minimum at some point. The optimization algorithm for this method is called *gradient descent*. The steepness of the function is determined by its slope which in turn is defined by the function's derivative or *gradient*, when dealing with multivariate functions. Fig. 3.2 shows how gradient descent operates. It is an iterative algorithm which updates the parameters such that the error is lower with each weight update step. The magnitude and direction of the weight update is computed by taking a step in the opposite direction of the cost gradient. For parameters w and b at time step t magnitude and direction are given by:

$$\Delta w_t := -\alpha \frac{\partial \mathcal{L}}{\partial w}_t \tag{3.5}$$

$$\Delta b_t := -\alpha \frac{\partial \mathcal{L}}{\partial b}_t \tag{3.6}$$

where α is the *learning rate*, i.e. how big the steps should be that we take, when updating the parameters (orange arrows in Fig. 3.2). If the step size is too little, the training takes more time as we need more steps to reach the minimum (Fig. 3.2 *left*). However, if our step size is too big, we might overshoot our minimum and the algorithm starts to diverge (Fig. 3.2 *right*). Oftentimes, trial and error is needed to obtain the optimal learning rate. More advanced algorithms [66, 85] even have an adaptive learning rate.

The final update rules for step $t + 1$ is then given by:

$$w_{t+1} := w_t + \Delta w_t \tag{3.7}$$

$$b_{t+1} := b_t + \Delta w_t \tag{3.8}$$

Gradient descent is guaranteed to converge to the global minimum for convex functions and find some local minimum for non-convex functions if the step size is small enough.

Back Propagation

To compute the gradients $\frac{\partial \mathcal{L}}{\partial w}$ and $\frac{\partial \mathcal{L}}{\partial b}$ of $\mathcal{L}$ with respect to the network parameters w and b we make use of another algorithm called *back propagation.* As the name suggests, the error is *propagated backwards* through the network using the chain rule. Let $e(x)$, $f(x)$, $g(x)$, $h(x)$ and $y(x)$ with parameters w and b be some functions applied to an input x as seen in Fig. 3.1.

$$h_{w,b}(x) := y(g(f(e(x)))) \tag{3.9}$$

To calculate $h_{w,b}(x)$ with respect to w or b we need to apply the chain rule:

$$\frac{\partial h}{\partial w} = \frac{\partial y}{\partial g} \cdot \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial e} \cdot \frac{\partial e}{\partial w} \tag{3.10}$$

$$\frac{\partial h}{\partial b} = \frac{\partial y}{\partial g} \cdot \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial e} \cdot \frac{\partial e}{\partial b} \tag{3.11}$$

Back propagation in the network above works in a similar fashion. Here $W^{(1-4)}$ are the weight tensors of layers 1 to 4, and $b^{(1-4)}$ their respective biases. e, f, g, y are the outputs of layers 2 to 5. The back propagation process can then be summed up the following way:

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} := \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial y}{\partial g} \cdot \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial e} \cdot \frac{\partial e}{\partial W^{(1)}} \tag{3.12}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(1)}} := \frac{\partial \mathcal{L}}{y} \cdot \frac{\partial y}{\partial g} \cdot \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial e} \cdot \frac{\partial e}{\partial b^{(1)}} \tag{3.13}$$

$$\frac{\partial \mathcal{L}}{\partial W^{(2)}} := \frac{\partial \mathcal{L}}{y} \cdot \frac{\partial y}{\partial g} \cdot \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial W^{(2)}} \tag{3.14}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(2)}} := \frac{\partial \mathcal{L}}{y} \cdot \frac{\partial y}{\partial g} \cdot \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial b^{(2)}} \tag{3.15}$$

$$\frac{\partial \mathcal{L}}{\partial W^{(3)}} := \frac{\partial \mathcal{L}}{y} \cdot \frac{\partial y}{\partial g} \cdot \frac{\partial g}{\partial W^{(3)}} \tag{3.16}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(3)}} := \frac{\partial \mathcal{L}}{y} \cdot \frac{\partial y}{\partial g} \cdot \frac{\partial g}{\partial b^{(3)}} \tag{3.17}$$

$$\frac{\partial \mathcal{L}}{\partial W^{(4)}} := \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial y}{\partial W^{(4)}} \tag{3.18}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(4)}} := \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial y}{\partial b^{(4)}} \tag{3.19}$$

3.3 Improved Optimization Methods

The idea behind gradient descent is simple and easy to implement. However, over the time there have been made many improvements to the algorithm. Here we briefly explain the Stochastic Gradient Descent (SGD) and Adaptive Moment Estimation (Adam) algorithms. SGD is a well-known advancement of the traditional gradient descent algorithm described in Section 3.3.1.

Nevertheless, it is not as efficient as other modern optimization algorithms which also make use of an adaptive learning rate or take other factors like previous steps into account. Adam is a modern optimization algorithm which has been used in a variety of contexts in machine learning [6, 53, 94, 105, 133]. Its ability to generalize to different machine learning problems has contributed to its popularity in this field. We make use of this optimization algorithm in the conducted experiments in this thesis.

3.3.1 Stochastic Gradient Descent (SGD)

In traditional gradient descent we compute the cost gradient based on the whole training set, hence it also sometimes called *batch gradient descent* (BGD). However, taking the complete training set into account can take a lot of computing time and power if we deal with a lot training samples. Using gradient descent can thus be very costly, since we are taking a single step at a time for one pass over the whole training set. The more training samples we have, the more time it takes the algorithm to converge to a minimum. As described in 3.2.1, the update rule for batch gradient descent looks like this:

$$w_{t+1} := w_t - \alpha \frac{\partial \mathcal{L}}{\partial w_t} \tag{3.20}$$

Here the gradient $\frac{\partial \mathcal{L}}{\partial w}_t$ is calculated by using the whole training set. This performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. In SGD however, the gradients are only computed based on a single training sample x_i and y_i at a time t:

$$w_{t+1} := w_t - \alpha \frac{\partial \mathcal{L}_i}{\partial w_t} \tag{3.21}$$

Therefore, it is usually much faster and can also be used for online learning [10]. The term *stochastic* comes from the fact that using a single sample is a “stochastic approximation” of the whole dataset. Due to this, SGD tends to fluctuate more than BGD. On one hand, this can be of advantage since this fluctuation enables the algorithm to jump to new and potentially better local minima. On the other hand, this may make the algorithm overshoot a global minimum.

Mini-Batch Gradient Descent (MB-GD)

Mini-batch gradient descent (MB-GD) combines the best of SGD and BGD, as it does not use the whole dataset but a subset or *mini-batch* of it. A mini-batch contains n training samples which are used to update the weights in one step:

$$w_{t+1} := w_t - \alpha \frac{\partial \mathcal{L}_{i,i+n}}{\partial w_t} \tag{3.22}$$

This leads to a more stable convergence compared to SGD but it is still faster than BGD.

3.3.2 Adaptive Moment Estimation (Adam)

Adam is a variation of MB-GD which includes the *momentum method* in the update rule and techniques to adapt the learning rate accordingly during training.

Momentum Method

First introduced by Rumelhart et al. [111], this weight update method does not only take the current gradient update Δw_t but also the previous Δw_{t-1} into account:

$$\Delta w_t := -\alpha_t \frac{\partial \mathcal{L}}{\partial w_t} \tag{3.23}$$

$$w_{t+1} := w_t + \Delta w_t + \beta_t \Delta w_{t-1} \tag{3.24}$$

Δw_{t-1} decays by $0 < \beta_t < 1$. The intuition behind the momentum method stems from the *momentum* in physics. The weight tensor w, akin to a particle traveling through parameter space [111], incurs acceleration from the gradient (akin to a force). Unlike classical gradient descent algorithms, the momentum method keeps traveling in the same direction, damping oscillations at high curvature. The intuition behind it is similar to a ball traveling through hills, speeding up whenever the descent is steep and keep going in that direction. This method speeds up finding a local minimum.

Adam Algorithm

Algorithm 1 *Adam* [66]. $\odot$ indicates element-wise multiplication and $\oslash$ an element-wise division respectively. All operations on tensors are element-wise. Default settings suggested in [66] are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. β_1^t and β_2^t are β_1 and β_2 to the power t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $\mathcal{L}(\theta)$: Stochastic loss function $\mathcal{L}$ with parameters $\theta$
Require: $\theta_0$: Initial parameter tensor
  $m_0 \leftarrow 0$ (Initialize $1^{st}$ moment vector)
  $v_0 \leftarrow 0$ (Initialize $2^{st}$ moment vector)
  $t \leftarrow 0$ (Initialize time step)
  while $\theta_t$ not converged do
      $t \leftarrow t + 1$
      $g_t \leftarrow \frac{\partial \mathcal{L}_t}{\partial \theta_{t-1}}$ (Get gradients w.r.t. stochastic loss at time step $t$)
      $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
      $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t \odot g_t$ (Updated biased second raw moment estimate)
      $\alpha_t \leftarrow \alpha \cdot \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t}$ (Adapt learning rate for bias corrected moment estimate)
      $\theta_t \leftarrow \theta_{t-1} - \alpha_t \cdot m_t \oslash (\sqrt{v_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

As shown in Alg. 1, Adam takes two moments into account. The first order moment m_t (mean of current and previous gradient) which we have described in the *moment method* and a second order moment v_t which is the uncentered variance of the gradient. Note how m_t and v_t are initialized with zeros, which leads to moment estimates that are biased towards zero, especially in the initial time steps. This is counteracted with a learning rate that is adapted with the factor $\frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t}$. The factor indicates the discrepancies between m_t and g_t, and v_t and $g_t \odot g_t$, respectively [66].

The adaptive learning rate with the use of the momentum method, make this algorithm a powerful tool for many optimization problems in machine learning. It is able to generalize over a variety of problems. In practice Adam is usually used with a mini-batch method.

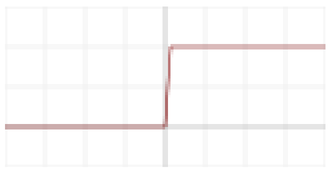
3.4 Layer Catalogue

Artificial neural networks can be seen as hierarchical models of subsequent building blocks which are stacked on top of each other. These building blocks are called layers. In this section we describe a variety of layers used in common ANN architectures, as well as in this thesis.

3.4.1 Activation Layers

Activation layers are element-wise operations on a given input which determine how much of the input are propagated to the next layer. Such activation functions are an abstraction of action potentials in biologically inspired neural network architectures. There are different types of activation functions with different properties. In this section we explain them in more detail. x in the activation functions below is a real-valued tensor and the input to the neuron with the corresponding function.

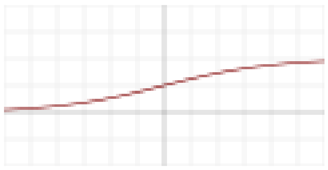
Binary Step



$$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases} \qquad f'(x) = \begin{cases} 0 & x \neq 0 \\ undef & x = 0 \end{cases} \tag{3.25}$$

The binary step function, also called the Heaviside step function, is a function between 0 and 1. Its value becomes 0 for every negative input and 1 for every positive input. Its derivative is either zero or undefined. This makes this type of activation function not very practical in ANN architectures, since its gradient vanishes instantly.

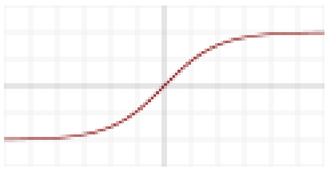
Sigmoid



$$f(x) = \sigma(x) = \frac{1}{1+e^{-x}} \qquad f'(x) = f(x)(1-f(x)) \tag{3.26}$$

The Sigmoid activation function, also called Soft-step function, is a differentiable version of the binary step function. Instead of a sudden jump from 0 to 1 it increases gradually. Its derivative values $f'(x)$ are between 0 and 0.25.

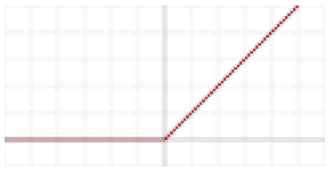
Tanh



$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad f'(x) = 1 - f(x)^2 \tag{3.27}$$

Tanh is similar to the Sigmoid function, except its values span from -1 to 1. This gives us the advantage that it approximates the identity function near the origin. Furthermore, its derivative values are between 0 and 1 which means that it is better against the vanishing gradient problem than the Sigmoid function.

ReLU



$$f(x) = \max(0,x) \qquad f'(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases} \tag{3.28}$$

First introduced by Hahnloser et al. [43], the rectified linear unit (ReLU) outputs values between 0 and ∞. Despite its non-differentiability at $x = 0$ Glorot et al. [39] demonstrate a better trainability of deep networks with such an activation due to its sparsity for values $x \leq 0$. Sigmoid or Tanh functions on the other hand, are more likely to produce non-zero values resulting in dense representations. The ReLU activation function also counteracts the vanishing gradient problem [47] due to its gradient having a constant value of 1 when $x > 0$. This constant gradient of ReLU also results in faster learning.

Softmax

$$f_i(x) = \frac{e^{x_i}}{\sum_j e^{x_j}} \qquad \frac{\partial f_i(x)}{\partial x_j} = f_i(x)(\delta_{ij} - f_j(x)) \tag{3.29}$$

The Softmax activation function is a generalization of the Sigmoid activation function when more than just two labels are used. It outputs a K-dimensional vector where each entry is between 0 and 1 and they all sum up to 1. Due to this, this type of activation function is usually used in the very last layer to determine to how much percent some input x belongs to a class k. δ_{ij} in its derivative is the Kronecker delta with $\delta_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$.

3.4.2 Weight Layers

The input of activation layers is usually a weighted sum. The weights are arranged in such weight layers.

Dense Layer

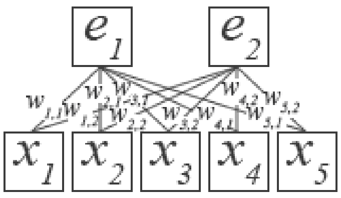


FIGURE 3.3: Every input node x_i is connected with every output node e_i in a fully-connected or dense layer.

This type of layer is also called fully-connected layer or dot-product layer. As the name suggests it connects every input node with every output node with a parameter. The ANN depicted in Fig. 3.1 uses this type of layer throughout. The layer treats the input as a simple vector [1, 58] and produces an output in the form of a single vector. This is done by computing the dot-product between the input nodes and the parameter matrix W that this layer defines. I.e. when we have some input vector $\vec{x} \in \mathbb{R}^N$ and some output vector $\vec{y} \in \mathbb{R}^M$, the dense layer's weight parameters have to be of the form $W \in \mathbb{R}^{N \times M}$ to connect its input and output nodes. E.g. in Fig. 3.3 W would be formulated as

$$W = \begin{pmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \\ w_{3,1} & w_{3,2} \\ w_{4,1} & w_{4,2} \\ w_{5,1} & w_{5,2} \end{pmatrix}$$

Finally, bias parameters $\vec{b} \in \mathbb{R}^M$ are added to the results. The equation for this layer can be formulated as:

$$\vec{y}(W, \vec{b}) = f(\vec{x} \cdot W + \vec{b}) \tag{3.30}$$

f is an element-wise non-linear activation function. During training the parameters W and $\vec{b}$ are adjusted such that the element-wise difference between the prediction y and a true label $\hat{y}$ is minimal.

Convolution Layer

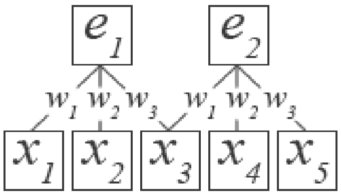


Figure 3.4: Weights in a convolution layer are shared across the input. The filter weights are defined as $\{w_1, w_2, w_3\}$.

When dealing with high-dimensional data it may be unfeasible to connect every output node with every input node as we have seen in the dense layer. In a *convolution layer* or *conv-layer* an output node is instead connected to a local region of its input using *shared weights* (Fig. 3.4). The spatial extend of this connectivity is a hyper-parameter called the *receptive field*, inspired by the receptive field of the sensory neurons in our brain [45, 62]. Such a layer computes a *discrete convolution* which is a weighted sum within this receptive field. The weighting field is also called convolution kernel or filter. In a convolution the kernel is moved along the spatial dimensions of the input to compute a weighted sum within its receptive field. For a one-dimensional input $\vec{x}$, a one-dimensional kernel $\vec{w}$ and the output vector $\vec{a}$, a convolution is defined as:

$$\vec{a}[n] = (\vec{x} * \vec{w})[n] = \sum_{m=-M}^{M} \vec{x}[m]\vec{w}[n-m] \tag{3.31}$$

Although in deep-learning frameworks [1, 58] it is mostly implemented as a cross-correlation, i.e. the kernel is not mirrored like in the traditional convolution definition:

$$\vec{a}[n] = (\vec{x} \star \vec{w})[n] = \sum_{m=-M}^{M} \vec{x}[m]\vec{w}[n+m] \tag{3.32}$$

The weights of the kernel are optimized the same way as the weights of other layers. Finally, bias weights are added on the convolution outputs and an element-wise non-linear activation function f can be applied after that to obtain the layer output $\vec{y}$:

$$\vec{y}[n] = f(\vec{a}[n] + \vec{b}[n]) \tag{3.33}$$

1D Convolution. A 1D convolution convolves an input in only one spatial dimension. This type of convolution is also sometimes called a *temporal* convolution as it is mostly used on sequential data which only spans to one spatial (temporal) dimension (see Fig. 3.5).

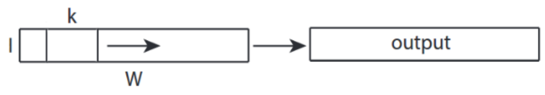


FIGURE 3.5: A 1D convolution with kernel width k and input of size $1 \times W$.

2D Convolution. A two dimensional convolution is a convolution which is extends its locality to two spatial dimensions (see Fig 3.6).

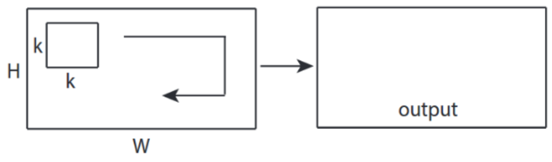


FIGURE 3.6: A 2D convolution with kernel size k and input of size $H \times W$. *Image source:* [128]

3D Convolution. Respectively, a 3D convolution is local in three spatial dimensions (see Fig. 3.7).

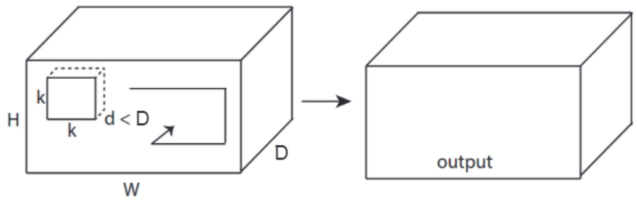


FIGURE 3.7: A 3D convolution with kernel size k and depth d with an input of size $H \times W \times D$. *Image source:* [128]

Convolutions in deep-learning frameworks are only local in their spatial extend but full along the feature-map dimension [1, 17, 58, 62].

Pooling Layer

Pooling is used to subsample the most important features within a feature-map. This is either done by computing the maximum within one receptive field (Max-Pooling) or the average (Ave-Pooling). Similar to the convolution layer the window (or receptive field) of the pooling layer is moved across one feature-map to compute local maximums or averages. It is important to note that only one feature-map at a time is considered (unlike in the convolution layer).

Cross-Channel Pooling. Unlike "normal pooling" which does not consider more than one feature-map at a time, this type of pooling does a pooling operation not only across the spatial dimensions of a layer but also across the feature-map dimension. In the feature-map dimension the spatial extend is not local anymore but full, similar to a convolution layer. An average pooling layer of this type can be considered a convolution layer with all weights set to $\frac{1}{n}$, where n is the number of nodes within its receptive field.

3.4.3 Loss Layers

These types of layers are usually used during training only to determine whether the prediction from the network is correct or not. To do so a loss function is calculated. There are multiple kinds of loss functions. Here we list the two most commonly used ones.

Mean Squared Error (MSE)

The mean square error (MSE) loss computes the squared Euclidean difference between a model's prediction y and the true label $\hat{y}$, i.e. it simply looks at the "average difference":

$$\mathcal{L}(y,\hat{y}) = \frac{1}{2N}\sum_{i=1}^{N} \|y_i - \hat{y}_i\|^2 \tag{3.34}$$

N is the number of total labels, hence $y, \hat{y} \in \mathbb{R}^N$. A factor of $\frac{1}{2}$ is usually multiplied, such that the factor of 2 in the derivative calculation is eliminated. The value of MSE can only be positive. A value close to 0 means that the prediction is close to the true label. This type of loss is usually used for regression problems, i.e. where the ground truth is not necessarily a one-hot vector with values between 0 and 1.

Cross Entropy Error

A cross entropy loss, or log loss, can be used with models whose output are probability values between 0 and 1, e.g. after a Sigmoid or Softmax activation function. This type of loss is most commonly used for classification problems.

Given two distributions over x, where $q(x)$ is the estimate for a true distribution $p(x)$, the cross entropy H is given by [40]:

$$H(q,p) := -\sum_{\forall x} p(x)\log(q(x)) \tag{3.35}$$

In a neural network where the prediction of the model is given by $y \in \mathbb{R}^N$ and the ground truth by $\hat{y} \in \mathbb{R}^N$, the cross entropy loss formula then looks like:

$$\mathcal{L}(y,\hat{y}) = -\hat{y} \cdot \log(y) \tag{3.36}$$

It exists on the range $[0, \infty)$. A cross entropy value close to 0 means that the model's prediction is close to the ground truth. The higher the value of the cross entropy loss is, the further away is the model from the true label.

3.4.4 Other Layers

Dropout

When a neural network is trained on the same training data again and again, it looses the ability to generalize to new examples that the network has not seen before. This process is called *over-fitting*. This is characterized by having a training accuracy which is relatively high and a low test accuracy. This can either happen by using too many training epochs on the training data or training data that is too homogeneous compared to the actual variety. When the latter is the case, one has to augment the training samples with more variety. When the prior is the case one can overcome this by using less training epochs. However, when not a lot of data is available this might leave to under-fitting, i.e. the network does not have enough samples to learn what the data is supposed to represent. This is usually classified by both training and testing accuracies being low. A better way to overcome that type over-fitting problem is to also include variety in the training data or train many different classifies on the set and average their results. However, when not a lot of data or computing power is available both of these options are not very feasible. A technique introduced by Srivastava et al. [123] called *Dropout* combines both of these solutions elegantly without having to look for new training samples or having to train multiple different networks.

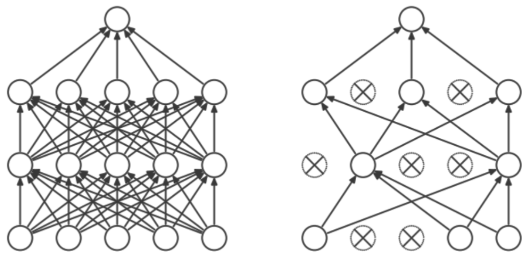


FIGURE 3.8: Dropout neural net model. *Left:* A standard neural net with two hidden layers. *Right:* After applying dropout. *Image source:* [123]

Dropout "switches" various nodes in a layer "off". This is done by setting random weights in a layer to zero. With each training epoch different nodes are switched off, essentially creating different networks. Fig. 3.8 shows a network without Dropout (*left*) and one with Dropout (*right*). This technique also "augments" the training data with noise, essentially achieving more variety in the training samples. Dropout is usually applied in the last layers of a network during training. Usually 50%-80% of nodes are switched off. Dropout is only applied during training and not used during testing.

Residual Layer

Very deep neural networks are usually affected by the vanishing gradient problem. Popularized by He et al. [46], the idea behind residual layers is that deeper models should not produce higher training error than its shallower counterpart. This can be achieved by adding the output of the previous layer to the residual block's output. Such a residual block can be seen in Fig. 3.9.

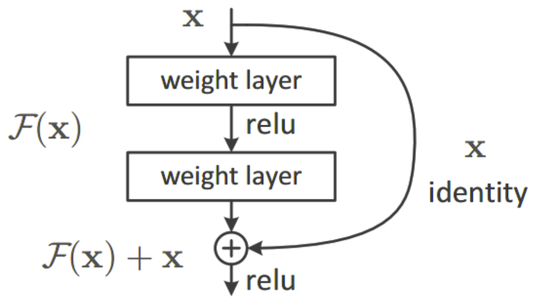

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
$\mathbf{x}$
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

FIGURE 3.9: Residual block. *Image source:* [46]

The illustration shows that the output of the previous layer $\mathbf{x}$ is forwarded to weight layers which filter the input $\mathbf{x}$ and output a feature map $\mathcal{F}(\mathbf{x})$. $\mathbf{x}$ is then added again to that feature map. This forces the network to learn an identity function, i.e. $\mathcal{F}(\mathbf{x}) \approx 0$, for an unneeded layer.

3.5 Feed-Forward Neural Networks (FFN)

The most basic and widely used type of networks are feed-forward neural networks (FFN), as seen in Fig. 3.1. These networks are connected from one hierarchical level to the next, i.e. nodes within one layer are not connected to each other. These parameter connections are also sometimes referred to as (weight) layers. In FFNs these connections only span from neuron layer to the next neuron layer.

3.5.1 Convolutional Neural Networks (CNN)

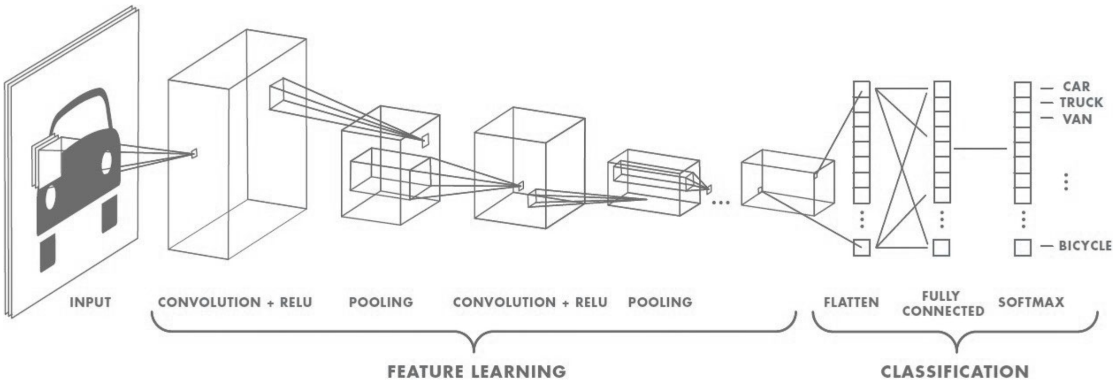


FIGURE 3.10: Typical convolutional neural network architecture with alternating convolution and pooling layers for feature extraction in the beginning, and fully-connected layers in the end for classification. *Image source:* [90]

A Convolutional Neural Network (CNN) is special kind of FFN. This type of network is usually used for object classification tasks in images as they produce filtered images as output feature maps. Although, they can also take other types of data as input. The distinguishing characteristic of this type of network is that it is able to detect local spatial structures within its input, hence its common use for images. Such a network can be seen in Fig. 3.10. The most common CNN [46, 69, 118, 125] configuration are alternating convolution and pooling layers for feature extraction in the beginning, and then a couple of fully-connected layers in the end which re-sample the feature outputs and compute a class probability vector.

3.6 Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNNs) are often used for sequential data such as time-series, i.e. data that has a specific order and a time-step depends on the previous time-step. In contrast to FFNs, they also connect to nodes within the same layer. RNNs are called *recurrent* because they use the same node for every element of a sequence, where the output depends on the previous computations. The dependence of previous information only, makes this type of network *causal*. Such a recurrent node can be seen in Fig. 3.11.

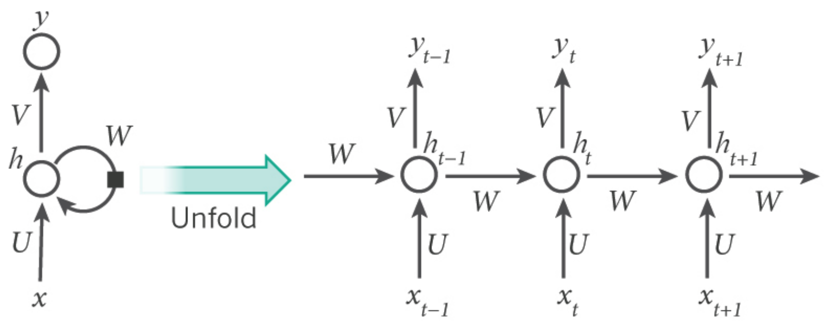


FIGURE 3.11: A recurrent neural network and the unfolding in time of the computation involved in its forward computation. *Image source (edited):* [77]

The above illustration shows a recurrent node being *unfolded*. Unfolding means to write out the full network layer for the complete sequence. x_t is the input and h_t the hidden state at time step t. h_t can be seen as the "memory" of the layer. It is calculated based on the previous hidden state h_{t-1} and input x_t of the current step:

$$h_t = f(Ux_t + Wh_{t-1} + b) \tag{3.37}$$

The function f is a non-linear activation function, U and W are weight matrices and b is the bias. $y_t = Vh_t$ is the output of the layer at time t, where V is another weight matrix.

Bidirectional RNN

A bi-directional RNN is based on the assumption that the output at time t may not only depend on the previous elements but also on future elements (see Fig. 3.12). This makes this type of network *acausal*.

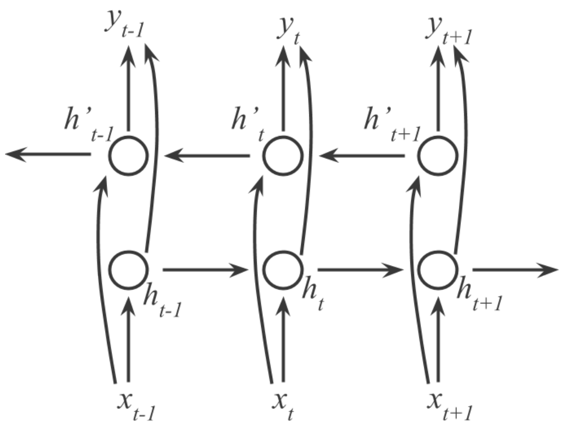


FIGURE 3.12: Bidirectional recurrent neural network.

3.6.1 Long Short-Term Memory (LSTM) Networks

A Long Short-Term Memory (LSTM) [48] network is a type of recurrent network. In theory normal recurrent networks can "memorize" sequences of infinite length. In practice however, this is not the case due to the *vanishing gradient* [47] problem. Since the activation function in a recurrent layer is usually a Sigmoid or Tanh function whose output gradient values are between 0 and 1, the gradient in the back-propagation algorithm gets closer to 0 with each time step because it is multiplied with a gradient value $\frac{\partial \mathcal{L}}{\partial h} < 1$. This means that at some time step t new information from the sequence does not contribute to any change in the output.

An LSTM network tries to overcome this problem by introducing three gates and a *cell state*. An illustration of an LSTM cell can be seen in the Fig. 3.13.

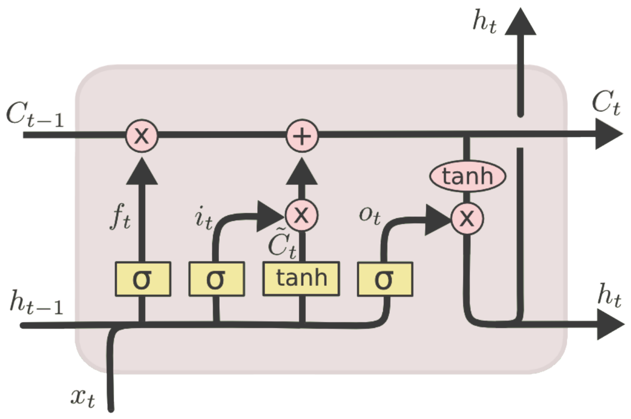
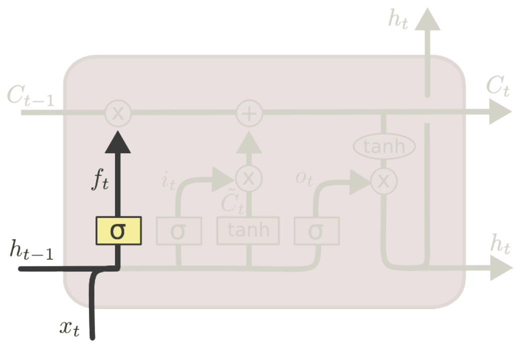


FIGURE 3.13: Visualization of an LSTM cell. *Image source (edited):* [101]

The cell state is the horizontal line running through the top of the diagram (C_{t-1} to C_t). It is key to the LSTM concept. The LSTM has the ability to add or remove information from this cell state. To do so three gates are introduced - the *forget gate* f_t, *input gate* i_t and the *output gate* o_t.

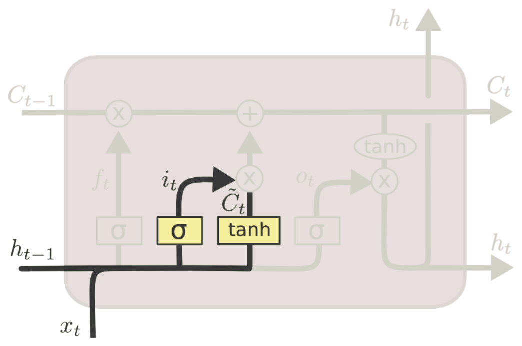


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

FIGURE 3.14: Forget gate highlighted. *Image source (edited):* [101]

The forget gate handles how much of the old cell state C_{t-1} should be forgotten by looking at the sequence element x_t at time step t and the prediction h_{t-1} from the previous time step. A

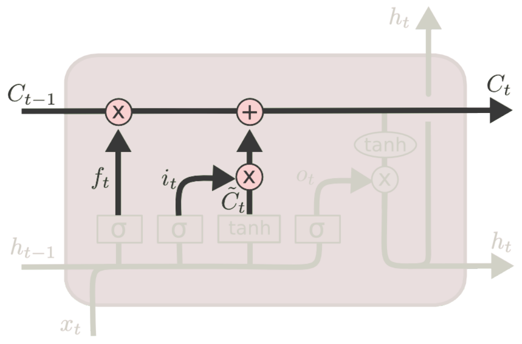
Sigmoid activation function σ is used for that. A value of 0 means that C_{t-1} should be entirely "forgotten", while 1 means it should be kept.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

FIGURE 3.15: Input gate with new candidate values $\tilde{C}_t$ highlighted. *Image source (edited):* [101]

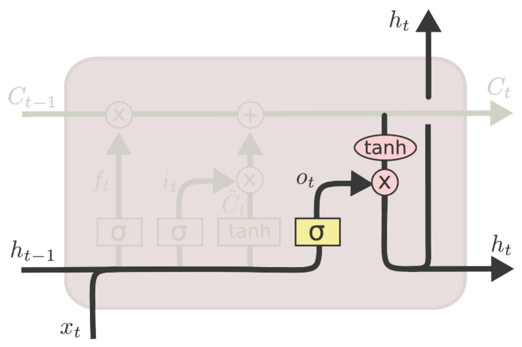
Next, the network has to decide what new information to store in the new state. This is done via the input gate i_t and the new candidate values $\tilde{C}_t$ that could be embedded into the new cell state C_t.



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

FIGURE 3.16: New cell state C_t is computed. *Image source (edited):* [101]

We multiply C_{t-1} with f_t to decide how much of the old state goes into the new state. We then add $i_t \cdot \tilde{C}_t$ to it.



$$o_t = \sigma\left(W_o \ [h_{t-1}, x_t] \ + \ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

FIGURE 3.17: Output gate highlighted. *Image source (edited):* [101]

Finally, we decide what we are going to output. We push the values of the cell state C_t between -1 and 1 with a Tanh function and then we decide which parts of the cell state we are going to output by multiplying the result with the output gate o_t to obtain the prediction h_t for the current state.

The LSTM is not able to entirely eliminate the vanishing gradient problem but it is able to learn longer dependencies due to the weighing of how much of the old cell state should be included in the new cell state. When the forget gate is 1, information of the past is not forgotten and we have something that is called a *constant error carousel* [37, 48] with the function

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \tag{3.38}$$

which enables us to learn long-term dependencies.

Chapter 4

Motion Image Dataset

In this chapter we describe the datasets used for our experiments for the developed models in this thesis.

4.1 BVH Dataset

Our dataset consists of 1378 sequences, which have been captured at 72 FPS using an OptiTrack™[51] motion capture system by project partners in the context of INTERACT [30] and Hybr-iT [28]. It has been captured using 50 joints and is divided into the following takes: *carry*, *carry sideways*, *look around*, *pick*, *place*, *pull*, *push*, *screw*, *side step*, *transfer from one hand to another*, *turn and walk*, *walk*. These takes contain several sequences of the same "take type". Most of these takes contain several other actions within one take, e.g. the picking takes contain some walking bits, while the person is carrying the object. However, apart from the initial "type label" of a sequence it is not further annotated.

4.1.1 BVH Structure

The dataset has been captured in the BVH file format. This section briefly describes this format.

The BVH (BioVision Hierarchical) format has originally been developed by Biovision [91]. The file format is divided into two parts: the first part defines specifications of the initial pose of the character's skeleton (Lst. 4.1), and the latter the motion information per frame (Lst. 4.2).

Listings 4.1 and 4.2 show a short BVH file with just two frames.

BVH Header

```
HIERARCHY
ROOT Hips
{
    OFFSET  0.00    0.00    0.00
    CHANNELS 6 Xposition Yposition Zposition Zrotation Xrotation Yrotation
    JOINT Chest
    {
        OFFSET   0.00    5.21    0.00
        CHANNELS 3 Zrotation Xrotation Yrotation
        JOINT Neck
        {
            OFFSET   0.00    18.65   0.00
            CHANNELS 3 Zrotation Xrotation Yrotation
            JOINT Head
            {
                OFFSET   0.00    5.45    0.00
                CHANNELS 3 Zrotation Xrotation Yrotation
                End Site
                {
                    OFFSET   0.00    3.87    0.00
                }
            }
        }
        JOINT LeftCollar
        {
            ...
            ...
```

Listing 4.1: Skeleton hierarchy in BVH file. *Source:* [38]

Lst. 4.1 shows the header of the BVH file, starting with the keyword **HIERARCHY**, which contains the information of the hierarchical skeleton structure and its initial pose. This is followed by the **ROOT** keyword and the name of the *root joint*. All other joint information is described in relation this root joint. After the hierarchy of the skeleton is described, it is possible to define another hierarchy starting with the keyword **ROOT**. In theory, a BVH file may contain any number of skeleton hierarchies. In practice, however, the number of defined skeletons is kept to one per file [91].

The structure of the skeleton is defined in a recursive manner where each joint's definition, including any children, is encapsulated in curly braces. This is delimited with the keyword **JOINT** (or **ROOT** in case of root joint) on the previous line, followed by the name of the corresponding joint.

The **OFFSET** keyword within the curly braces specifies the translational offset of the joint's origin with respect to its parent's origin (or globally in the case of the root joint) along the X-, Y- and Z-axis. Since there is no further information in a BVH file about how an object should be drawn, the offset information of the first child serves an additional purpose of defining a length and direction for drawing the parent joint.

The second line of a joint's definition defines its degrees of freedom (DOF) and it is prefixed with the keyword **CHANNELS**. This is followed by a number, which specifies the number of DOF, and a list of that many labels indicating the type of each channel. The order of these labels must correspond to their channel order in the motion section of the BVH file. Furthermore, it indicates the concatenation order of the joint's Euler angles to create its rotation matrix. It is permissible to use a different ordering for each joint, as long as it stays consistent across the motion section and when creating the joint's rotation matrix.

After the channel definition comes one of two keywords. Either **JOINT** for the next child or **End Site** to indicate that this is a "leaf joint", i.e. no more children follow.

BVH Motion Information

```
MOTION
Frames:    2
Frame Time: 0.033333
 8.03    35.01   88.36  -3.41    14.78  -164.35  13.09   40.30  -24.60   7.88    43.80
      0.00   -3.61   -41.45   5.82    10.08   0.00    10.21   97.95  -23.53  -2.14
    -101.86 -80.77  -98.91   0.69    0.03    0.00   -14.04   0.00   -10.50  -85.52
    -13.72  -102.93  61.91  -61.18   65.18  -1.57    0.69    0.02    15.00   22.78
    -5.92    14.93   49.99   6.60    0.00   -1.14    0.00   -16.58  -10.51  -3.11
    15.38   52.66  -21.80   0.00   -23.95   0.00
 7.81    35.10   86.47  -3.78    12.94  -166.97  12.64   42.57  -22.34   7.67    43.61
      0.00   -4.23   -41.41   4.89    19.10   0.00    4.16    93.12  -9.69   -9.43
     132.67 -81.86   136.80  0.70    0.37    0.00   -8.62    0.00   -21.82  -87.31
    -27.57  -100.09  56.17  -61.56   58.72  -1.63    0.95    0.03    13.16   15.44
    -3.56    7.97    59.29   4.97    0.00    1.64    0.00   -17.18  -10.02  -3.08
    13.56   53.38  -18.07   0.00   -25.93   0.00
```

Listing 4.2: Motion information in BVH file. *Source:* [38]

Once the skeletal hierarchy is defined, the second section of a BVH file, denoted with the keyword **MOTION**, contains the number of frames (keyword **Frames:**), frame rate (keyword **Frame Time:**) and the channel data. To get the frames per second (FPS), one simply divides 1 by the frame time. So in case of 0.033333, we get a frame rate of 30 FPS. The motion data of the channels follows directly after the frame time information in the next line. Each line of float values represents an animation frame. The numbers appear in the order of the channel specifications.

From this information one can create the animation. To do that, a transformation matrix needs to be inferred from the information that is obtained from the BVH file.

BVH Data Interpretation

For any joint segment the local translation information is simply obtained from the **OFFSET** data as defined in the hierarchy section. The rotation data comes from the **MOTION** section.

For the root object, the translation data is the sum of the offset data and the translation data from the motion section.

The general equation for a transformation matrix is given by

$$M = TRS \tag{4.1}$$

where T is a translation matrix, R a rotation matrix and S a scaling matrix, respectively. Since the BVH files do not contain any scaling information, we can discard this matrix for constructing the local transform and are left with

$$M = TR \tag{4.2}$$

The construction of the rotation matrix R, can be easily done by multiplying together the rotation matrices for each of the different channel axes in the order they appeared in the hierarchy section of the file. For example, consider the following joint channels:

```
CHANNELS 3 Zrotation Xrotation Yrotation
```

This means that R is calculated with

$$R = R_z R_x R_y \tag{4.3}$$

where R_x is the rotation matrix along the X-axis, R_y along the Y-axis and R_z along the Z-axis. The amount of how much is rotated around an axis, is obtained from the **MOTION** section from the file in the order of the channel specifications.

The local translation T is directly obtained from the joint's origin and the per-frame translation data that is added to it. Equation 4.4 illustrates the transformation matrix M after multiplying the rotation matrix R and the translation matrix T together. T_x, T_y and T_z represent the summation of a joint's local position and frame translation data.

$$M = TR = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & T_x \\ r_{21} & r_{22} & r_{23} & T_y \\ r_{31} & r_{32} & r_{33} & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.4}$$

To obtain the global position $\vec{\mathbf{x}}'$ for a given joint, the local transform needs to be pre-multiplied by its parent's global transform, which itself is derived my multiplying its local transform with its parent's global transform and so on [91]. Equation 4.5 illustrates this process. $\vec{\mathbf{x}}$ defines the local origin of the *Head* joint. M_{JOINT} are the local transforms of the *Head* joint's parent joint and its parent joint etc. until the root joint is reached. M' defines the global transform matrix as the composition of these local transformation matrices. $\vec{\mathbf{x}}'$ is finally the global position of the

Head joint in this case:

$$\vec{\mathbf{x}}' = M'\vec{\mathbf{x}} = M_{Hips}M_{Chest}M_{Neck}M_{Head}\vec{\mathbf{x}} \tag{4.5}$$

The *Hips* are the root joint in this case.

4.2 Annotating the Dataset

Of the 1378 sequences we have been able to manually annotate 178 for semantic segmentation. To start of simple, we have annotated the simplest action first - *walking*. For that we have made use of takes which only contain walking and no other action. We have annotated 108 of such *walk* sequences. For more complex experiments we alternate between *pick* and *place* takes, which also contain *walk*, *carry* and *turn* actions. We have annotated 70 of such *mixed* sequences (35 from *pick* and 35 from *place* takes).

The 178 motion capture sequences are made of the 5 main actions *walk*, *pick*, *place*, *carry* and *turn and walk*, and have a length of between 641 and 1532 frames. These have been annotated by us using the following 10 motion primitive labels with their corresponding colors:

Motion Primitive	**Color Label**	**Motion Primitive**	**Color Label**
none	⬛	*end left step*	🟦
left step	🟨	*end right step*	🟪
right step	🟦	*reach*	🟧
begin left step	🟩	*retrieve*	🟫
begin right step	🟥	*turn* (while standing)	🟪

TABLE 4.1: Motion primitive labels and their corresponding colors in label images.

The motion primitive label *none* is used for standing or "masking" other motion primitives in different label granularities as a background class (more on that later in Section 4.2.1).

The 178 sequences show walking in different directions, picking and placing things from and to different regions on different heights, as well as, carrying items. In contrast to other datasets, e.g. [16, 82, 97, 100], our annotations contain fine-grained motion primitive segmentations, such as *left step* or *turn*, instead of basic actions like *walking* to be able to use them for graph-based motion synthesis models [92, 93]. Furthermore, the dataset contains an unbalanced number of frames per label.

We furthermore distinguish between the following different motion primitive types, depending on their classification complexity:

Type 1: Left Step, Right Step, Turn

Motion primitives likes *left step* or *right step* mostly depend on the positions of the lower body joints, e.g. knees, ankles and feet. They are also relatively "simple" in nature, i.e. they do not have any temporal constraints like the *begin/end* step labels.

We do not distinguish in our labels between carrying something or having the arms down, while doing a left or right step.

Furthermore, the motion primitive label *turn* is a special case of the step labels. A turn right for example is essentially a left step, where the whole body rotates around 90° to the right, changing the character's facing direction. Likewise, a turn left is essentially a right step, where the whole body rotates to the left. The *turn* primitive starts from both feet adjacent to each other and also ends with that after the turn.

Type 2: Begin/End Left Step, Begin/End Right Step

These motion primitives are of similar nature to ones in *Type 1*. However, they have an additional temporal constraint to them.

In many motion synthesis frameworks [92, 93] transitioning between two types of actions, e.g. standing and walking is important. Hence, we distinguish between a normal *left step*, where the left leg starts the movement from behind the right leg, and a *start left step*, where both feet are adjacent to each other. *End left step*, on the other hand, ends with both feet adjacent to each other and starts with the left leg behind the right leg. The same applies for *right step* and *begin right step*, *end right step* respectively.

This type of motion primitive is much harder to classify, due to the similarity between a beginning step, a normal step and an ending step - especially when given just a single or a few frame(s) for classification.

Type 3: Reach, Retrieve

This motion primitive type *mostly* depends on the upper body but not entirely.

In the *reach* motion the character either reaches out for an object, while it is picking something up, or it reaches its arm out, while it is placing an object down. The *retrieve* motion is the motion in which the character moves its arm back to its body after it has picked something up or placed something down. Both of these motion primitives do not only depend on the upper body, but also heavily on the hips and knees. This is because the picking or placing can be from and to anywhere, e.g. from the ground or table to a shelve high up or back to the ground. This makes this type of motion primitive harder to classify than *Type 1*, for example.

Since our data uses the hip joint as root joint, all other joint values are relative to it. In Section 4.3 we will explain why using local coordinates is better for frame classification than using global coordinates, despite losing a degree of freedom (the root joint position information) from a such a local coordinate representation. A drawback of these local coordinates however, is that the position and hence also the velocity of the hips is not captured. I.e. *reach/retrieve* movements that are depending on the hips a lot, e.g. bending over to pick something up from the ground, might add to the difficulty of classifying these motion primitives.

4.2.1 Different Granularities

Walk Sequences

The 108 sequences which only contain a character walking have been labeled using following two different granularities:

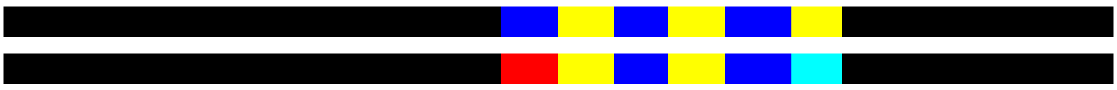


FIGURE 4.1: *Walk 3-labels* and *Walk 7-labels* in comparison on the same sequence.

- *Walk 3-labels*:
  Here only the 3 labels *none* (for standing), *left step* and *right step* are used. No temporal labels such as *begin right step* or *end left step* are utilized. Beginning or ending steps are also classified as either *left step* for *begin/end left step* or *right step* for *begin/end right step*, making classification of this granularity level easier, since the classifier does not need to distinguish between beginning, ending and other steps anymore.

Motion Primitive	**Color**	**#Frames**
none	⬛	52824
left step	🟨	20742
right step	🟦	20768

TABLE 4.2: Number of frames for *Walk 3-labels*.

- *Walk 7-labels*:
  In this version of the dataset, the labels are more fine-grain, i.e. begin and end steps are differentiated from "normal" steps. Hence we make use of the 7 labels *none* (for standing), *begin left step*, *begin right step*, *end left step*, *end right step*, *left step* and *right step*. With these additional labels, we would like to see how well the classifier is able to handle temporal dependencies of labels from each other.

Motion Primitive	**Color**	**#Frames**	**Motion Primitive**	**Color**	**#Frames**
none	⬛	52824	*begin left step*	🟩	367
left step	🟨	16067	*begin right step*	🟥	4898
right step	🟦	14355	*end left step*	🩵	4308
			end right step	🟪	1515

TABLE 4.3: Number of frames for *Walk 7-labels*.

A comparison of these two granularities can be seen in Fig. 4.1.

72 of the 108 sequences are used for training and the rest for testing.

Mixed Sequences

The remaining 70 sequences which contain picking, placing, walking and carrying actions are classified using the following four granularity levels:

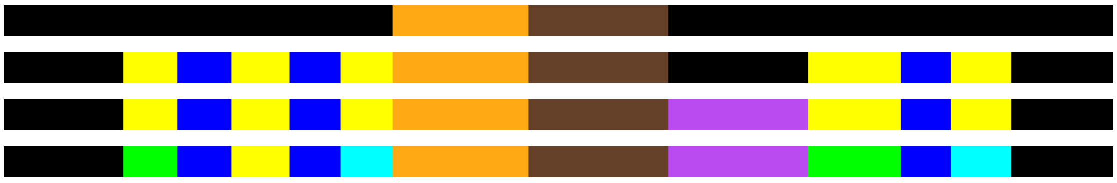


FIGURE 4.2: Different label granularities of the same sequence. From top to bottom: *Mixed 3-labels*, *Mixed 5-labels*, *Mixed 6-labels*, *Mixed 10-labels*.

- *Mixed 3-labels*:
  Since all of the remaining 70 of the total 178 labeled sequences include picking and placing parts, the initial granularity we start with is using 3 labels with *none*, *reach* and *retrieve*. Despite having the same granularity level as *Walk 3-labels*, classifying these sequences is more difficult due to the reasons explained in the previous section 4.2 under *Type 3* motion primitives. All other motion primitives are put into the *none* category. I.e. walking parts in the sequence are not specifically labeled with *left step*, *right step* etc.

Motion Primitive	**Color**	**#Frames**
none	⬛	71021
reach	🟧	10659
retrieve	🟫	6813

TABLE 4.4: Number of frames for *Mixed 3-labels*.

- *Mixed 5-labels*:
  In this level of granularity, we want to see how well the classifier is able to handle data with multiple actions. Hence, here the walking bits in the sequences are not classified as *none* anymore, but have the *left step*, *right step* labels. Beginning and ending steps are classified as their "normal" step equivalents. Turning motions are ignored and put into the *none* label, along with standing. So all in all we make use of the following labels in this granularity level: *none*, *left step*, *right step*, *reach*, *retrieve*.

Motion Primitive	**Color**	**#Frames**	**Motion Primitive**	**Color**	**#Frames**
none	⬛	36620	*reach*	🟧	10659
left step	🟨	8958	*retrieve*	🟫	6813
right step	🟦	7201			

TABLE 4.5: Number of frames for *Mixed 5-labels*.

- *Mixed 6-labels*: This label granularity level is similar to *Mixed 5-labels*. All labels from *Mixed 5-labels* are used, except that turning motions are also considered and not put into the *none* category. We thus utelize the labels *turn*, *none*, *left step*, *right step*, *reach* and *retrieve*.

Motion Primitive	**Color**	**#Frames**	**Motion Primitive**	**Color**	**#Frames**
none	⬛	29990	*reach*	🟧	10659
left step	🟨	8958	*retrieve*	🟫	6813
right step	🟦	7201	*turn* (while standing)	🟪	6630

TABLE 4.6: Number of frames for *Mixed 6-labels*.

- *Mixed 10-labels*: In this highest granularity level, we use all 10 motion primitive labels as described in table 4.1. *none* is here only used for a standing character. Beginning and ending steps are no longer only classified as either *left step* or *right step* but as *begin left/right step* and *end left/right step*.

Motion Primitive	**Color**	**#Frames**	**Motion Primitive**	**Color**	**#Frames**
none	⬛	29990	*end left step*	🟦	4763
left step	🟨	8958	*end right step*	🟪	4797
right step	🟦	7201	*reach*	🟧	10659
begin left step	🟩	2173	*retrieve*	🟫	6813
begin right step	🟥	6509	*turn* (while standing)	🟪	6630

TABLE 4.7: Number of frames for *Mixed 10-labels*.

A comparison between these granularity levels in the 70 mixed sequences can be seen in Fig. 4.2.

60 of the 70 sequences will be used for training and the rest for testing.

4.3 Motion Image Generation

To turn the motion segmentation problem into an image segmentation one, we transform our motion capture data to an RGB image, much in the spirit of [5, 64, 71]. In this section we describe how such *motion images* are obtained.

Of the 50 original joints we use the following 19 for further motion processing:

- *hips (root joint)*
- *lower spine*
- *upper spine*
- *neck*
- *head*
- *left shoulder*
- *right shoulder*
- *left elbow*
- *right elbow*
- *left wrist*
- *right wrist*
- *left hand*
- *right hand*
- *left knee*
- *right knee*
- *left ankle*
- *right ankle*
- *left foot*
- *right foot*

4.3.1 From XYZ to RGB

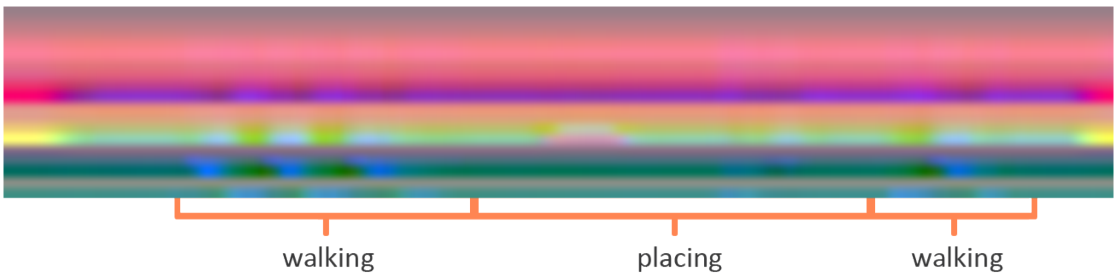


FIGURE 4.3: Sample generated motion image from local XYZ joint position coordinates. Each columns represents a frame in the motion capture sequence, while each row the XYZ joint locations in RGB space. The image is then scaled vertically to a fixed height using bi-cubic interpolation.

While the standard format for BVH data comes in the form of joint angles in the local coordinate systems, we make use of the joint positions instead. The disadvantage of using the joint angles directly is that joint rotations are incremental in a hierarchical skeletal model. Additionally, the same rotation produces different values (e.g. $-90° = 270°$). However, positions relative to root are absolute and easy to directly learn from [49].

Let $S_{xyz} \in \mathbb{R}^{N\times M\times 3}$ be the matrix representation of a sequence of body skeletons with N joints in the Cartesian coordinate system. Each column-vector of the matrix represents one such Cartesian pose with the XYZ joint position coordinates in its entries at a frame m. Such a representation is shown as follows:

$$S_{xyz} = \begin{pmatrix} [x_1(1), y_1(1), z_1(1)] & \dots & [x_1(M), y_1(M), z_1(M)] \\ \vdots & \ddots & \vdots \\ [x_N(1), y_N(1), z_N(1)] & \dots & [x_N(M), y_N(M), z_N(M)] \end{pmatrix} \tag{4.6}$$

N and M represent number of joints and frames, respectively. For our dataset we use $N = 19$ for the 19 joints listed above with the same order (from *top* to *bottom*, *left* to *right*), with the first one being the hips and the last one being the right foot.

The values in S_{xyz} are then normalized to the interval [0, 255]. The maximum value in each X-, Y- and Z-dimension within one motion sequence is set to 255, while the lowest one in each dimension is set to 0:

$$S_{xyz} = \begin{pmatrix} [r_1(1), g_1(1), b_1(1)] & \dots & [r_1(M), g_1(M), b_1(M)] \\ \vdots & \ddots & \vdots \\ [r_N(1), g_N(1), b_N(1)] & \dots & [r_N(M), g_N(M), b_N(M)] \end{pmatrix} \tag{4.7}$$

Finally, we scale the resulting images vertically to a fixed height H using bi-cubic interpolation such that our method can be applied to datasets with other skeletons and number of joints. For our experiments we set $H = 224$. Our resulting motion image has thus the dimensions $224 \times M \times 3$. An example of the resulting RGB image can be seen in Fig. 4.3.

4.3.2 Local Coordinates as Motion Data Normalization

For many machine learning algorithms, such as Support Vector Machines (SVM) [18], K-nearest neighbor [2] or neural network classifiers [110], data normalization is an important pre-processing step [122]. In image classification, for example, the mean RGB value across all training images is computed and then subtracted from each image that is supposed to be classified [69, 118]. This is known as mean-subtraction. Other forms of normalization are scaling the features, such that the covariance is the same, or Whitening [65]. The latter achieves minimum correlation among all features. LeCun highlighted the importance of these normalization techniques in terms of convergence speed in [78]. However, these techniques are very general in nature and not task specific. Since we know, we only deal with motion capture data, we propose a simpler normalization method.

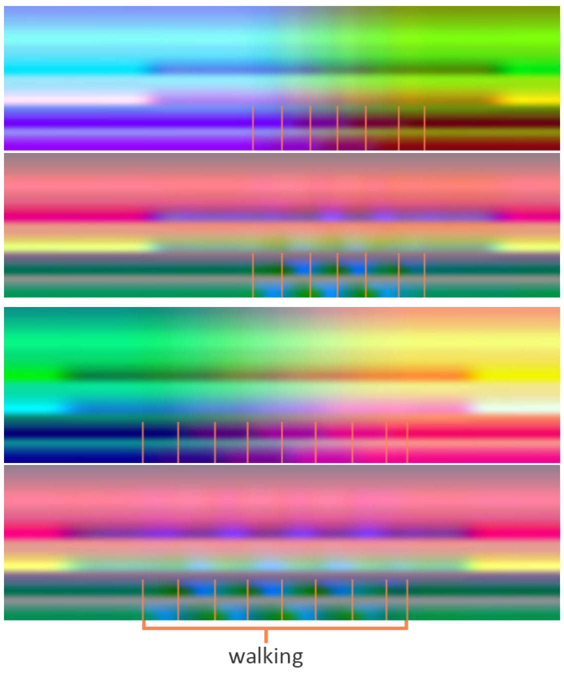


Figure 4.4: Global and local coordinates of two walking sequences. The first and the third image show the motion images obtained from XYZ coordinates in global space, while the second and fourth images show the motion images obtained from the local XYZ coordinate space. Despite both sequences only containing walking and standing actions, the images obtained from global coordinates look very different from each other, while the ones obtained from local space have a homogeneous pattern. The boundaries of the left and right steps are highlighted. Note how in the motion images obtained from local coordinates each step has the same color pattern, while in the images obtained from global coordinates the colors differ from each other and the step boundaries are not as clearly visible.

When applying our method on global and local coordinates (Fig. 4.4), we see that motion images with global coordinates look vastly different from each other, despite showing the same action. Hence, making it harder to detect specific patterns. Thus, we propose a normalization method specifically for motion data by using the local coordinates instead. We obtain them by setting the transforms of the root joint to 0. Motion images from these coordinates look similar to each other for the same type of action. Even a human observer can see some patters in the motion images obtained from local XYZ coordinates as shown in Fig. 4.4. Using the local coordinates, keeps the RGB values in the same range for the same motions, i.e. a left step looks the same (only translated within an image) across all images.

Chapter 5

Dense-FFN for Frame Classification

In this chapter we introduce a simple method for segmenting the generated *motion images.* We do that with a shallow fully-connected feed forward neural network, by classifying one frame at a time.

5.1 Architecture

We start off by using a simple feed-forward network consisting of only fully-connected layers.

Similar to Holden et al. [49] we use a two hidden-layer network with 512 features each, with ReLU activations [43]. Our last layer is another dense layer with a Softmax activation function. Our architecture can be seen in Table 5.1.

Layer Name (type)	**Activation**	**Weight Matrix Dims.**	**Output Shape**
Input (input)	None	None	(H, 1, 3)
Flatten (flatten)	None	None	(1, 672)
FC1 (dense)	ReLU	(672, 512)	(1, 512)
FC2 (dense)	ReLU	(512, 512)	(1, 512)
Out (dense)	Softmax	(512, C)	(1, C)

Table 5.1: Dense-FFN architecture.

Our initial layer is an input layer, which takes a frame of shape $H \times 1 \times 3$. $H = 224$ is here the height of the sample, 1 the width and 3 the three RGB channels. The frame represents a column of the motion image. Since our network consists of only fully-connected layers, we need to flatten this input by reshaping it to be 1×672. This is done by concatenating the channels using the *Flatten* layer operation in Keras [15]. The next two layers are then fully-connected layers with 512 outputs each. The final layer is another dense layer but with a Softmax activation function which maps the output from its previous layer to the number of C classes.

5.2 Evaluation

We start our experiments on our simplest label granularity *Walk 3-labels*. Once we successfully segmented these sequences, we continue to more sophisticated datasets, which include temporal information (e.g. *begin left step*, *begin right step*, *end left step*, *end right step* in *Walk 7-labels*) or deal with mixed actions with root-joint dependent motion primitives, e.g. in the *Mixed x-label* granularities.

5.2.1 Implementation Details

Due to the high-dimensional parameter space, we found the Adam optimizer [66] to perform the best. The suggested parameters in [66] are used for the optimizer. Furthermore, we use a cross-entropy loss and 100 epochs. The experiments are done using Keras [15] with a Tensorflow [1] back-end.

5.2.2 Data Balancing

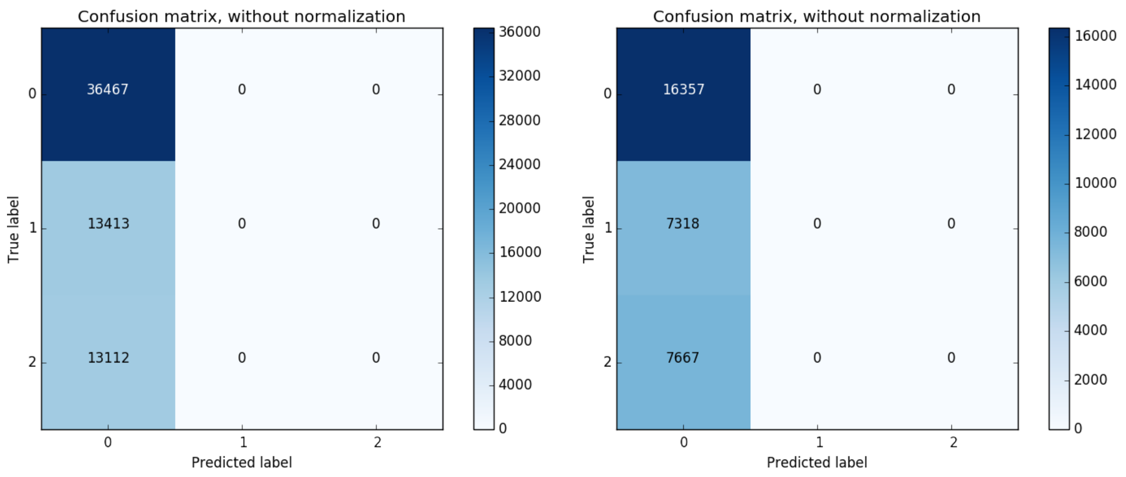


FIGURE 5.1: Confusion matrices for unbalanced training (*left*) and testing (*right*) sets for *Walk 3-labels*. Labels: 0 - *none*, 1 - *right step*, 2 - *left step*.

When we directly apply our network to the dataset, our network classifies all frames to class *none*, as seen in the confusion matrix in Fig. 5.1. This is because most of the frames have the label *none* (around 50%-60%). So a simple solution of minimizing the loss for a classifier that is not very complex, is to classify all frames to the class which is represented the most in the training data. Hence, for our classifier to work properly, we first need to balance the data. An easy way to do that is to determine which class has the least samples. This number of samples is given by S_{min}. Then one just selects S_{min} samples across all classes.

5.2.3 Evaluation on *Walk 3-labels*

We start with the simplest dataset, which contains only the 108 walking sequences and the label granularity of *Walk 3-labels*. In total we have 72 training sequences for walking with 62992 frames summed up. After balancing the data, we obtain $S_{min} = 13112$ frames for each of the three labels *left step*, *right step* and *none* (for standing). For the testing data we have 36 motion capture sequences with a total of 31342 frames. For comparison between train and test accuracy, we also balance the test data, after which we obtain 21954 frames ($S_{min} = 7318$ frames per class).

Results

For the balanced dataset, we obtain 97.52% training accuracy and 93.22% test accuracy. The confusion matrix can be seen Fig.5.2. Example train and test results when applied to whole sequences can be seen in Fig. 5.3.

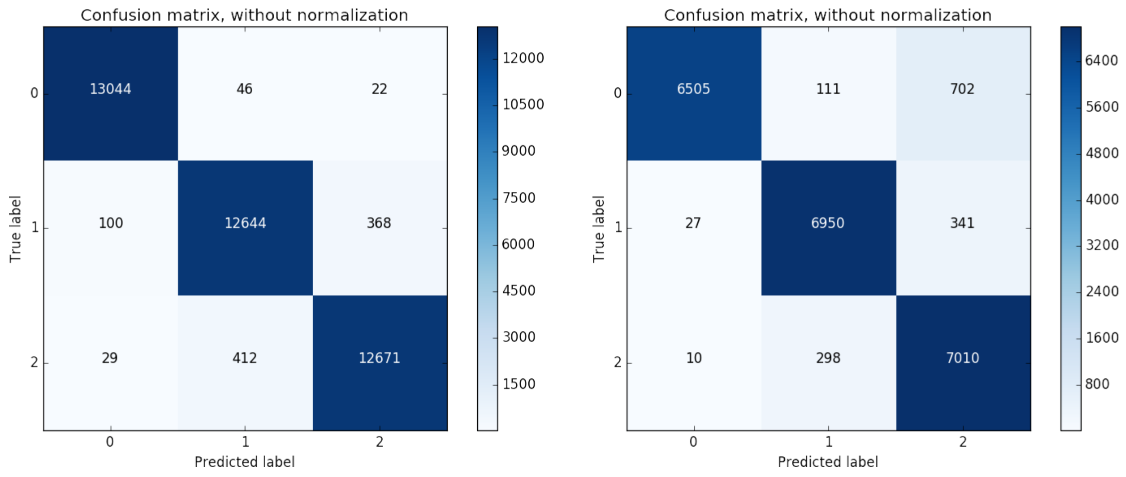


FIGURE 5.2: Confusion matrices for balanced training (*left*) and testing (*right*) sets for *Walk 3-labels*. Labels: 0 - *none*, 1 - *right step*, 2 - *left step*.

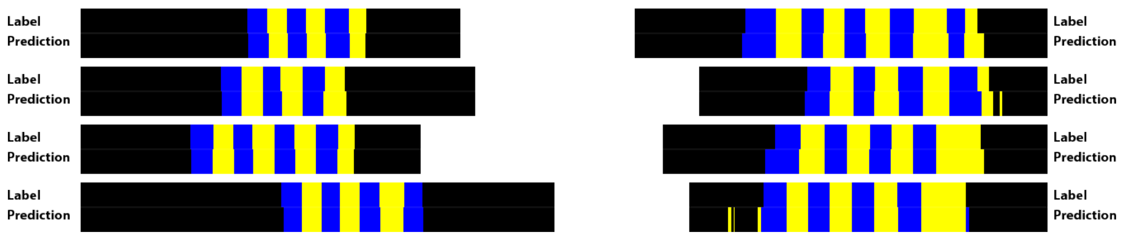


FIGURE 5.3: Four training (*left*) and testing (*right*) results on *Walk 3-labels*. The top half in each example sequence is the ground truth and the second half the prediction from the network. Examples show "typical" performance across dataset. ■: *none*, 🟦: *right step*, 🟨: *left step*.

The results show that a simple two hidden-layer dense network with a receptive field of just one frame can pick up the core features of left and right steps. Nevertheless, we notice - especially in the test results - some over-segmentation.

5.2.4 Evaluation on *Walk 7-labels*

For many motion synthesis models it is important to transition between two motion primitives (e.g. from standing to left step). Hence, data-driven motion models might require motion primitives such as *begin left step*, *end left step*, *begin right step* and *end right step*. These motion primitives are much harder to detect, as they do not differ too much from *left step* and *right step*, respectively. This task is especially difficult for this network, as it takes only one frame at a time and thus has a receptive field size of only one frame. Furthermore, since our network is very prone to unbalanced data, balancing the data with the method mentioned in Section 5.2.2 will leave us with far less training examples, as there are much less beginnings and endings of steps, than there are "normal" steps. We could increase the data by artificially generating new samples from the ones that we already have. However, we show in later chapters that this type of data augmentation is not needed when using more sophisticated types of neural networks.

Results

After balancing the data for *Walk 7-labels* we are left with a mere 2219 frames ($S_{min} = 317$ frames per class) for training and 350 frames ($S_{min} = 50$ frames per class) for testing. This is leaves us at a training accuracy of 95.36%, which is still remarkable, but a mere 62.86% for testing.

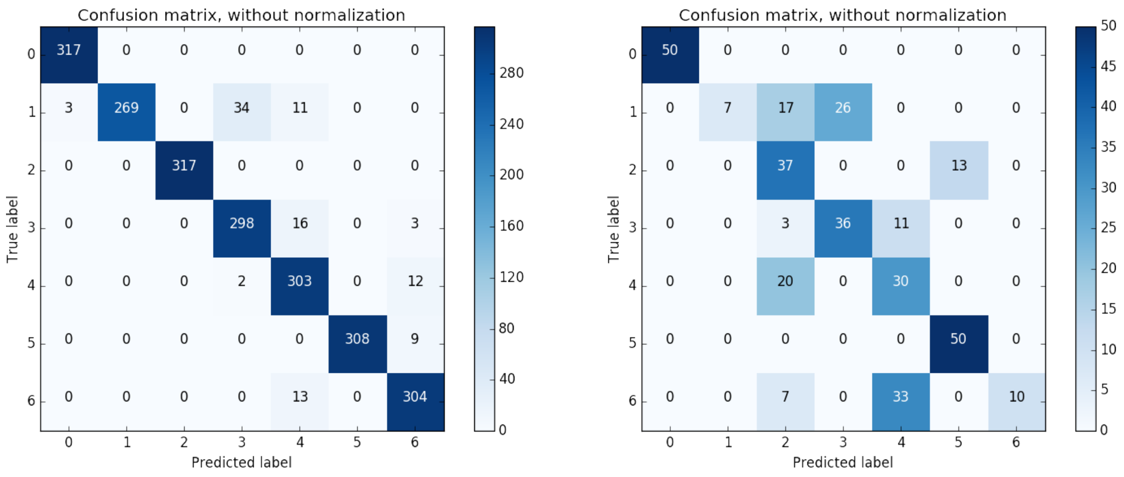


FIGURE 5.4: Confusion matrices for balanced training (*left*) and testing (*right*) sets for *Walk 7-labels*. Labels: 0 - *none*, 1 - *begin right step*, 2 - *begin left step*, 3 - *right step*, 4 - *left step*, 5 - *end right step*, 6 - *end left step*.

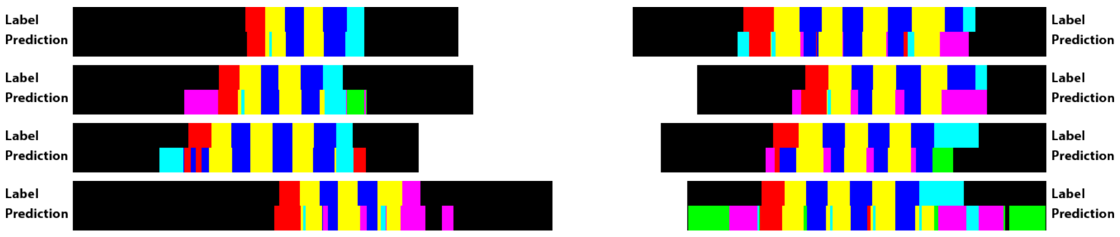


FIGURE 5.5: Four training (*left*) and testing (*right*) results on *Walk 7-labels*. The top half in each example sequence is the ground truth and the second half the prediction from the network. Examples show "typical" performance across dataset. ■: *none*, 🟥: *begin right step*, 🟩: *begin left step*, 🟦: *right step*, 🟨: *left step*, 🟪: *end right step*, 🟦: *end left step*.

While the confusion matrix of the training set looks fine, the confusion matrix of the test dataset (Fig. 5.4 *right*) reveals how *begin right step* (label 1) is often confused with *right step* (label 3). The same can be said about *end left step* (label 6) and *left step* (label 4). From the segmented images, we notice high ambiguities at the starting and ending walking frames, especially in the test samples. Nevertheless, the center steps, seem to be picked up correctly on the training and testing data.

5.2.5 Evaluation on *Walk 3-labels* with a *Walk 7-labels* Balance

To test that the huge drop in accuracy does not result from the 18-fold drop from balancing the training data in the *Walk 7-labels* granularity, we conduct an additional experiment. Instead of using 13112 frames per class for training, as it is the case in *Walk 3-labels*, we use 317 frames per class for training and 50 frames per class for testing, akin to *Walk 7-labels*. Note that the total training and testing samples are even less now in *Walk 3-labels*, since we only have 3 instead 7 classes. Hence, we are dealing with only 951 training frames now and 150 for testing, instead of 2219 and 350 as it is the case with *Walk 7-labels*.

Results

We still obtain a training accuracy of 97.48% on the balanced data and even a test accuracy of 95.99%. The sequence results in Fig. 5.7 show that the results are not as good as the ones in section 5.2.3 but they do not suffer from as much "scattering" as it is the case in the *Walk 7-labels* experiment. However, we do note that there is heavier over-segmentation at the beginning and ending of the walking motions. For a better comparison, we used the same training and testing sequences in Fig. 5.7 as in Fig. 5.5.

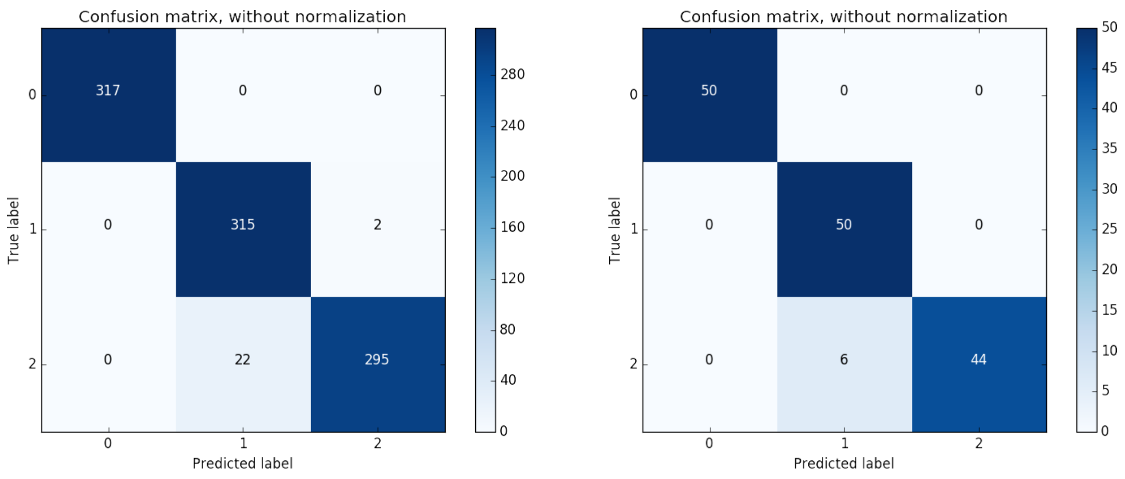


FIGURE 5.6: Confusion matrices for balanced training (*left*) and testing (*right*) sets for *Walk 3-labels* with a balancing from *Walk 7-labels*. Labels: 0 - *none*, 1 - *right step*, 2 - *left step*.

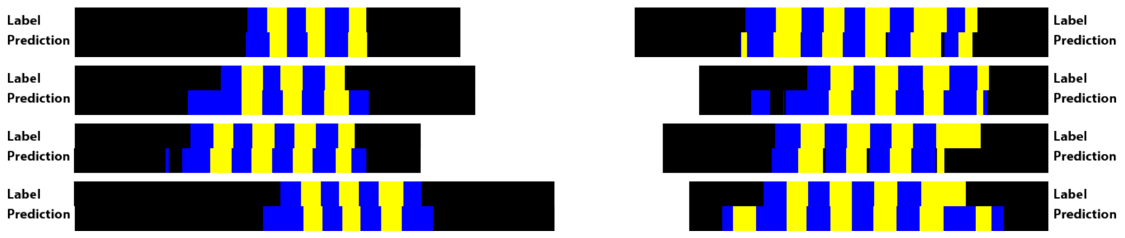


FIGURE 5.7: Four training (*left*) and testing (*right*) results on *Walk 3-labels* with a *Walk 7-labels balance*. The top half in each example sequence is the ground truth and the second half the prediction from the network. Examples show "typical" performance across dataset. ■: *none*, 🟦: *right step*, 🟨: *left step*.

This goes to show us that the cause for the drop of accuracy is not entirely due to the drop of training samples, but the similarities between a "normal" step and a beginning or ending step could also play an important role.

5.2.6 Evaluation on *Mixed 3-labels*

Section 4.3.2 shows that "normalizing" the motion data to local coordinates gives us a more homogeneous appearance for each motion primitive across different motion capture sequences. In computer graphics, subtracting the coordinates of the so-called *pivot point* (i.e. center point of a rotational system) from the other mass points or vertices, results in local coordinates. In motion capturing data the pivot point is equivalent to the root joint. Subtracting its coordinates from all global joint coordinates results in local coordinates. However, by doing so, we also lose a degree of freedom, as our root joint will now always have the coordinates at zero. This makes motions that depend on the root node harder to classify and segment.

In this section we evaluate our model on the 70 mixed action sequences with the granularity type *Mixed 3-labels*. It contains picking and placing actions, which are very hip dependent since picking can also occur from the ground. Placing to the ground is also possible. The label granularity contains the motion primitive labels *reach*, *retrieve* and *none*. The latter describes all walking motion primitives, as well as turning and standing; essentially, everything that does not fall into the category of reaching ones arm out to either grab or place something, or to retrieve ones arm back after having picked something up or put something down. From the 70 sequences, we use 60 for training and 10 for testing. The training sequences have a total of 75211 frames. After balancing the data, we are left with 16473 training frames ($S_{min} = 5491$ frames per class). For testing we have 10 sequences with a total of 13282 frames. Balancing it, leaves us at $S_{min} = 1099$ frames per class.

Results

We obtain a training accuracy of 86.78% from the balanced dataset and a testing accuracy of 58.62%.

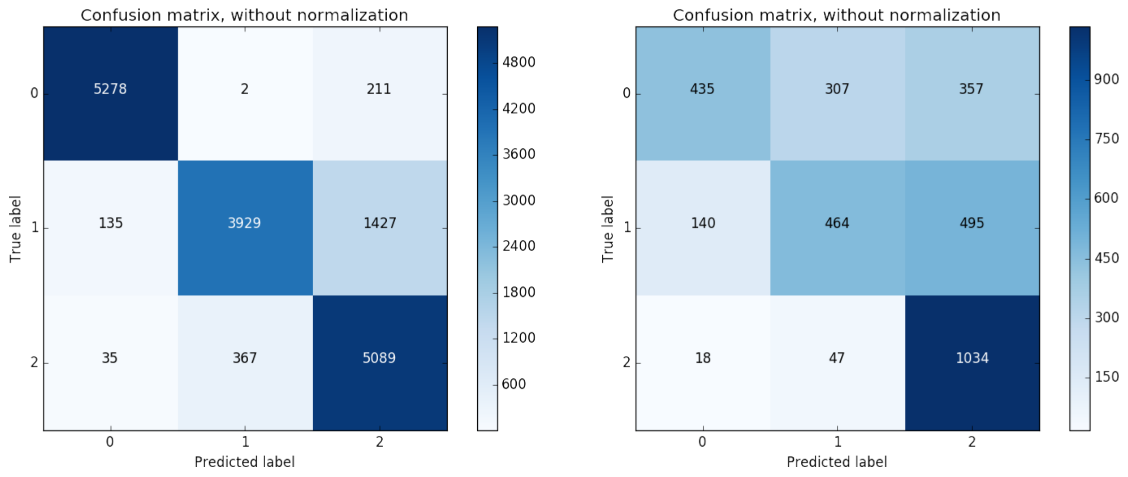


FIGURE 5.8: Confusion matrices for balanced training (*left*) and testing (*right*) sets for *Mixed 3-labels*. Labels: 0 - *none*, 1 - *reach*, 2 - *retrieve*.

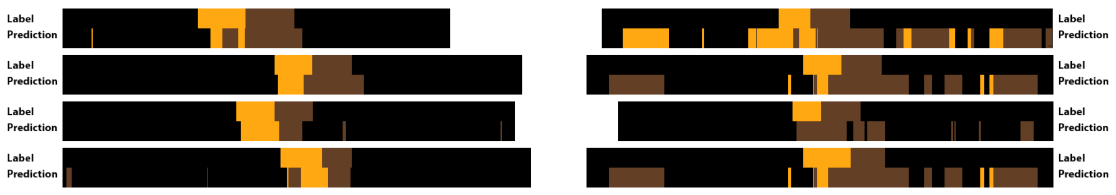


FIGURE 5.9: Four training (*left*) and testing (*right*) results on *Mixed 3-labels*. The top half in each example sequence is the ground truth and the second half the prediction from the network. Examples show "typical" performance across dataset. ■: *none*, 🟧: *reach*, 🟫: *retrieve*.

We notice from the example results in Fig. 5.9 that the network roughly picks up where the picking or placing segments are in the training data but even with a training accuracy of ~90% on the balanced data, the sequences tend to have too many misclassifications. The classifications on the test sequences (Fig. 5.9 *right*) tend to have chunks of one class in a row. However, these chunks seem rather randomly placed, similar to the temporal labels in Section 5.2.4.

5.2.7 Evaluation on *Mixed 5-labels*

Here we evaluate our model on the label granularity *Mixed 5-labels* in the mixed sequence dataset. I.e. stepping motions will not be labeled as *none* but as *left step* and *right step*, respectively. Begin and end step motions are classified as their respective "none begin/end" counterparts. After balancing the dataset, we are again left with $S_{min} = 5491$ training frames per class and $S_{min} = 1099$ testing frames per class.

Results

Interestingly, our training and testing accuracies increase when adding the walking classes, instead of putting them in the *none* category. Our training accuracy increases to 90.95% and our testing accuracy to 69.32%. We assume this is because now the network has more training samples all together. I.e. with *Mixed 3-labels* it uses 3 times 5491 training samples, totaling 16473 samples. However, in this experiment it has $5 \times 5491 = 27455$ samples. Knowing the label of these additional samples of course increases a network's classification accuracy, as it is less likely to confuse them with reach or retrieve.

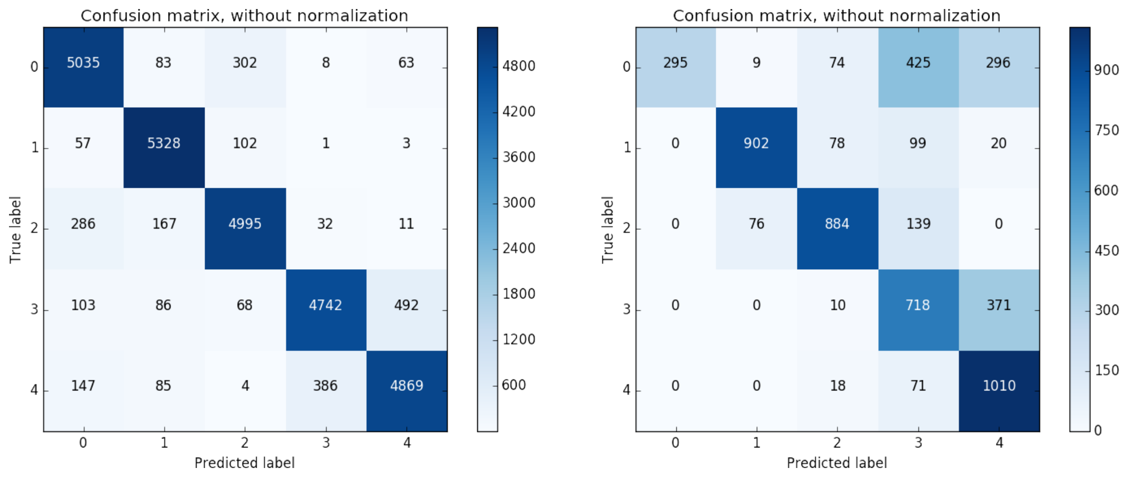


FIGURE 5.10: Confusion matrices for balanced training (*left*) and testing (*right*) sets for *Mixed 5-labels*. Labels: 0 - *none*, 1 - *right step*, 2 - *left step*, 3 - *reach*, 4 - *retrieve*.

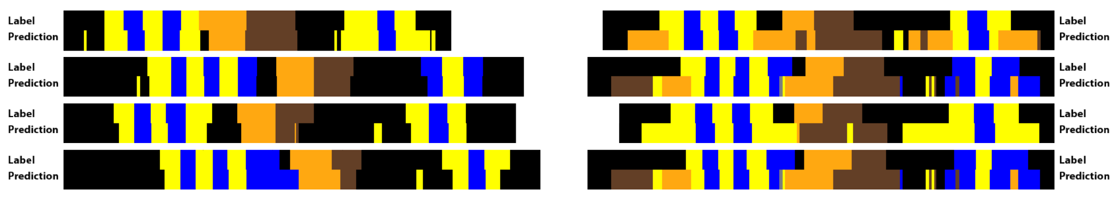


FIGURE 5.11: Four training (*left*) and testing (*right*) results on *Mixed 5-labels*. The top half in each example sequence is the ground truth and the second half the prediction from the network. Examples show "typical" performance across dataset. ■: *none*, 🟦: *rightStep*, 🟨: *leftStep*, 🟧: *reach*, 🟫: *retrieve*.

Fig. 5.11 shows some of the training sequences are classified almost perfectly with some over-segmentation errors. Despite the low testing accuracy on the balanced dataset, the classifier picks the walking and picking or placing parts in the testing sequences up, even if the boundary regions are sometimes far from perfect.

5.2.8 Evaluation on *Mixed 6-labels*

In this experiment we change the label the *turn* segments to their actual *turn* label, instead of putting them in the *none* category. This totals the 6 labels *none*, *left step*, *right step*, *reach*, *retrieve* and *turn*. Balancing the dataset leaves us again at the same number of frames per class for training and testing as in the prior experiments on the mixed sequences. However, since we have 6 labels now, we have a total of $6 \times 5491 = 32946$ training samples and $6 \times 1099 = 6594$ test samples.

Results

Again, the test accuracy increases. The training accuracy decreases a little however. We obtain a train accuracy of 89.38% and a test accuracy of 74.3%. That the test accuracy increased by $\sim 5\%$, while the training accuracy got a bit worse, means that adding the additional samples to the training regularizes the network and decreases overfitting.

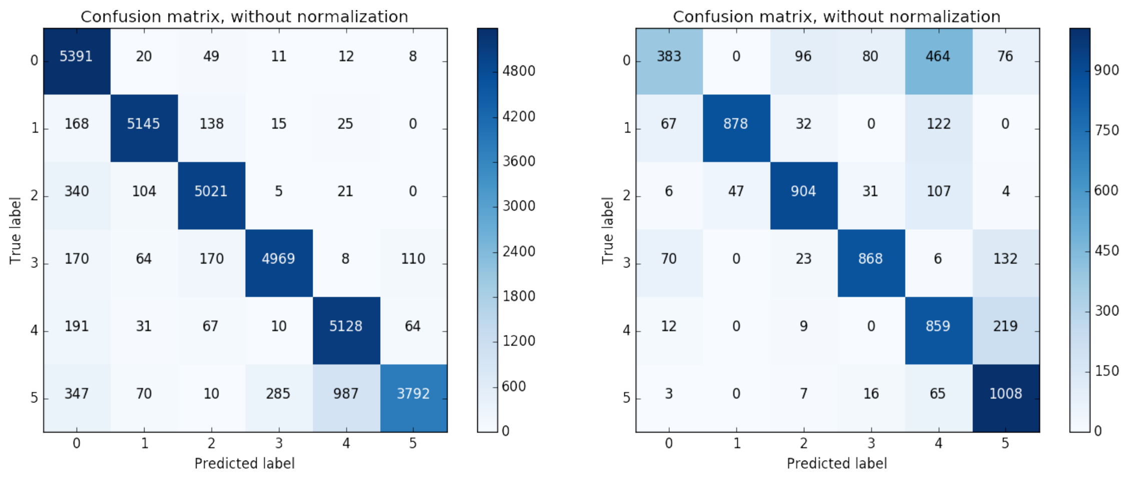


FIGURE 5.12: Confusion matrices for balanced training (*left*) and testing (*right*) sets for *Mixed 6-labels*. Labels: 0 - *none*, 1 - *right step*, 2 - *left step*, 3 - *turn*, 4 - *reach*, 5 - *retrieve*.

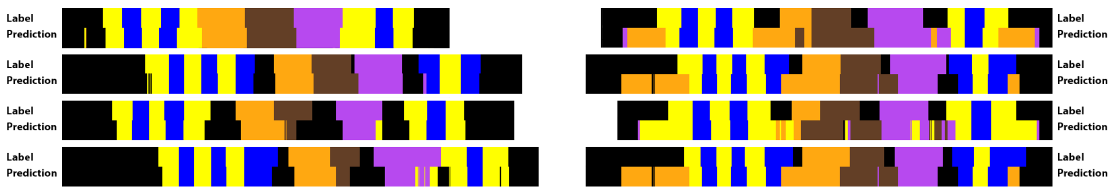


FIGURE 5.13: Four training (*left*) and testing (*right*) results on *Mixed 6-labels*. The top half in each example sequence is the ground truth and the second half the prediction from the network. Examples show "typical" performance across dataset. ⬛: *none*, 🟦: *right step*, 🟨: *left step*, 🟪: *turn*, 🟧: *reach*, 🟫: *retrieve*.

Some of the training sequence results are again almost perfect (Fig. 5.13, *left*), despite having a receptive field size of just one frame. What is interesting is that, especially in the test sequences (Fig. 5.13, *right*) the parts before the first step and after the last step are classified as *reach* in almost every test sequence. This may be because in these sections the character is changing from a T pose to a “normal” standing position and vice versa. These movements of the arms might be the reason why the network consistently classifies these parts as *reach*.

5.2.9 Evaluation on *Mixed 10-labels*

In this last experiment with this model, we try to distinguish all labels in the mixed sequence dataset, i.e. *none* (now used exclusively for standing), *reach*, *retrieve*, *left step*, *right step*, *turn*, *begin left step*, *begin right step*, *end left step* and *end right step*.

Results

After balancing we obtain 18460 training frames ($S_{min} = 1846$ per class) and 3270 test frames ($S_{min} = 327$ per class), which is around half of what we had with *Mixed 6-labels*. Hence, it is no surprise that our test accuracy drops almost 10% from 74.3% to 65.87% on balanced data set, compared to the previous experiment. Our training accuracy is 88.16% on the balanced data.

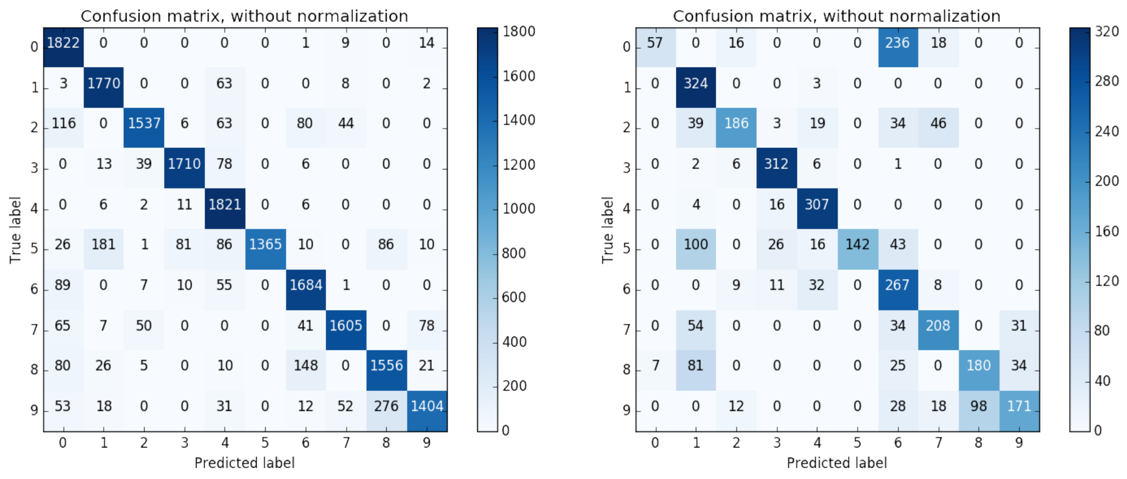


FIGURE 5.14: Confusion matrices for balanced training (*left*) and testing (*right*) sets for *Mixed 10-labels*. Labels: 0 - *none*, 1 - *begin right step*, 2 - *begin left step*, 3 - *right step*, 4 - *left step*, 5 - *end right step*, 6 - *end left step*, 7 - *turn*, 8 - *reach*, 9 - *retrieve*.

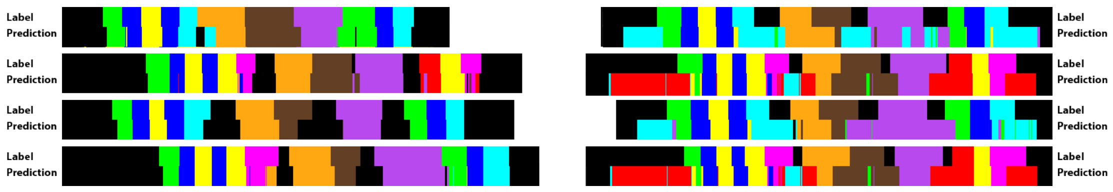


FIGURE 5.15: Four training (*left*) and testing (*right*) results on *Mixed 10-labels*. The top half in each example sequence is the ground truth and the second half the prediction from the network. Examples show "typical" performance across dataset. ■: *none*, 🟥: *begin right step*, 🟩: *begin left step*, 🟦: *right step*, 🟨: *left step*, 🟪: *end right step*, 🟦: *end left step*, 🟪: *turn*, 🟧: *reach*, 🟫: *retrieve*.

Despite the beginning and ending classes being often times misclassified in the training and testing sequences, the overall segmentation - especially in the picking/placing and the intermediate step parts - is satisfactory. While the network is able to somewhat localize and distinguish between the picking or placing and walking (with or without carrying something) parts, the overall segmentation in the test results is still far from state-of-the-art and usable for industry motion synthesis tools. Even though the overall number of training samples was almost halved in this experiment due to the existence of much less beginning and ending steps, the network still catches the overall "idea" of the sequence. I.e. where the walking happens and where the picking or placing happens. Only locating begin and end steps in the test sequence is difficult, which is not much different from the test results in the experiment with *Walk 7-labels*.

5.3 Conclusion

In this chapter we have conducted experiments on two different datasets - one which has only contained *walking* and *standing* actions, and the other *standing*, *walking*, *picking*, *placing*, *turning* and *carrying* actions. We have used seven different granularities in total (see Tab. 5.2). Our network has a total of $\sim 610K$ trainable parameters in the three dense layers, of which two are hidden layers. In these experiments we have done frame-by-frame classification of a single frame at a time. The results are shown in Table 5.2.

	Train Acc.	Test Acc.	# Train Frames	# Test Frames	# Train Frames per Class	#Test Frames per Class
Walk 3-l.	97.53%	93.22%	39336	21954	13112	7318
Walk 7-l.	95.36%	62.86%	2219	350	317	50
Walk 3-l. (7-l. balance)	97.48%	95.99%	915	150	317	50
Mixed 3-l.	86.78%	58.62%	16473	3297	5491	1099
Mixed 5-l.	90.95%	69.32%	27455	5495	5491	1099
Mixed 6-l.	89.38%	74.30%	32946	6594	5491	1099
Mixed 10-l.	88.16%	65.87%	18406	3270	1846	327

TABLE 5.2: Training and testing accuracies on different datasets and their respective number of training and testing frames. Only *Walk 7-l.* and *Mixed 10-l.* contain "temporal labels", e.g. *begin left step* or *end right step*.

We show that a very simple network like the one we have used in this chapter is already able to distinguish simple motions like *left step* and *right step* from each other (rows *Walk 3-l.* and *Walk 3-l.* with 7-label balance in Tab. 5.2). More complex motion primitives as used in the *Mixed* sequences are harder to classify. Nevertheless, the segmentation results shown in the previous subsections illustrate that the network is able to pick up the general notion of the sequence, i.e. where the walking or the picking and placing parts are, despite not being very accurate at the boundaries, as well as having a receptive field size of a single frame. I.e. the network does not have any information regarding the velocity or acceleration of the motion. When we compare *Mixed 3-labels* with *Mixed 6-l.* and *Mixed 10-l.*, we see that not only is the number of different training samples important but the network also tends to perform better when e.g. *right step* is labeled as *right step* and not as *none*.

What our network struggles the most with are “temporal labels”, i.e. *begin left step*, *end left step*, *begin right step* and *end right step*. Both *Walk 7-labels* and *Mixed 10-labels* perform much worse compared to the other granularities. To test that this is not solely due to the reduced training data size but also the similarities between “normal steps” and beginning and ending steps, we have conducted another experiment, where the granularity *Walk 3-labels* uses the same number of frames per class as *Walk 7-labels* (row *Walk 3-l.* with 7-label balance in Tab. 5.2). We show that despite using even less total training data than in *Walk 7-labels*, the network’s results are comparable to the results in *Walk 3-labels* (first row in Tab. 5.2). This concludes that using more training data may not be enough to increase training accuracy. In the next chapters we show how to improve the model, such that it is better in handling these tasks.

Chapter 6

From Dense-FFN to Fully-Convolutional Network

The model defined in Chapter 5 performs well on simple tasks like differentiating between a left and a right step. However, when it comes to motions where a timely relationship or order matters, e.g. defining when we have a first step or a last step, it struggles (see Tab. 5.2 rows *Walk 7-l.* and *Mixed 10-l.*). One solution to that is to increase the receptive field size of our network, i.e. increasing the amount of frames it takes as an input. This way information regarding the velocity and acceleration of the motion are also incorporated in the network input because it is able to look at multiple frames at a time. The downside of simply increasing the number of input frames, is that we have to create a new dataset for every receptive field size that we want to test, which is a lot of overhead. Furthermore, classifying a window of frames at a time is not efficient enough as we need to wait until the previous window is processed. Hence, in this chapter we introduce a method which can take an input of arbitrary size and produce a correspondingly sized output with efficient inference and learning. The classification of each window or receptive field happens in parallel.

6.1 *Convolunizing* Fully-Connected Layers

"In Convolutional Nets, there is no such thing as fully-connected layers. There are only convolution layers with 1x1 convolution kernels [...]."
—Yann LeCun, 2015 [76]

At first glance this quote by LeCun may seem absurd because when we think of 1×1 convolutions, we usually assume that it is simply a point operation. I.e. a scalar is multiplied to some tensor. In neural network architectures however, this is not necessarily the case. Fully-connected layers are usually implemented in deep-learning frameworks [1, 17, 58] as a dot-product. Thus, these networks are restricted to a fixed input size. Fig. 6.1 shows a network with three input values and two output values and fully-connected weights, i.e. every input is connected to every output.

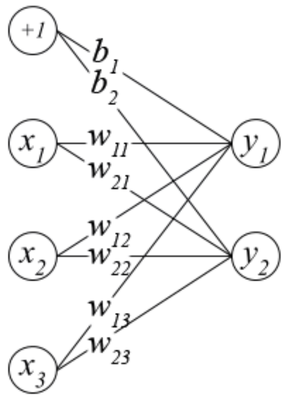


FIGURE 6.1: A fully-connected layer with three input nodes and two output nodes.

The following equation shows the dot-product of the network illustrated above:

$$\left(\begin{bmatrix}x_1 & x_2 & x_3\end{bmatrix} \cdot \begin{bmatrix}w_{11} & w_{21}\\ w_{12} & w_{22}\\ w_{13} & w_{23}\end{bmatrix}\right)^T + \begin{bmatrix}b_1\\ b_2\end{bmatrix} = \begin{bmatrix}y_1\\ y_2\end{bmatrix} \tag{6.1}$$

This representation makes this type of layer very efficient to compute. However, it restricts its input to the size of the weight matrix, which in this case is a 3×2 matrix. Hence, the input has to be of size $H \times 3$ for the dot-product in Eq. 6.1 to work. In the above case H even has to be $H = 1$ for the correct mapping between input vector $\vec{x} = \begin{bmatrix}x_1 & x_2 & x_3\end{bmatrix}$ and output vector $\vec{y} = \begin{bmatrix}y_1 & y_2\end{bmatrix}^T$.

To understand how we can change such an operation to a 1×1 convolution we first have to understand how convolution layers in deep-learning frameworks work.

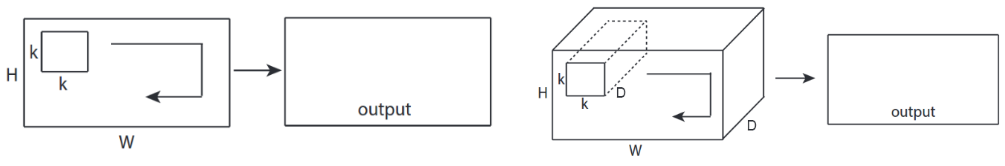


FIGURE 6.2: A 2D convolution with local connectivity in space (i.e. height and width), but full in the depth of the input volume. *Left:* An input of depth 1 is used. *Right:* An input of depth D is used. The kernel depth is equal to the input depth. *Image source:* [128]

As briefly mentioned in Section 3.4.2, in deep-learning frameworks a convolution is only local in space (i.e. width and height) but full along the entire depth of the input volume. Hence, a 1D-convolution layer is actually defined with a 2D-convolution filter kernel and the number of filters in that layer; a 2D-conv layer with a 3D-kernel (Fig. 6.2) and the number of filters in that layer, and so on. Therefore, a 1D-convolution layer is defined with a 3D-tensor of shape $W \times D \times F$, while a 2D-conv layer is defined with a 4D-tensor of shape $H \times W \times D \times F$, respectively. This continues in higher dimensions. H is the kernel height, W the kernel width and D the kernel depth. F is the number different filter kernels in a layer that compute the F feature-maps. For a convolution to be full along the depth of its input, D has to be equivalent to the number of the input feature-maps F.

Deep-learning frameworks have this configuration since their convolution layers are supposed to take input tensors of a higher dimension, e.g. a 2D-conv layer should be able to find spatial features in RGB images.

6.1.1 1×1 Convolution Layer as a Fully-Connected Layer

First introduced by Lin et al. [83], this type of convolution has various useful properties. At first glance it might seem confusing and counter-intuitive if you think of it as a 2D 1×1 image filter, like Sobel [121], applied on some 2D image. However, in deep-learning frameworks this convolution is a 3D-convolution which is full along the input feature-maps, as shown in 6.2 the image below:

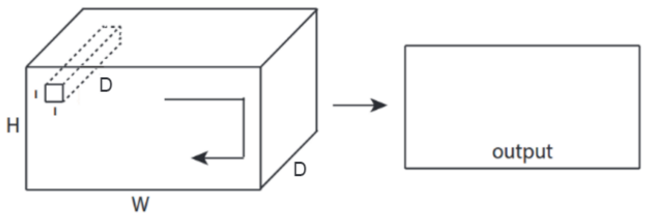


FIGURE 6.3: A 1×1 convolution on a 3D input tensor is equivalent to a fully-connected layer. *Image source:* [128]

The 3D-convolution operation that results from such a setting, essentially computes a dot-product within the convolution window. This is because an output node y_{ij} at position $(i, j) \in ([1, H], [1, W])$ is a result of a dot-product between the input nodes $\vec{x}_{ij}$ across the input feature-maps and the D values in the convolution kernel channels. Hence, it is a channel- or depth-wise dot-product, which makes it equivalent to a fully-connected layer. Fig. 6.4 shows how the fully-connected layer in Fig. 6.1 can be changed to a 1×1 2D-conv layer.

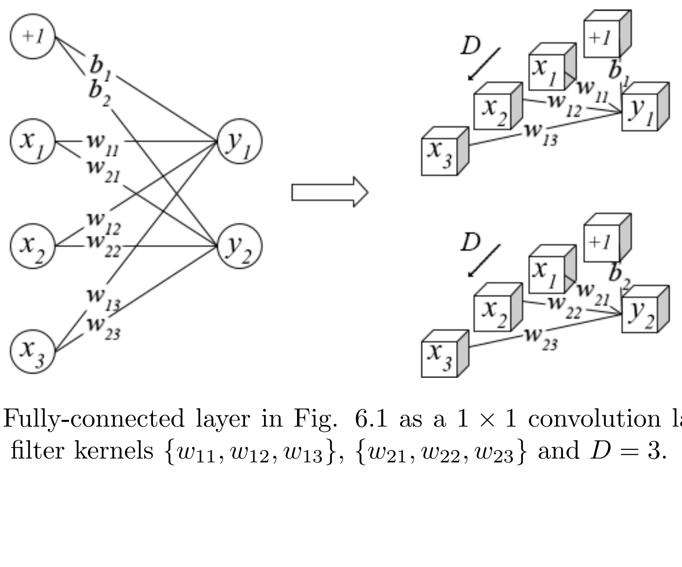


FIGURE 6.4: Fully-connected layer in Fig. 6.1 as a 1×1 convolution layer with two filter kernels $\{w_{11}, w_{12}, w_{13}\}$, $\{w_{21}, w_{22}, w_{23}\}$ and $D = 3$.

Apart from now being able to include arbitrarily sized inputs in our network once we change fully-connected to convolutoinal layers, this type of layer is also used to change the dimensionality in the feature space without changing the spatial dimensions [83, 125]. Thus, this type of layer is also sometimes called a "bottleneck layer" [115].

6.1.2 $H \times 1$ Convolution Layer as a Fully-Connected Layer

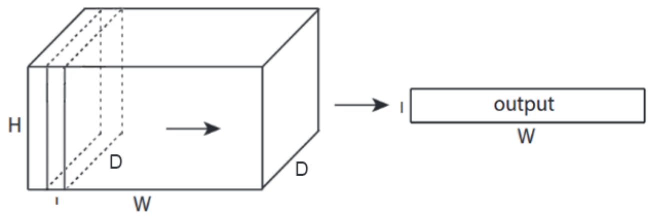


FIGURE 6.5: $H \times 1$ convolution on a 3D input tensor to obtain temporal features.

Another possibility to transform a fully-connected layer to a convolution layer is to use a $H \times 1$ convolution. This is possible when the output is supposed to be one-dimensional to emphasize temporal features. Assume we have an input tensor of shape $H \times W \times D$. However, instead of an output shape of $H \times W$, we wish to have a shape of $1 \times W$. We can achieve this by having a convolution kernel of shape $H \times 1 \times D$, where the kernel is full along the height and depth of the input tensor (see Fig. 6.5). This is equivalent to concatenating the feature-maps of the input across the first dimension (resulting in an input shape of $H \cdot D \times W$) and of the kernel across the channel-dimension (resulting in a kernel shape of $H \cdot D \times 1$), and computing the dot-product between each column of the input tensor and the resulting filter, which is in fact a fully-connected operation between an input layer of shape $W \times H \cdot D$ and a fully-connected weight matrix of shape $H \cdot D \times 1$.

6.1.3 Fully-Convolutional Network (FCN)

With a *fully-convolutional* architecture like this, where every weight layer is defined with a convolution, we can not only extend our network to take arbitrarily sized inputs but we are also no longer bound to creating a new dataset every time we want to test a new receptive field size. We can use the same dataset with the sequences in their full length and simply enlarge the width of the convolution kernel to consider more frames at a time for its output. Finally, an output is computed for each window of size of the receptive field of the network. Such an architecture where every weight layer is a convolution is called a fully-convolutional network (FCN). It can be seen as a stack of high-dimensional non-linear filters, making it a non-linear filter on its own.

In the next section, we describe how to change the network introduced in Chapter 5 to take inputs of arbitrary sizes.

6.2 Architecture

The architecture used in the previous chapter is listed below:

Layer Name (type)	**Activation**	**Weight Matrix Dims.**	**Output Shape**
Input (input)	None	None	(H, 1, 3)
Flatten (flatten)	None	None	(1, 672)
FC1 (dense)	ReLU	(672, 512)	(1, 512)
FC2 (dense)	ReLU	(512, 512)	(1, 512)
Out (dense)	Softmax	(512, C)	(1, C)

TABLE 6.1: Dense-FFN architecture.

The equivalent FCN architecture is thus:

Layer Name (type)	**Activation**	**Kernel Shape**	**Output Shape**
Input (input)	None	None	(H, 1, 3)
Conv1 (2D conv)	ReLU	(H, 1, 3, 512)	(1, 1, 512)
Conv2 (2D conv)	ReLU	(1, 1, 512, 512)	(1, 1, 512)
ConvOut (2D conv)	Softmax	(1, 1, 512, C)	(1, 1, C)

TABLE 6.2: FCN architecture of Dense-FFN.

Notice how we do not need to flatten our input before we hand it in to the convolutionized fully-connected layer. Instead of a 1×1 convolution, the first fully-connected layer has been turned to a layer with $H \times 1$ convolutions, where the height of the convolution kernels is equivalent to the height of the image, making the use of a prior flattening of the input layer redundant. We now get a classification output for every $H \times 1$ ($H = 224$) window of the input sequence.

6.2.1 Validate Conversion

To validate that our conversion from Dense-FFN to an FCN is correct we now have to insert the trained parameters of the Dense-FFN into the trainable parameters of our FCN and confirm that we get the same accuracies for the training and test sets that we used for the experiments in Chapter 5.

In Keras our Dense-FFN model is defined the following way:

```
model = Sequential()
model.add(Flatten(input_shape=(224, 1, 3)))
fc1 = Dense(512, activation='relu')
model.add(fc1)
fc2 = Dense(512, activation='relu')
model.add(fc2)
out = Dense(num_classes, activation='softmax')
model.add(out)
```

LISTING 6.1: Keras Implementation of our Dense-FFN model.

And the FCN model the following way:

```
model_fcn = Sequential()
conv1 = Conv2D(512, activation='relu', kernel_size=(224, 1), input_shape=(224,None,3))
model_fcn.add(conv1)
conv2 = Conv2D(512, activation='relu', kernel_size=(1, 1))
model_fcn.add(conv2)
conv_out = Conv2D(num_classes, kernel_size=(1, 1), activation='softmax')
model_fcn.add(conv_out)
```

Listing 6.2: Keras Implementation of our FCN model.

To set the trained parameters of the dense layers to the trainable parameters of the convolution layers, we use the following code:

```
conv1.set_weights([fc1.get_weights()[0].reshape(-1, 1, 3, 512), fc1.get_weights()[1]])
conv2.set_weights([fc2.get_weights()[0].reshape(-1, 1,512,512), fc2.get_weights()[1]])
conv_out.set_weights([out.get_weights()[0].reshape(-1, 1, 512, num_classes), out.
    get_weights()[1]])
```

Listing 6.3: Keras Implementation of setting the trainable parameters of the FCN model to the trained parameters of the Dense-FFN model.

The `get_weights()` function of a layer, returns its trainable parameters. The first index returns the weights which need to be reshaped such that they fit their respective convolution sizes. The second index returns the biases which can be directly transfered without any reshaping.

Once this is done we can evaluate our FCN model on the training and test datasets used to train the Dense-FFN model and we indeed get the exact same accuracy values as in Table 5.2, confirming that our conversion is correct and that the networks are equivalent.

6.3 Fine-Tuning on Sequences

Up until this point we have not increased the receptive field size of our model, yet. Before we do this, we first try to fine-tune our convolutionzed model on sequences to see if that already makes a difference. We do that by keeping the Dense-FFN weights, which were trained on single-frames, and doing some additional training on the whole sequences. Note that training on the whole sequence does not change the receptive field size of the network. The receptive field size of the network is determined by the receptive field sizes of the convolution kernels within the network. Since their widths in this network are all 1 (see Tab. 6.2), the receptive field size is 1 frame, as the network processes one frame at a time to compute one frame label. A more detailed calculation of receptive field size is described later in Section 7.3.

6.4 Evaluation

We fine-tune our FCN-model on the same datasets used in Section 5.2 but without balancing the frames. Instead, we give the whole sequence to the network and the corresponding label sequence. For each dataset we use the weights from the corresponding pre-trained Dense-FFN model and fine-tune it with the 72 *walking* training sequences for the *Walk x-labels* experiments and the 60 *mixed* training sequences for the *Mixed x-labels* experiments.

6.4.1 Implementation Details

We again use the Adam [66] optimizer for training with the same settings and 100 training epochs. The experiments are carried out using Keras [15] with a Tensorflow [1] back-end using a cross-entropy loss.

6.4.2 Results

	Walk 3-l.	*Walk 7-l.*	*Mixed 3-l.*	*Mixed 5-l.*	*Mixed 6-l.*	*Mixed 10-l.*
Train D.-FFN	**97.53%**	**95.36%**	86.78%	**90.95%**	**89.38%**	**88.16%**
Train FCN	93.35%	92.79%	**91.48%**	78.73%	77.93%	72.96%
Test D.-FFN	**93.22%**	62.86%	58.62%	69.32%	74.30%	65.87%
Test FCN	92.40%	**91.08%**	**91.29%**	**79.20%**	**76.28%**	**68.68%**

TABLE 6.3: Training and testing accuracies on different datasets for our FCN model (2nd and 4th row) and Dense-FFN model (1st and 3rd row) for comparison.

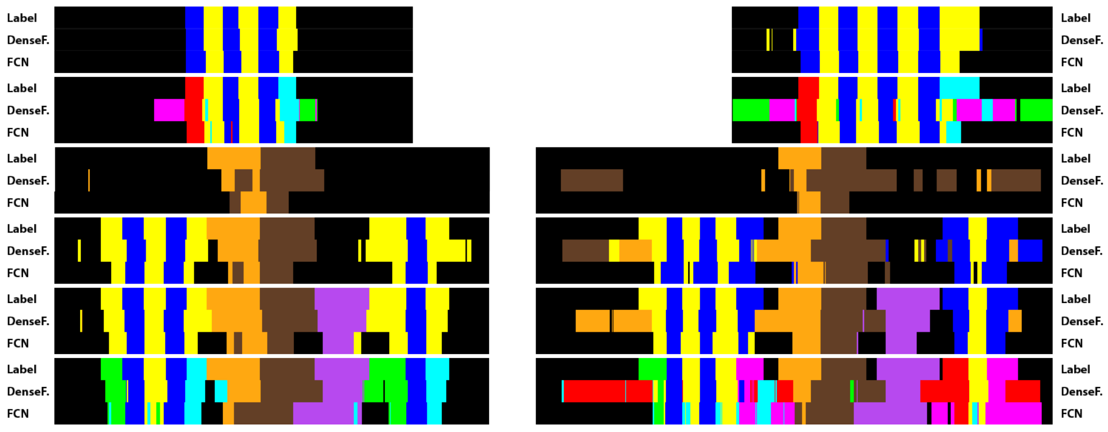


FIGURE 6.6: One training (*left*) and testing (*right*) example for each label granularity. The top third in each example sequence is the ground truth, the second third the prediction from Dense-FFN and the third the prediction from FCN. Examples show "typical" performance across dataset. Label granularity from *top* to *bottom*: *Walk 3-l.*, *Walk 7-l.*, *Mixed 3-l.*, *Mixed 5-l.*, *Mixed 5-l.*, *Mixed 6-l.*, *Mixed 10-l.* For a better comparison the same train and test sequence from the *Walking* dataset and *Mixed* sets have been used with their different label granularities.

From the results in Table 6.3 we see that the training accuracies are worse in almost every experiment except *Mixed 3-labels* for our FCN model compared to the Dense-FFN model. However, the test accuracies have improved almost everywhere or are at least comparable to the Dense-FFN's. This implies that the convolutionization of the layers also had some regularizing effect. In *Walk 7-labels* and *Mixed 3-labels* we even see an improvement of the test accuracies of around 30%. Despite our FCN having the same number of parameters and the same receptive field size as our Dense-FFN, the results have improved in almost every experiment or are at least as good. In the *Walk 7-labels* experiment the FCN model is able to segment some test sequences almost perfectly while others still have some over-segmentation artifacts (see Fig. 6.6 *second row*). This suggests that even a receptive field size of just one frame may be enough to recognize temporal relationships between frames to some extend, when the classifier is made to optimize across all frames within a sequence at once. This may also be the reason why we obtain a regularizing influence as a side effect. Additionally, these results show that when using a fully-convolutional network, a prior balancing of data is not needed.

While doing the experiments, we noticed, that segments, that were misclassified by the observer (even in the training examples), the network was able to correct them (Fig. 6.7). We believe this is due to the network computing a distribution of all training samples and being able to down-weight these misclassifications automatically as outliers. This feature makes it somewhat robust toward human biases and errors. In Chapter 7 this robustness is further tested and compared in other benchmark models.

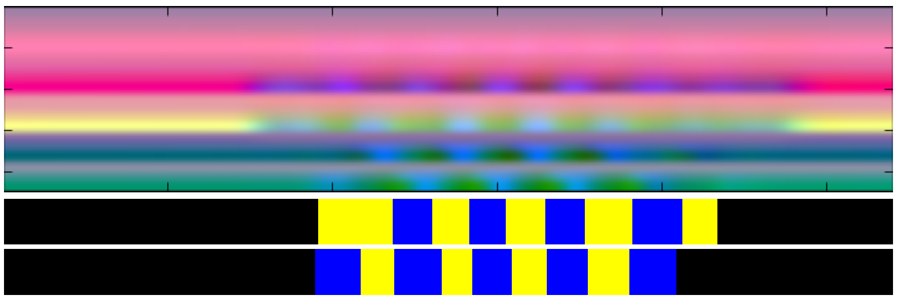


FIGURE 6.7: Corrected training example. From *top* to *bottom*: Motion image from a walking sequence, training annotation, corrected predicted labels. Notice how the network correctly identifies the first right step, although it has been trained on the faulty example.

6.5 Conclusion

In this chapter we have transformed a neural network with just dense layers that can only take a single frame at a time to a fully-convolutional classifier which can take an arbitrarily sized input. Furthermore, the convolutionization of the dense layers makes it possible to test various different receptive field sizes without having to create different datasets for each receptive field size, i.e. dividing the original sequences such that they have 1, 3 or e.g. 10 frames at a time. Although we have used the same receptive field size and number of parameters as our Dense-FFN, we still achieve superior results. Furthermore, the network does not classify the whole sequence as just the majority class anymore even though we are dealing with an unbalanced number of frames per class. We do not even need to adjust the loss function such that it would weigh classes with more frames less compared to classes which only have a couple of frames, as is described in [86].

This may be because now the optimizer has to minimize the loss function for all frames within one sequence at once which may enforce that the network is less likely to be stuck in an inferior local minimum. This may also be the reason why despite having a receptive field size of one frame, the network is still able to recognize some temporal relationships between frames, e.g. in the *Walk 7-labels* experiments.

Chapter 7

Dilated Temporal Fully-Convolutional Network

In this chapter we describe how to increase the receptive field size of our fully-convolutional model by increasing the widths of the convolutions, as well as the number of layers. We also introduce dilated temporal convolutions to further increase the receptive field of our network, and refer to the presented architecture as dilated temporal fully-convolutional network (DT-FCN).

7.1 Inspiration

Before we start defining our network architecture, we highlight the nature of semantic segmentation and sequence modeling tasks for images and time dependent data to discuss similarities and differences between our proposed network and previous models [7, 86] that have tackled similar problems, and our inspiration behind our proposed architecture in this chapter.

7.1.1 Semantic Segmentation of Images

Semantic segmentation of images refers to the dense labeling of pixel data, where each pixel value is assigned a distinct label, depending on structural features in spatial neighborhood regions. Formally speaking, such a model produces a mapping of the form

$$f : X \to Y \tag{7.1}$$

where $X(i,j) \in \mathbb{R}^{H \times W \times D}$ and $Y(i,j) \in \mathbb{R}^{H \times W \times C}$. H represents the height of the image and W the width. The number of channels is given by D, while the number of classes is represented by C. For each pixel (i,j) in Y, a vector of length C is computed with the probabilities of each class defining that pixel. The final pixel labels are then often given by $c(i,j) = \operatorname{argmax} Y(i,j)$. Such labeling does not only provide a segmented output but also a semantic meaning for each pixel.

Fully-Convolutional Neural Networks

Long et al. [86] have popularized the idea of using fully-convolutional neural networks for such a semantic segmentation of images. In contrast to traditional CNNs that do not extend to arbitrary-sized inputs, due to the use of dense layers, FCNs convolutionize such dense layers to compute a probability vector per "dense layer convolution", as shown in the previous chapter. While convolutionizing may extend a CNN with dense layers to arbitrary-sized inputs, it does not automatically give a dense labeling. In fact, due to the heavy sub-sampling in common CNN architectures [46, 69, 118, 125] because of pooling, the semantic predictions are relatively coarse (Fig. 7.1 bottom left FCN32s prediction).

Long et al. overcome this problem by up-sampling and using *skip* connections [86] (also known as *residual layers* [46]). Such connections incorporate feature maps from prior layers in the final output which results in a more fine-grain output of the same dimensions as the input (Fig. 7.1 top).

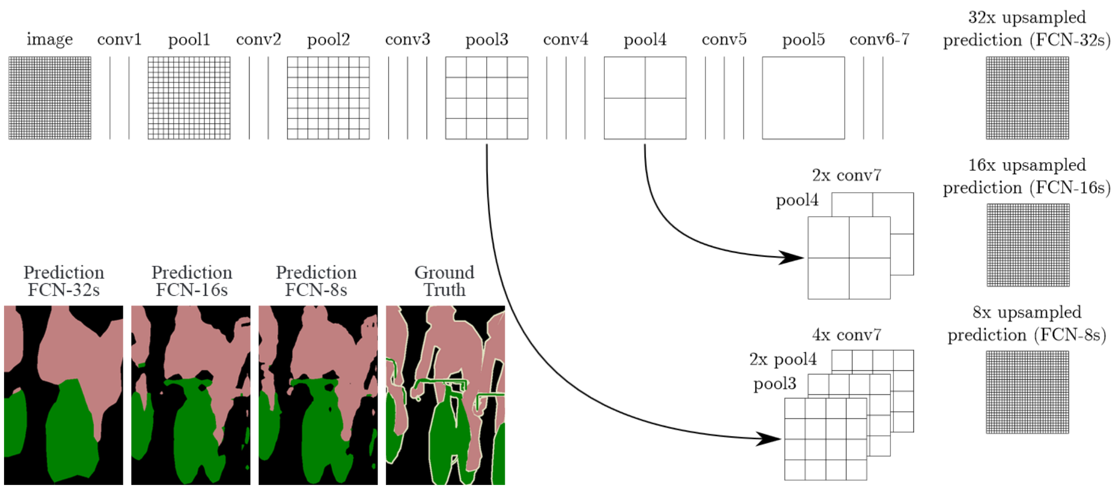
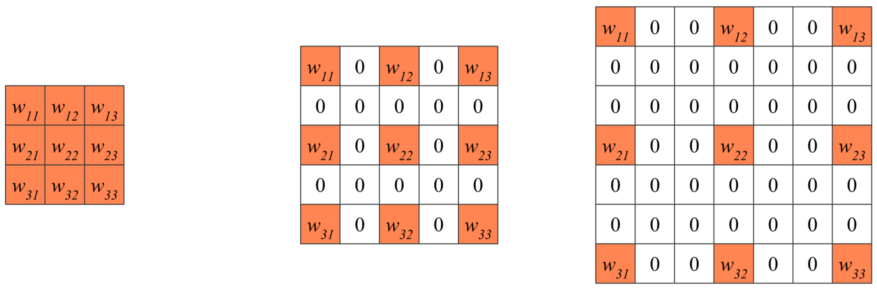


FIGURE 7.1: The residual connections combine high-layer information with fine, low layer information. Pooling and prediction layers are shown as grids that reveal relative spatial coarseness, while intermediate layers are shown as vertical lines. *First row (FCN-32s):* VGG-16 [118] single stream net without any skip connections. Its coarse prediction is shown on the bottom left. *Second row (FCN-16s):* Combining predictions from the final and *pool4* layer lets the network predict finer details (see second image bottom left). *Third row (FCN-8s):* Additional predictions from *pool3* provides precision (see third image bottom left). *Image source:* [86]

Yu and Koltun [135] use a different and more sophisticated approach to overcome this problem. They make use of *dilated* convolutions (also known as convolution with holes), which we later explain in more detail. Yu and Koltun *dilate* the convolution kernels of the pre-trained VGG-16 model [118], which has been trained on image classification [20], by effectively "filling them with zeros" between the original kernel values (see Fig. 7.2).



w_{11} w_{12} w_{13}
w_{21} w_{22} w_{23}
w_{31} w_{32} w_{33}
0

FIGURE 7.2: A 3×3 filter with dilation. *Left:* A dilated rate d of 1 is used. A "normal" convolution kernel is thus a special case of a dilated convolution with $d = 1$. *Middle:* A 3×3 filter with dilation rate $d = 2$. *Right:* A 3×3 filter with $d = 3$.

This "trick" is also referred to as *shift-and-stitch* in [86]. The effect of dilating the convolution kernels in a network is similar to *shifting* the input image of the un-dilated network multiple times. Each time it is shifted, a new segmentation result is computed. In the end multiple such segmentation results are obtained together with their corresponding shift vectors. These segmented output images with the shift vectors can be *stitched* together to get a better resolution in the final semantic segmentation. A comparison between Long et al. [86] and Yu et al. [135] segmentation results is illustrated in Fig. 7.3. Note how the results of Yu et al. [135] are superior.

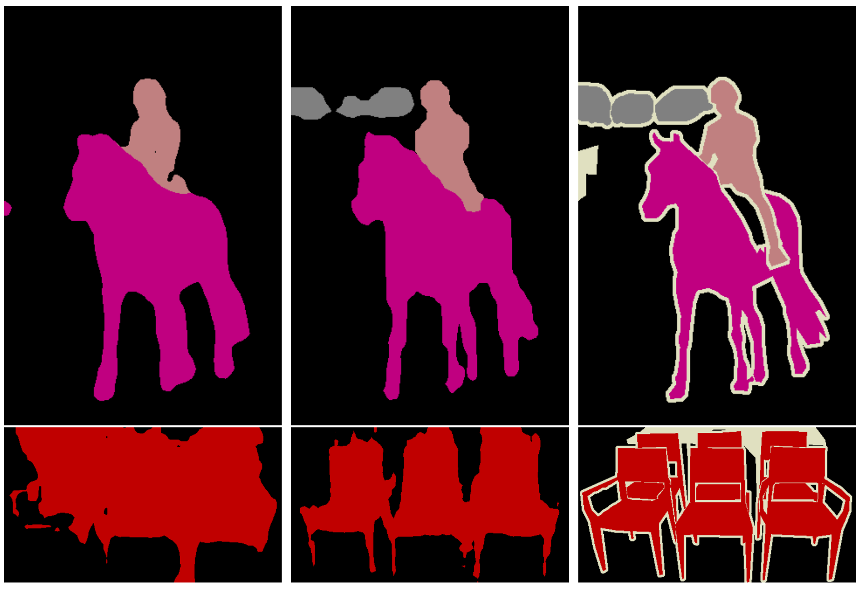


FIGURE 7.3: *Left column:* Segmentation results of Long et al. [86]. *Middle column:* Segmentation results of Yu et al. [135]. *Right column:* Ground truth of Pascal VOC-2012 [24] challenge.

7.1.2 Sequence Modeling

In sequence modeling tasks, we are given an input sequence

$$x_0, x_1, ..., x_{t-1}, x_t \tag{7.2}$$

and want to *predict* some corresponding output sequence

$$y_0, y_1, ..., y_{t-1}, y_t \tag{7.3}$$

Usually, this is under the constraint that an output y_t can only be predicted using samples that have been observed before time t, i.e. samples at times $0, ..., t-1$. Thus, our mapping function in this case is

$$f : x(t-A, ..., t-1) \rightarrow y(t) \tag{7.4}$$

where $A \geq 1$ for an output y at a given time t.

In recent years, such sequence modeling tasks have been associated with recurrent architectures of neural networks only. Goodfellow, Ng and other well established deep learning researchers focus almost exclusively on these architectures in their popular books [40] and lectures [99], when it comes to sequential data.

On the other hand, recent research [7] suggests that certain types of convolutional architectures, reach state-of-the-art results in various sequential tasks, such as audio synthesis, machine translation or language modeling [7, 19, 36, 61, 129]. Such convolutional neural architectures that deal with sequential, time-dependent data, have been coined the term *temporal convolutional networks* (TCN) [7, 74, 75]. To our knowledge, Lea et al. [74] were the first ones to use this term. Later, Bai et al. published an extensive description of the features of such networks in their study [7]. Hence, we will go by their definition of a TCN in this thesis.

Temporal Convolutional Neural Networks

According to Bai et al. [7], the two distinguishing characteristics of TCNs are:

1. The architecture can take a sequence of any given length and map it to another sequence of the same length.
2. There is no *leakage* from the future to the past, i.e. the mapping has to be *causal*, such that a future output at time t can only depend on past inputs at times $0, ..., t-1$.

The first characteristic is obtained by using one-dimensional convolution layers with zero padding of length *kernel size* - 1, so that subsequent layers have the same lengths as previous ones. With this no skip or even dilation layers are required to ensure a dense labeled output. The second characteristic is ensured by the use of causal convolutions. These convolve an input X at each time step t from X_{t-w} to X_t, where w is the length of the filter, while acausal convolutions convolve it from $X_{t-\frac{w}{2}}$ to $X_{t+\frac{w}{2}}$. The idea of causal convolutions is based on the causal nature of sequence modeling tasks that an output y_t can only be predicted using samples that have been observed before time t, e.g. when doing time-series predictions.

7.1.3 Temporal Semantic Segmentation of Motion Images

While image segmentation is based on segmenting *spatially* similar regions, the idea of sequence segmentation is to extract structurally similar regions in the same *temporal* neighborhood. Semantic segmentation adds a higher level meaning to each of these segments by using a semantic label to each region. Hence, semantic segmentation of sequences can be considered a dense labeling of time steps. Since one of our goals is to be able to distinguish between motions like *left step* (step while walking) from *begin* and *end left step* (step from/to standing position), the network needs to be able to look into the future and past. Thus, given an input sequence

$$x_0, x_1, ..., x_t, ..., x_T \tag{7.5}$$

our mapping function can be considered to be

$$f : x(t - A, ..., t, ..., t + B) \rightarrow y(t) \tag{7.6}$$

for an output y at time t with suitable adjustments to the borders. Due to this we use *acausal* convolutions, in contrast to the previously mentioned TCN architecture. Since our goal is to semantically segment our motion images, the mapping function is further extended in its input:

$$f : X \rightarrow y(0, ..., T) \tag{7.7}$$

with $X \in \mathbb{R}^{N \times M \times 3}$, similar to the mapping in image segmentation tasks (Section 7.1.1).

Hence, we propose an acausal architecture, similar to previous FCN models, with the difference that it filters the input in the temporal dimension only (akin to previous TCN architectures). To ensure dense labeling and an output of the same length as the input, we also zero-pad each hidden layer, as described in [7]. This makes our model much simpler compared to Long et al. [86] because we do not necessarily need to make use of skip connections for a dense labeled output.

7.2 Temporal Convolutions

One-dimensional convolutions are considered temporal convolutions since they only span in one spatial (or in this case *temporal*) dimension. Usually, the input of such convolutions is also one-dimensional, with multiple feature-maps.

7.2.1 Temporal 2D Convolution

As our data consists of 2D RGB images, where one dimension represents the number of frames, the other represents the spatial properties and relations of the joints to each other and the third the 3D joint positions, we introduce a 2D convolution, which we apply in the direction of the time dimension only:

$$I \star K = \sum_{h=0}^{H-1} \sum_{W=0}^{W-1} \sum_{d=0}^{3-1} I(h, w, d) \cdot K(h, w + i, d) \tag{7.8}$$

Where $I(h, w, d)$ is the input image in $\mathbb{R}^{H \times W \times 3}$ and $K(h, k, d)$ the filter kernel in $\mathbb{R}^{H \times K \times 3}$. To enforce a temporal 2D convolution of arbitrary filter widths in deep-learning frameworks, we use a filter height, which is the same as the input image height *and* add a two-dimensional zero-padding of size *image height* $\times \lfloor \frac{kernel\ width}{2} \rfloor$ on each side of the temporal dimension. A special case of this type of convolution with a kernel width of 1 has already been described in Section 6.1.2.

7.3 Receptive Field Size

Since motion features can span many frames, the filter needs to be able to look far into the future and into the past. For example, when distinguishing between motions like *left step* from *begin* and *end left step*, there are often many time steps between beginning and ending of a full step. Thus, our network needs a large receptive field.

There are different ways to increase the receptive field size of a network to take more frames into account. In this section we show this by using examples with one-dimensional convolution kernels, however these methods generalize over multiple dimensions.

7.3.1 Kernel Size

The most obvious one is increasing the kernel size k:

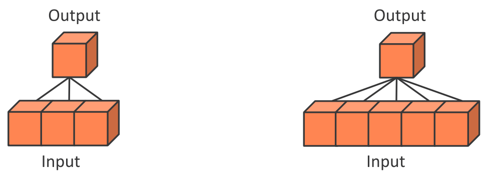


FIGURE 7.4: Increasing kernel size increases the receptive field of the convolution. *Left:* A kernel of size 3. *Right:* A kernel of size 5.

Fig. 7.4 illustrates this. The kernel which is able to take a larger number of frames into account has a bigger receptive field. The receptive field r in this case, is calculated the following way:

$$r = k \tag{7.9}$$

r is directly equal to the kernel size of the convolution filter.

7.3.2 Stacking Layers

The next technique one can do is stacking up multiple layers with kernel sizes greater than one to increase the receptive field size. Fig. 7.5 shows two convolution kernels of size 3 have the same receptive field size as a convolution kernel with size 5.

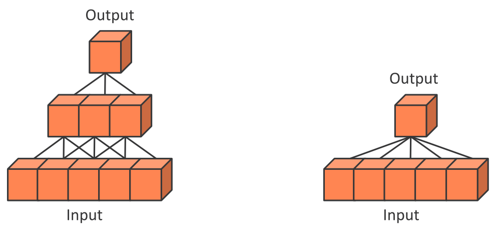


FIGURE 7.5: Two stacked convolutions of width 3 (*left*) have the same receptive field size as a convolution of width 5 (*right*).

The receptive field size r_l after each layer l is in this case

$$r_l = 1 + \sum_{l}^{L}(k_l - 1) \tag{7.10}$$

where k_l is the kernel size at layer l. When k_l is equal in all layers, r_l increases linearly with each layer.

7.3.3 Dilated Convolutions

A normal CNN is only able to look at a window linear to the depth of the network. This requires many layers and parameters to train. Following the work of van den Oord et al. [129], we apply dilated convolutions that enable exceptionally large receptive field sizes [135]. Such a convolution filter is essentially the equivalent to a larger filter, which has been derived from the original filter, dilated with zeros, but is significantly more efficient. The dilation allows the filter to operate on a coarser scale than a normal convolution. For 1D sequence $\mathbf{x} \in \mathbb{R}^N$ and filter $f : \{0, ..., k-1\} \to \mathbb{R}$ such a convolution is formally defined as:

$$(\mathbf{x} *_d f)(t) = \sum_{i=0}^{k-1} f(i) \cdot \mathbf{x}_{s-d\cdot i} \tag{7.11}$$

where d is the dilation and k the filter length. A normal convolution thus can be seen as a special case of a dilated convolution with $d = 1$. Fig. 7.6 shows how systematic dilation increases the receptive field size exponentially in depth with comparably few layers and parameters. A receptive field size of 15 time steps is already obtained after three layers using a filter width of three. For a non-dilated convolution with the same width, we would need seven layers and more than double the parameters to achieve the same receptive field size.

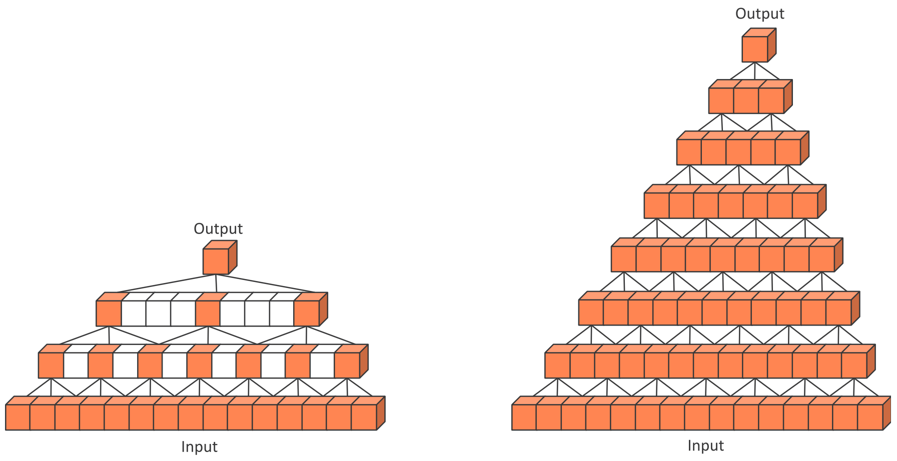


Figure 7.6: Dilated (*left*) and undilated (*right*) convolution. Systematic dilation increases the receptive field size exponentially with depth, with considerably fewer layers and thus parameters to be trained.

The receptive field size r_l after each layer l is now given by

$$r_l = 1 + \sum_{l}^{L} (k_l - 1) \cdot d_l \tag{7.12}$$

where d_l is the dilation rate at layer l.

7.4 Network Architecture

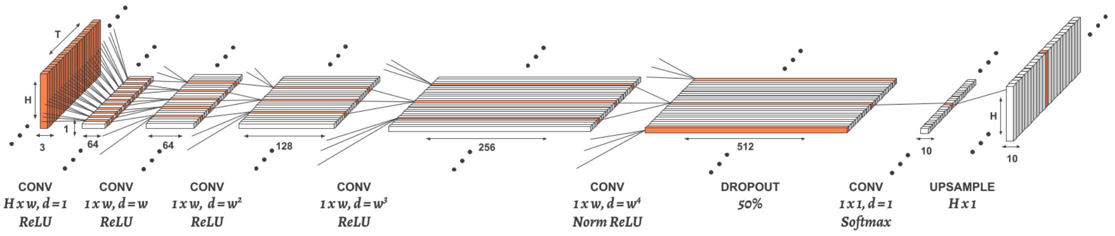


Figure 7.7: Our dilated temporal fully-convolutional neural network (DT-FCN) for motion capture segmentation. The initial layer consists of a traditional 2D convolutional layer, applied in the time dimension. The next layers are 1D temporal acausal convolutions with dilation. The final layer is a convolutionized dense-layer with a Softmax activation function.

Our model has a total of five convolution layers. To process RGB-image data, the initial one consists of an acausal temporal 2D convolution, followed by four subsequent layers of 1D temporal dilated acausal convolutions. An overview of our architecture can be seen in Fig. 7.7. Similar to [46, 118], we use the same width w for each convolution layer with a stride of 1. We use this parameter to adjust the receptive field size of the network later on by conducting different experiments.

The dilation rate d in our model increases with each layer l, according to $d = w^{l-1}$, where w is the kernel width. This makes our dilation rate larger than prior temporal models [7, 72, 129], which generally use a dilation rate of $d = 2^{l-1}$ and do not account for filter width. As shown in Fig. 7.8, this ensures that an output per receptive field size depends on every input within the receptive field (i.e. no frames within the receptive field are skipped) while a minimum of parameters is used.

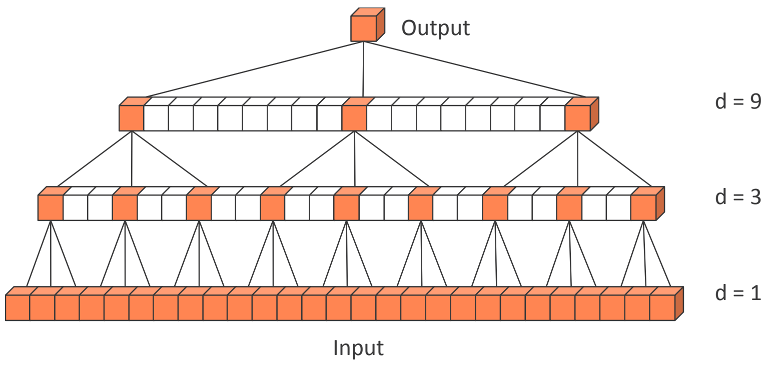


FIGURE 7.8: A receptive field size of 27 is reached with $w = 3$ and three layers using a dilation rate of $d = w^{l-1}$, compared to the receptive field size of 15 of the filter in Fig. 7.6 (*left*) that uses a dilation rate of $d = 2^{l-1}$.

A 50% Dropout is added during training time before the final convolutionized dense layer, which helps against over-fitting as described in Chapter 3. The convolutinzed dense-layer with a Softmax activation function is added to reduce the dimensionality in feature space without changing the spatial dimensions. Finally, we upsample the output for a better visualization.

We make use of the ReLU activation function in all of our layers (except the final bottle-neck layer) since they have proven to be more efficient and easier to train than Sigmoid or Tanh layers [39, 43]. However, since our initial input values are between 0 and 255 and we only have ReLU non-linearities in between, it could very well happen that the input lies outside of the area of the non-vanishing slope of the final Softmax classification layer. To ensure that the values created by the ReLU activation layers before the Softmax function do not exceed reasonable input values, we normalize the output of the last ReLU activation using the following function:

$$NormReLU(x) = \frac{ReLU(x)}{\max(ReLU(x)) + \epsilon} \tag{7.13}$$

With $\max(x)$ being the maximum value of the input tensor x. We use a value of $\epsilon = 1e^{-5}$ and found that this greatly improves accuracy.

7.5 Evaluation

This section describes various experiments and tests carried out to evaluate the performance of the presented motion data segmentation approach. We compare our model against a popular

RNN-method for sequence modeling [41] and two state-of-the-art TCN-based action segmentation models [72, 74]. We evaluate them on the finest label granularity level *Mixed 10-labels*. Furthermore, Bouchard et al. [12] have shown that a single person determines the segmentation boundaries within the same sequence differently every time they segment it. In addition to that, human annotators can become tired very easily, which introduces more segmentation errors and "randomness" in the segmentation result. To test how robust the models are against errors due to wrongly-classified labels, we further train them on noisy training labels and test them on the true test labels. Finally, to see how generalizable the models are, we test them on the walking dataset with *Walk 7-labels*, while they were trained on the mixed action dataset with *Mixed 10-labels*.

Random Noise Injection

Our initial noise experiment is simple in nature but already gives us surprising results. In this experiment we set a certain percentage of frames within one motion sequence to a random label of the given 10 labels, as shown in Fig. 7.9. We add noise in 10% intervals. We start from 0% noise and gradually increase the noise level by additional 10% until we reach 100%. We use 100% noise as ground truth with an expected accuracy of ~10% across all models.

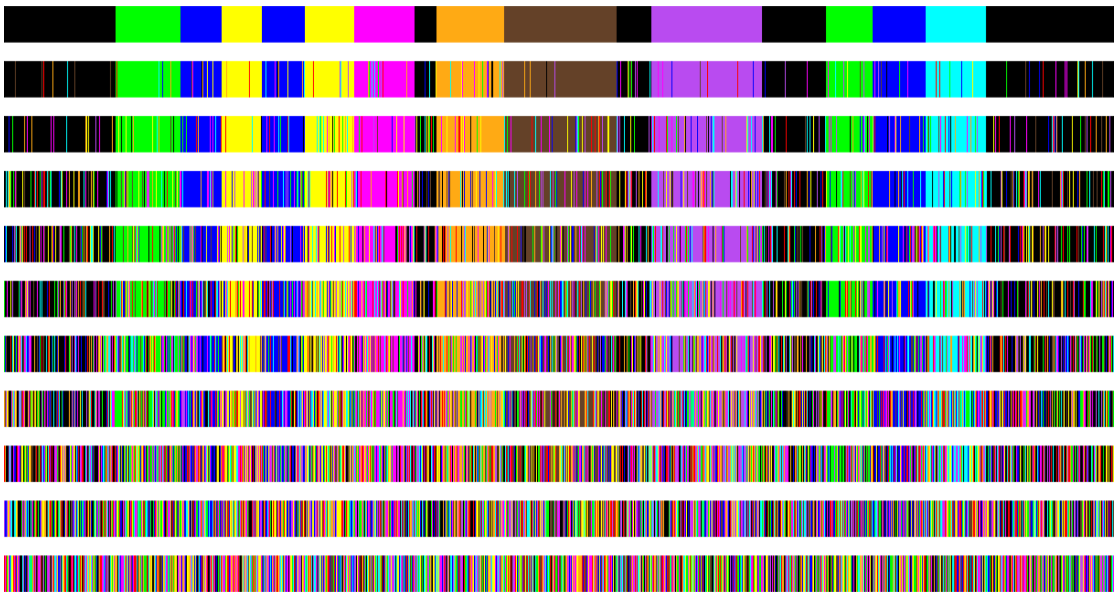


FIGURE 7.9: Training labels of a sample training sequence with random noise injection. Percentage of noise from top to bottom: 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%

Noisy Boundary Regions

While the data set with random noise injection gives us some evaluation form for robustness against noise, it is not very applicable for "real-world noise" in training labels. In reality, annotation disagreements usually happen around segment boundaries. Bouchard et al. [12] have shown that a single person determines the segmentation boundaries within the same sequence differently every time they segment it. The median of each segment boundary was computed and then the distance of each boundary from the median was computed and graphed as a histogram. The result is a standard deviation of 15.4 frames, or about $\frac{1}{2}$ seconds. Due to these inter- and intra-annotator disagreements, we adapt our initial data set (without any noise) and insert wrong information, i.e. noise, in a window around a segment boundary frame (Fig. 7.10). A boundary frame is determined by using the first frame of a new segment within a sequence. The width of the noise windows is set to $w_{noise} = 2n + 1$. Within this window we set the frames to a random label. We chose $n = \{5, 10, 15\}$ frames. The resulting training sequence labels are illustrated in Fig. 7.11.

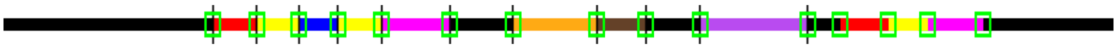


FIGURE 7.10: Boundary noise windows (highlighted with green edges). The line between two segments indicates a boundary frame. Frames within the green windows are set to random labels.

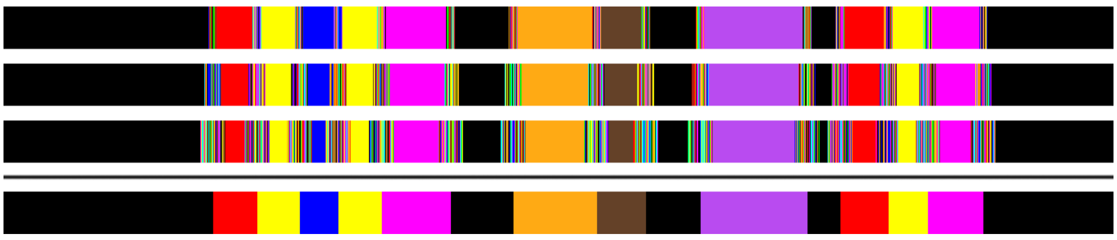


FIGURE 7.11: Boundary noise labels for an example training sequence. From top to bottom: $w_{noise} = 11$ frames, $w_{noise} = 21$ frames, $w_{noise} = 31$ frames, original sequence segmentation.

No Boundary Information

Additionally, we conduct an experiment, which is less "harsh" in nature: Instead of giving wrong information in the transition regions, we give no information to the networks. Essentially, leaving the models to their own interpretation of what to make of the frames within the transition windows. We do this by setting the loss within the boundary regions to zero. Hence, the networks should not be affected by the information in the boundary regions.

7.5.1 Benchmark Models

The following three benchmark models are used for comparison:

Bidirectional LSTM. LSTMs are a widely used method in sequence modeling tasks [7, 48]. They have been the go-to method for sequential data and have been successfully used in action segmentation tasks [22, 41, 119]. For these experiments we use a more sophisticated LSTM architecture which is also able to take information from future samples, called a bidirectional LSTM which has been used for action recognition [41, 119]. Fig. 7.12 illustrates the model. LSTM's can theoretically have an infinite "receptive field size".

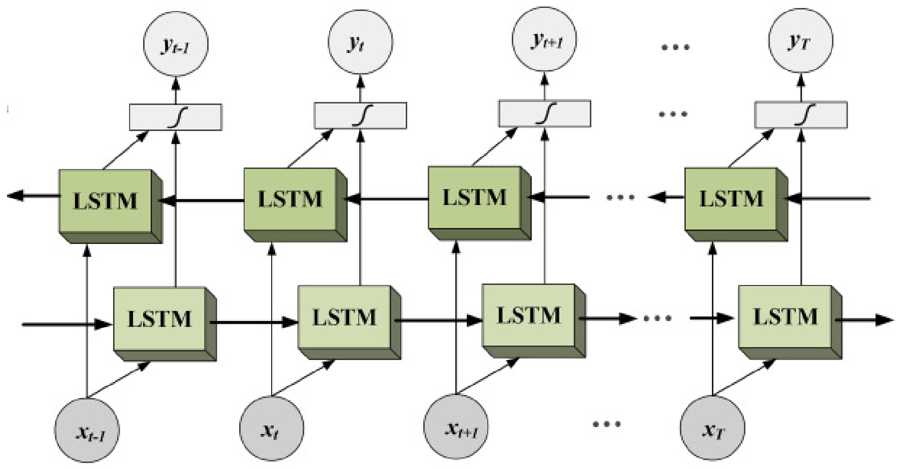


FIGURE 7.12: A bidirectional LSTM architecture. The outputs of the forward and backward layers are concatenated and a Softmax activation function is applied. *Image source:* [134]

Encoder-Decoder TCN. The next model we compare against is also an acausal TCN method, similar to ours. However, it has an encoder-decoder nature to it and does not use dilated convolutions. It is proposed by Lea et al. [74, 75] as a state-of-art action segmentation method in videos. We adapt their network for motion image segmentation. Fig. 7.13 shows an overview of their Encoder-Decoder TCN.

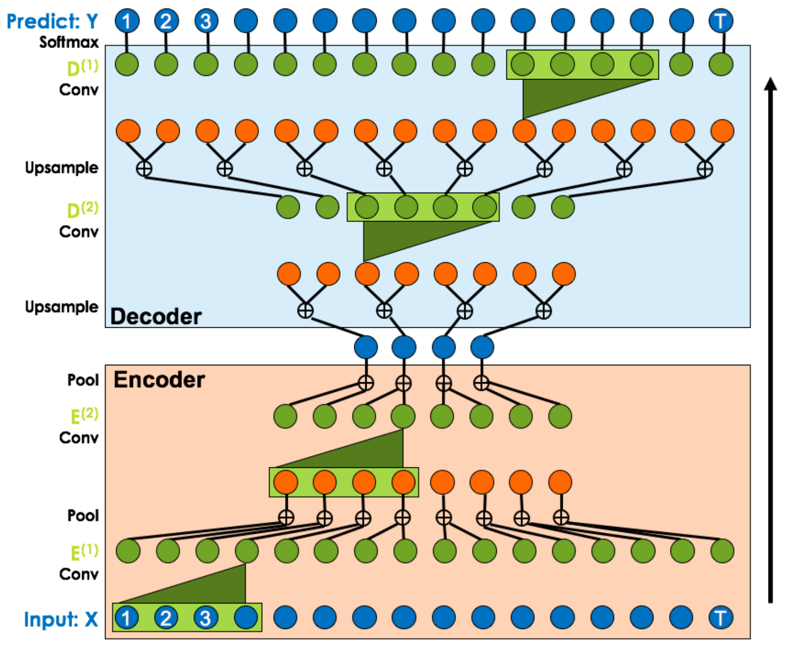


FIGURE 7.13: The Encoder-Decoder TCN (ED-TCN) network by Lea et al. *Image source:* [72]

First the input is filtered using a convolutional layer and then subsampled with a max-pooling layer to obtain the most important information. This is repeated until the decoding layers are reached. The decoding layers reverse the process by upsampling the subsampled features and performing a convolution on the upsampled output. Lea et al. use two encoding and two decoding layers with convolution kernels of width $w = 25$ each time. Each having a convolution and the respective upsampling or downsampling operation. The total receptive field size is 49 frames for the entire network.

WaveNet/TCN/DilatedTCN. WaveNet has originally been developed for audio synthesis by van den Oord et al. [129]. It is a causal dilated TCN architecture, which has also been used in various other sequence modeling tasks by Bai et al. [7], referring to this architecture simply as "TCN". Lea et al. [72] have adapted this architecture for the task of action segmentation in videos and called it "DilatedTCN". We make use of their hyper-parameter suggestions in terms of number of blocks and layers within these blocks, and hence refer to this architecture as DilatedTCN in the next sections.

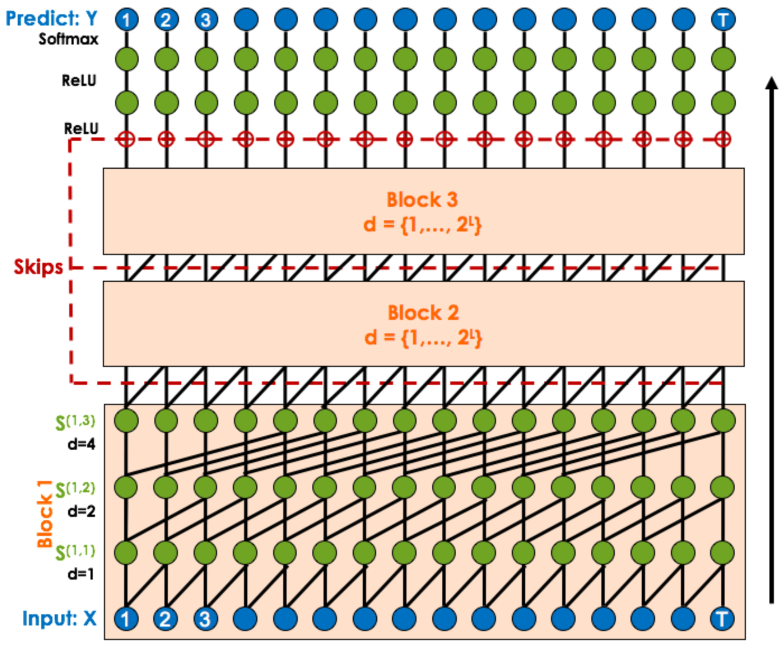


FIGURE 7.14: The DilatedTCN structure as used by Lea et al. *Image source:* [72]

Lea et al. [72] adaption is illustrated in Fig. 7.14. Each of the one-dimensional causal convolution layers has a dilation rate of $d = 2^l$ within one block. Furthermore, skip connections are applied between each convolution layer. Every block of three convolutions is then connected with additional skip connections. Lea et al. use 1 block with 4 layers and 3 blocks with 6 layers each. Their total receptive field is 254 frames.

7.5.2 Implementation Details

The parameters of our model are learned using the categorical cross-entropy loss with Stochastic Gradient Descent and the Adam optimizer [66]. We use the default settings of $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$ as described in the paper. Our model is implemented using Keras [15] with a Tensorflow [1] back-end. In all of the following experiments, we use non-randomized 7-fold cross-validation and 100 training epochs.

7.5.3 Evaluation of Receptive Field Size

In order to determine the optimal receptive field size (RFS) for our model, we test our described architecture on different convolution kernel widths w. For this we choose a model that just uses a single frame for its evaluation ($w = 1$, RFS = 1 frames), one that has a RFS of local structures in a sequence ($w = 3$, RFS = 243 frames, see Fig. 7.15) and one that is bigger than the longest sequence in our dataset ($w = 5$, RFS = 3125 frames). Table 7.1 shows the different RFS after each layer for the three versions that we evaluate of the architecture described in Section 7.4. The receptive field size RFS_L after each layer L is calculated by

$$\text{RFS}_L = 1 + \sum_{l}^{L} (w_l - 1) \cdot d_l \tag{7.14}$$

where d_l is the dilation rate and w_l the convolution kernel width at layer l.

	$w = 1$			$w = 3$			$w = 5$		
L	d	p	RFS	d	p	RFS	d	p	RFS
1	1	1	1	1	2	3	1	4	5
2	1	1	1	3	6	9	5	20	25
3	1	1	1	9	18	27	25	100	125
4	1	1	1	27	54	81	125	500	625
5	1	1	1	81	162	243	625	2500	3125
$\#P$	225,290			663,562			1,101,834		

Table 7.1: Model architectures and their receptive field sizes (RFS) in frames after each layer L for our proposed architecture. w is the kernel width, d the dilation rate and p the padding. $\#P$ describes the number of parameters in each model.

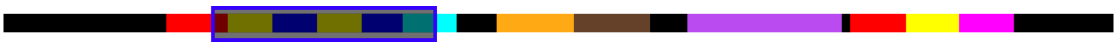


Figure 7.15: Receptive field size of 243 frames highlighted on a 1193 frame sequence.

We evaluate these three models on the *Mixed 10-labels* granularity with random noise injection. Fig. 7.16 shows that even though a width $w = 5$ (RFS: 3125 frames) covers the entire sequence, the test accuracy is worse on all noise levels compared to using $w = 3$ (RFS: 243 frames). This is in contrast to some claims [7, 129] in sequence modeling tasks that the receptive field of the network should at least cover the longest sequence in the data set. It supports a more intuitive suggestion that local structures (e.g. in Fig. 7.15) are enough to be able to classify important structural and temporal information. Furthermore, the increase in parameter space for $w = 5$ makes it also more prone for over-fitting, as compared to $w = 3$.

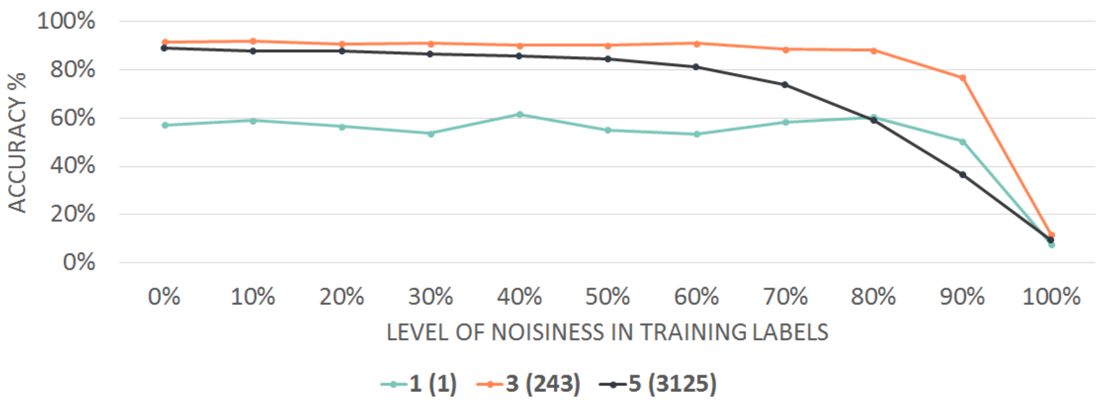


FIGURE 7.16: Test accuracies for different receptive field sizes of our proposed architecture depending on noise level. The model with RFS = 243 (*red*) constantly outperforms the others. Note how at 80% of noise in training labels still an accuracy of almost 90% in the test set is achieved. *Red*: RFS = 243, *black*: RFS = 3125, *blue*: RFS = 1.

Interestingly, Fig. 7.16 also shows that despite adding 80% noise in the training labels, an accuracy of almost 90% is reached on the true test labels for $w = 3$, i.e. a mere 20% of information is needed to achieve usable segmentation results. This is further backed by our segmentation results as seen in Fig. 7.17. Note how the segmentation result of 90% noise in training labels still is sufficient.

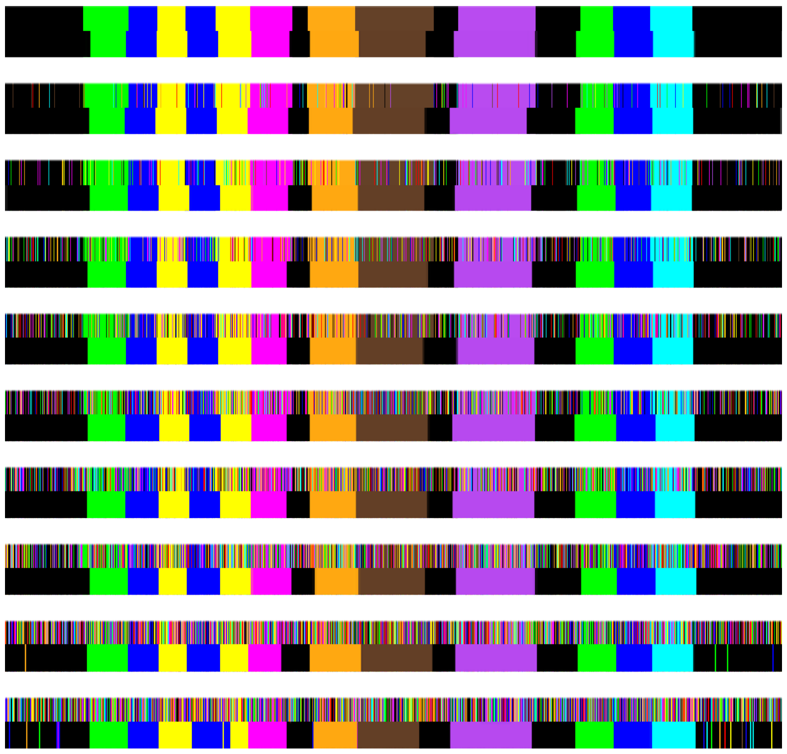


FIGURE 7.17: Segmentation results of an example training sequence of our $w = 3$ model. The noisy labels it has been trained on are on top of its respective predictions. The first result shows the result for 0% noise, while the last for 90%. Noise is added in +10% intervals. The upper part in each row is the training label and the bottom part the prediction.

7.5.4 Evaluation Against Benchmark Models

While evaluating the benchmarks against our best model ($w = 3$), we found that they performed significantly worse when the input values are between [0, 255]. Hence, we adjust the input values for the benchmark models according to their activation functions by scaling them between [-1, 1] for a fairer comparison, while keeping them between [0, 255] for our proposed models.

In all of our experiments the TCN-based methods are able to train magnitudes faster compared to the RNN-based method, due to the "embarrassingly parallel" nature of convolutions. Training takes ~1 minute for the TCN-based methods compared to ~40 minutes for Bi-LSTM for 100 epochs on a 4GB GTX970 and 16GB Intel-i7. Segmentation takes less than ~1 second for all methods.

Results on "Vanilla" *Mixed 10-labels*

	Bi-LSTM	ED-TCN	DilatedTCN	Ours ($w = 3$)
Train	86.32%	90.05%	90.64%	**95.42%**
Test	81.95%	88.69%	88.47%	**90.81%**
# Param.	378,634	1,613,770	388,874	663,562
RFS	∞	49	254	243
Causality	Acausal	Acausal	Causal	Acausal

Table 7.2: Per-frame accuracy for benchmark models and our best model ($w = 3$) on the data set without any noise in the training labels.

We first test our best model ($w = 3$) and the benchmark models on the "vanilla" labels without any noise. Fig. 7.18 shows the segmentation results for the benchmark models and our best model on a sample test sequence without any noise added into the training data. While DilatedTCN and ED-TCN have similar training and test accuracies (Tab. 7.2), the segmentation results of DilatedTCN tend to suffer more from over-segmentation, especially at the boundary frames. Bi-LSTM suffers the most from such an over-segmentation. Our model achieves similar results to ED-TCN, when trained on a clean data set (Fig. 7.18). However, it makes use of less than half of the parameters, while achieving better results in per-frame accuracy (Tab. 7.2).

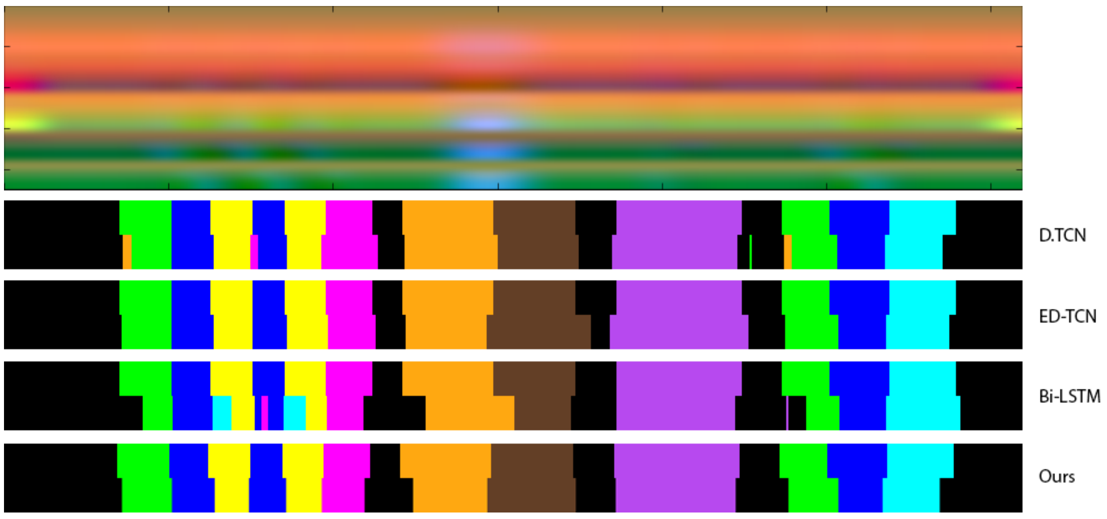


FIGURE 7.18: Segmentation results of benchmark models, as well as our best model ($w = 3$) on an example test sequence. The top half of each row is the ground truth segmentation of the depicted motion image, while the bottom half is the prediction from the respective network. Results show "typical" performance of each model across data set. Training was done without any noise added to the labels.

Results of Random Noise Injection Experiment

We now gradually increase the noise levels by 10% as described in section 7.5 in *Random Noise Injection*.

Noise in Training	Bi-LSTM	ED-TCN	DilatedTCN	Ours ($w = 3$)
0%	81.95%	88.69%	88.47%	**90.81%**
10%	83.70%	89.16%	89.37%	**90.99%**
20%	79.94%	89.46%	88.86%	**90.75%**
30%	85.02%	89.40%	88.90%	**91.20%**
40%	84.29%	89.02%	88.64%	**91.13%**
50%	81.72%	87.40%	87.93%	**91.04%**
60%	83.08%	88.65%	86.14%	**92.99%**
70%	82.16%	87.66%	79.46%	**90.11%**
80%	81.91%	80.46%	61.82%	**89.55%**
90%	71.74%	54.91%	34.99%	**81.18%**
100%	8.76%	10.39%	11.85%	11.24%

TABLE 7.3: Per-frame test accuracy for benchmark models and our best model ($w = 3$) with noise added into the training labels.

Gradually adding noise to the training data reveals that up until 60% added noise, all networks perform similar to their performance at 0% noise, as shown in Tab. 7.3. However, beyond that the performance of DilatedTCN already starts to deteriorate. ED-TCN stays robust up until

70% of added noise, while the per-frame accuracy of Bi-LSTM and our model stays on a steady level up until 80% noise. However, in contrast to Bi-LSTM our model consistently outperforms all other benchmarks on every noise level, while Bi-LSTM performs the weakest at noise levels lower than 80%. The results in Tab. 7.3 show that a mere 10% of correct information is needed for our model to perform as well as Bi-LSTM at a 0% noise level. Fig. 7.19 and 7.20 show a sample prediction of the tested models at 80% and 90% training label noise, respectively. DilatedTCN averages the output to the background class when 90% of noise in the training labels is used, while the others still obtain reasonable segmentation results. Our model struggles with over-segmentation more than the other models at 90% noise.

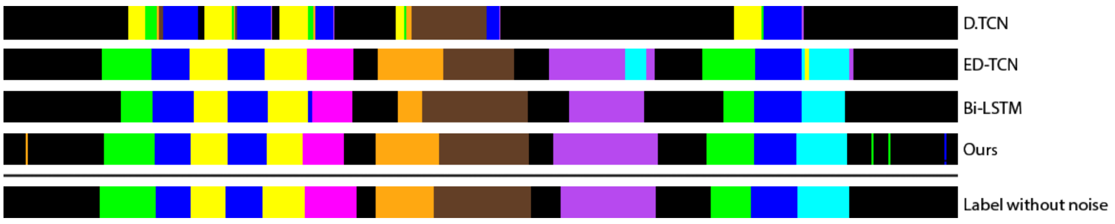


FIGURE 7.19: Segmentation results for training on 80% label noise. Bottom row shows the actual segmentation label without any noise for comparison. Example shows "typical" performance of networks across the dataset.

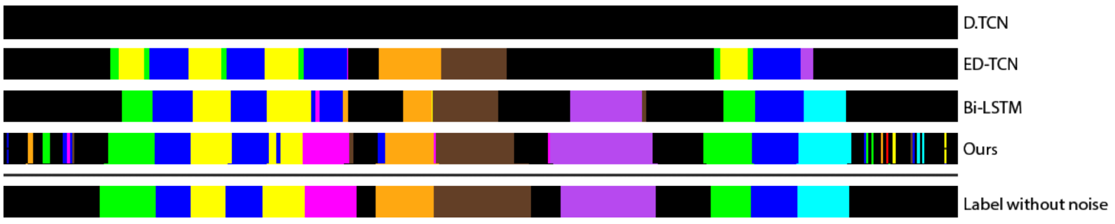


FIGURE 7.20: Segmentation results for training on 90% label noise. Bottom row shows the actual segmentation label without any noise for comparison. Example shows "typical" performance of networks across the data set.

Results of Boundary Noise Experiment

Since human annotators usually disagree the most in the boundary regions between two adjacent segments, we conduct an experiment where a window of width w_{noise} at a boundary frame contains randomly set, and hence wrong, training labels, as described in Section 7.5 *Noisy Boundary Regions*.

w_{noise}	Bi-LSTM	ED-TCN	DilatedTCN	Ours ($w = 3$)
11 frames	84.57%	88.15%	87.50%	**91.45%**
21 frames	81.47%	83.82%	85.48%	**89.03%**
31 frames	78.94%	75.75%	80.62%	**83.59%**

TABLE 7.4: Per-frame test accuracies for benchmark models and our best model ($w = 3$). Models were trained on noisy labels with random labels in the boundary regions and tested on non-noisy labels. The boundary noise window width in frames is indicated by w_{noise}.

Table 7.4 shows that despite DilatedTCN having the worst robustness performance in the previous noise experiment, it overall performs second best in terms of per-frame accuracy with $w_{noise} = \{21, 31\}$. Our network architecture still outperforms the other benchmark models, however. Fig. 7.21 shows an example training sequence, as well as the predicted segmentation by our model ($w = 3$) and benchmarks for $w_{noise} = \{11, 21, 31\}$. While the boundary prediction is almost perfect for $w_{noise} = 11$ in all models except DilatedTCN, one can see visible artifacts in the boundary regions with a noise window width of $w_{noise} = 21$ and $w_{noise} = 31$. The results show that our model still consistently performs better than the benchmarks in terms of per-frame accuracy (Tab. 7.4), as well as in terms of over-segmentation (Fig. 7.21).

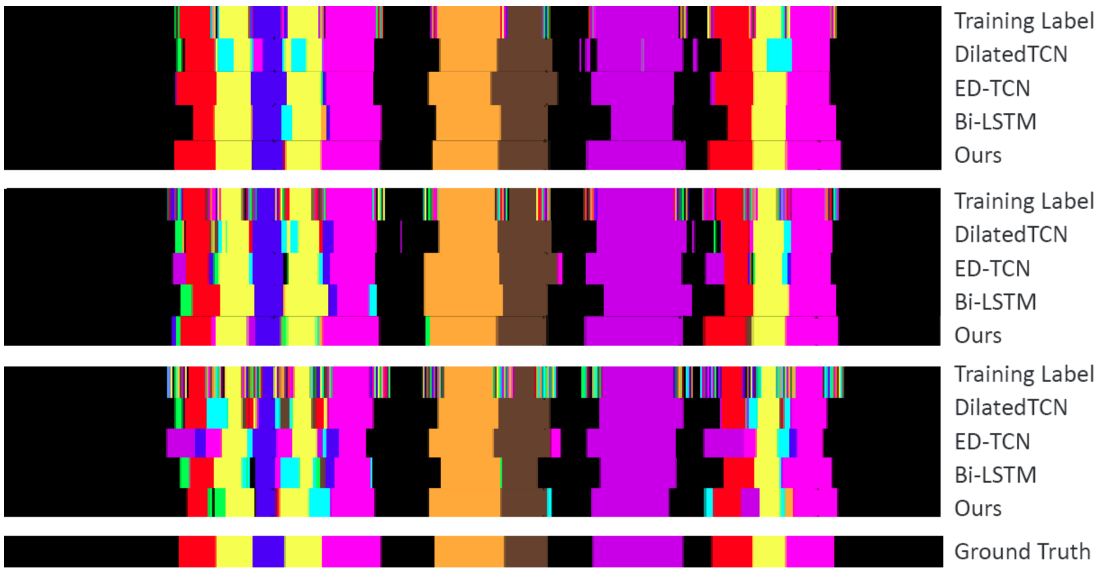


FIGURE 7.21: Predicted segmentations by our model ($w = 3$). Blocks from top to bottom: $w_{noise} = 11$ frames, $w_{noise} = 21$ frames, $w_{noise} = 31$ frames, original sequence segmentation. Each block consists of the training label, the network has been trained on and its corresponding predicted segmentation.

Results of No Boundary Information Experiment

In an additional experiment we mask the regions at the boundaries by setting the loss within the noise windows to zero. This leaves the networks to their own interpretation of the data.

w_{noise}	Bi-LSTM	ED-TCN	DilatedTCN	Ours ($w = 3$)
11 frames	81.02%	88.25%	89.04%	**92.42%**
21 frames	80.05%	85.88%	86.80%	**91.93%**
31 frames	76.71%	84.88%	86.01%	**90.48%**

TABLE 7.5: Per-frame test accuracies for our model ($w = 3$) and the benchmark models with the masked the boundary regions.

Tab. 7.5 shows that the accuracy increases for all convolution-based models but decreases for the RNN-based model, compared to the previous experiment with noise in the boundary regions during training time. Interestingly, some of the per-frame accuracy results are better than the results on the original vanilla labels from Tab. 7.2. A comparison is shown in Tab. 7.6. This might be due to inter- and intra-annotator disagreements [11] as described earlier. When there is no information given in these regions, the networks learn some representation from the geometric information only. This might be on average more accurate on unseen examples than learning from "human generated" annotations, which tend to be different from annotator to annotator (or even from using the same annotator) in boundary regions. Fig. 7.22 compares the results from Fig. 7.21 to predictions where the boundary regions are masked out for our model. The third row in each segmentation block corresponds to the latter prediction method. While masking these regions still leaves us with wrong classifications in some cases, the overall accuracy improves.

	Bi-LSTM	ED-TCN	DilatedTCN	Ours ($w = 3$)
Vanilla	**81.95%**	**88.69%**	88.47%	91.65%
11 frames	81.02%	88.25%	**89.04%**	**92.42%**
21 frames	80.05%	85.88%	86.80%	91.93%
31 frames	76.71%	84.88%	86.01%	90.48%

TABLE 7.6: Per-frame test accuracies for our model ($w = 3$) and the benchmark models with the masked the boundary regions, compared with the results of the *Vanilla Labels Experiment*.

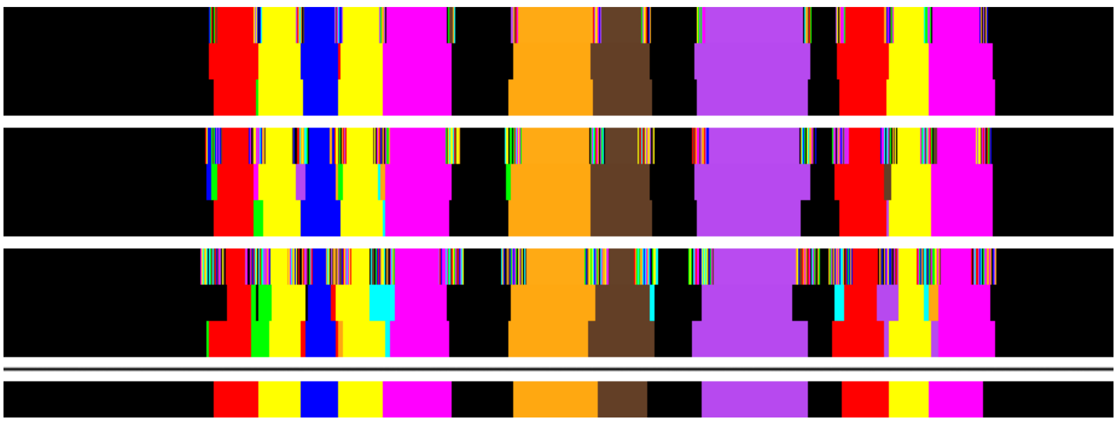


FIGURE 7.22: Predicted segmentations by our model ($w = 3$). Blocks from top to bottom: $w_{noise} = 11$ frames, $w_{noise} = 21$ frames, $w_{noise} = 31$ frames, original sequence segmentation. Each block consists of the (noisy) training label (top part in block), the corresponding predicted segmentation without masking out the noise (middle part in block) and with masking it out (bottom part of block).

To see what the network is seeing when no information on the boundary regions is given, we plot the class activation maps (CAMs) [139] (Fig. 7.23, 7.24 and 7.25) for our network. As expected the network trained on the noisy boundary regions has more uncertainty across the different class labels. The segment boundaries in some of the shown heatmaps (e.g. for the *none* class) have transitions which are not as distinct as the ones without noise or even without any information given at all. Interestingly, the transitions are much distincter when there is no information given in the boundary regions during training time than when the true class labels are given. This is true across all w_{noise} sizes, as well as, almost all CAMs. Again, this might be because the annotations given by humans tend to be more ambiguous in nature due to the inter- and intra-annotator differences. When no information at the boundaries is given, the network is able to detect boundaries using distinctive features in the motion images which are less ambiguous across the data set than the variations obtained by human annotators.

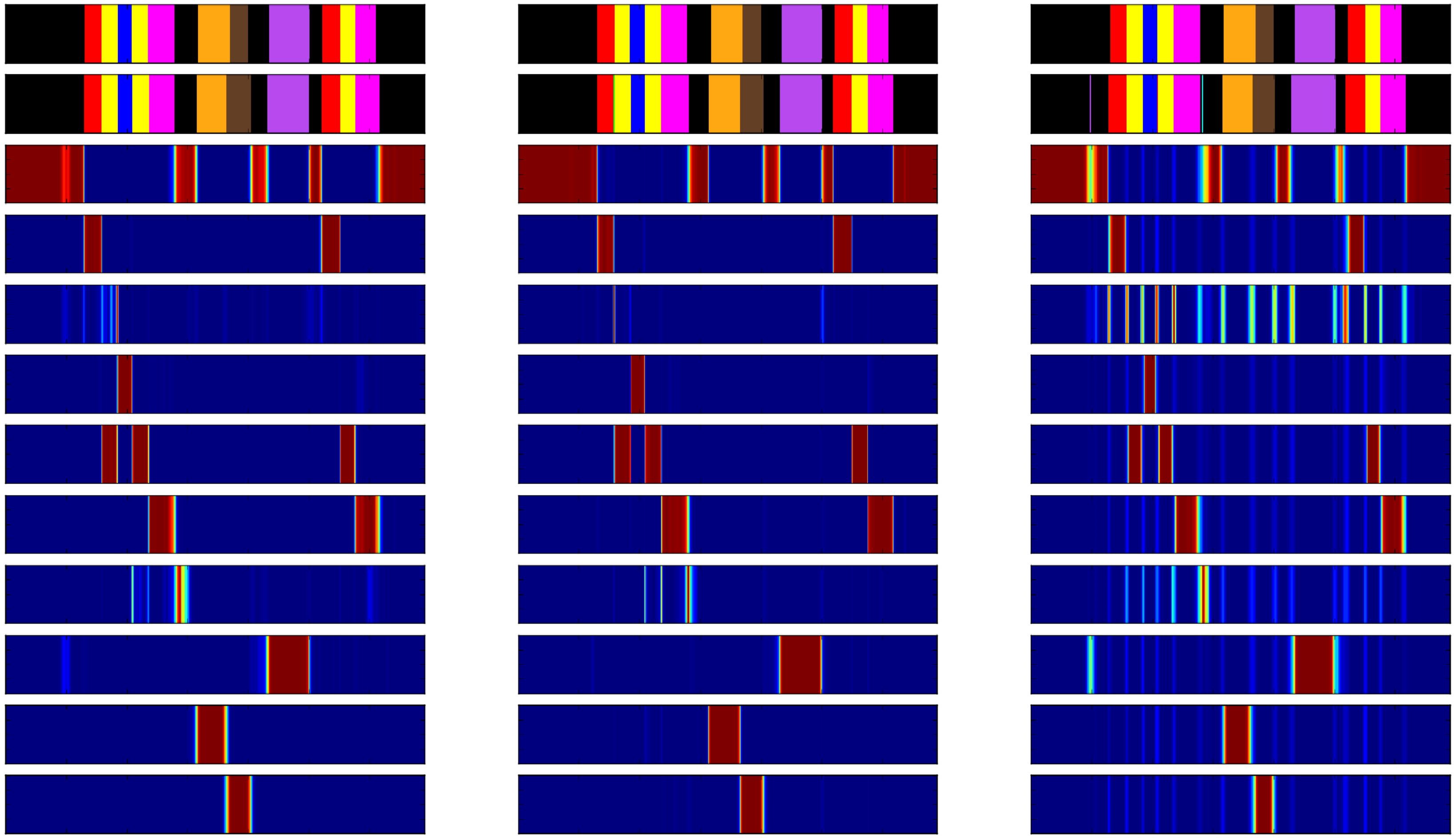


FIGURE 7.23: Class Activation Maps (CAM) for a transition window size $w_{noise} = 11$ with an example training sample. From *left* to *right*: CAM without any noise in transition window for training labels. CAM with no information in transition window for training labels. CAM with noise in transition window for training labels. The first row in each column shows the true labels without any noise and the row below that the prediction done by the network when trained on the noisy example. *Top* to *bottom* activation map: *none* (⬛), *begin right step* (🟥), *begin left step* (🟩), *right step* (🟦), *left step* (🟨), *end right step* (🟪), *end left step* (🟦), *turn* (🟪), *reach* (🟧) and *retrieve* (🟫)

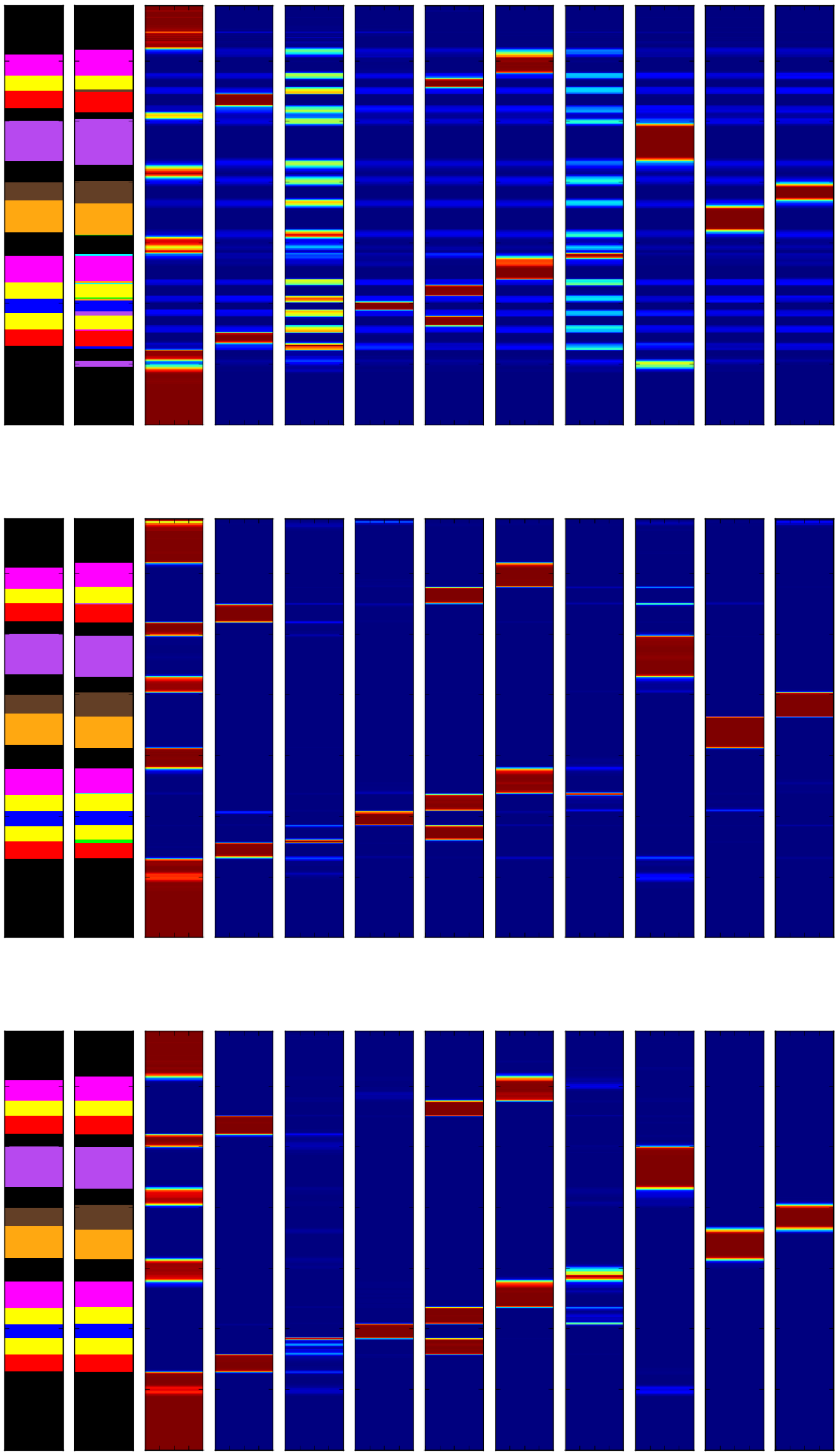


Figure 7.24: Class Activation Maps (CAM) for a transition window size $w_{noise} = 21$ with an example training sample. From *left* to *right*: CAM without any noise in transition window for training labels. CAM with no information in transition window for training labels. CAM with noise in transition window for training labels. The first row in each column shows the true labels without any noise and the row below that the prediction done by the network when trained on the noisy example. *Top* to *bottom* activation map: *none* (⬛), *begin right step* (🟥), *begin left step* (🟩), *right step* (🟦), *left step* (🟨), *end right step* (🟪), *end left step* (■), *turn* (🟪), *reach* (🟧) and *retrieve* (🟫)

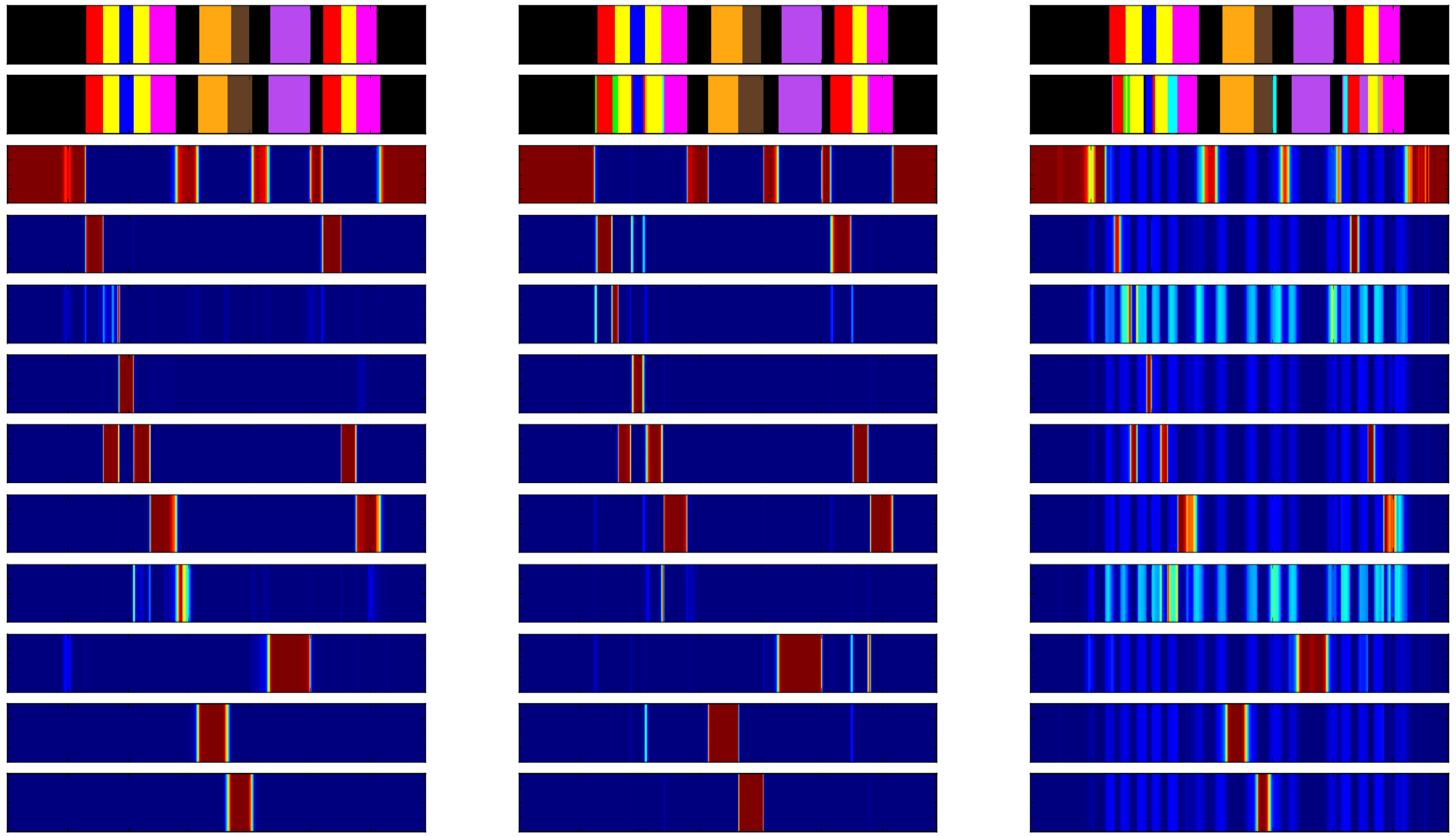


FIGURE 7.25: Class Activation Maps (CAM) for a transition window size $w_{noise} = 31$ with an example training sample. From *left* to *right*: CAM without any noise in transition window for training labels. CAM with no information in transition window for training labels. CAM with noise in transition window for training labels. The first row in each column shows the true labels without any noise and the row below that the prediction done by the network when trained on the noisy example. *Top* to *bottom* activation map: *none* (⬛), *begin right step* (🟥), *begin left step* (🟩), *right step* (🟦), *left step* (🟨), *end right step* (🟪), *end left step* (🟦), *turn* (🟪), *reach* (🟧) and *retrieve* (🟫)

Results on Generalization to *Walk 7-labels*

To test how well the models generalize to the walking sequences, we evaluate them on the *Walk 7-labels* set, which only contains walking sequences, while the models have only been trained on the *Mixed 10-labels* set.

Bi-LSTM	ED-TCN	DilatedTCN	Ours ($w = 3$)
87.62%	90.73%	**92.03%**	78.07%

TABLE 7.7: Per-frame accuracy for benchmark models and our best model ($w = 3$) on the walking dataset with *Walk 7-labels* granularity. The models were trained on the mixed dataset with *Mixed 10-labels* granularity.

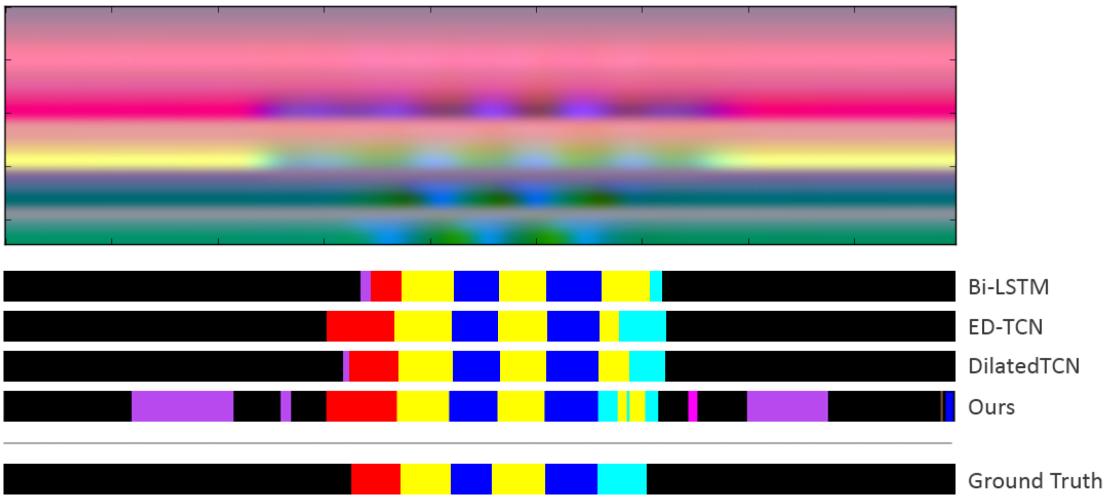


FIGURE 7.26: *Top:* Sample walking sequence. *Middle:* Test results on the *Walk 7-labels* from the benchmark and our models which were trained on the *Mixed 10-labels*. *Bottom:* Ground truth label.

The results in Tab. 7.8 show that our model performs worst on the dataset which only contains walking labels compared to the other model. Fig. 7.26 gives us further insight on what is happening. Many times the background class for standing is misclassified as *turn* in this dataset with our model. This might be because of the different scales of the input for our model and the benchmark models. As described in Section 7.5.4, the benchmark models operate on input scales between -1 and 1, while our model has been designed to directly operate on image data which typically has rounded values between 0 and 255.

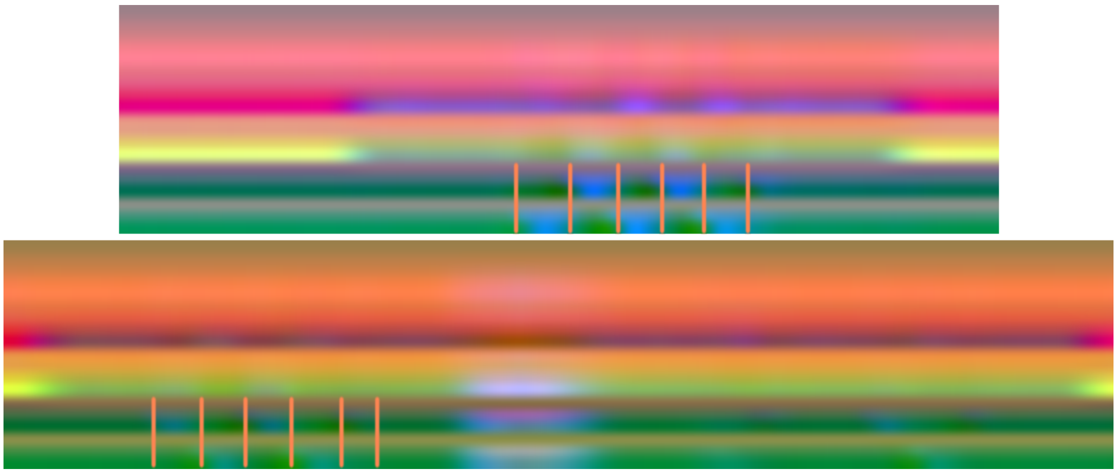


FIGURE 7.27: *Top:* Walking sequence. *Bottom:* Mixed sequence. The walking steps are much fainter in the mixed sequence due to interpolating between 0 and 255.

However, as we compare the motion images of a typical walking sequence and a sequence with mixed actions (see Fig. 7.27), we notice that the steps in the mixed sequence are much fainter compared to the walking sequence. This is because the maximum XYZ-value of that specific sequence is set to 255, which in the case of the mixed sequences is usually within in the picking or placing parts and the values for the other actions are further down and closer to each other. When just the walking action is given, the maximum lies somewhere in that action, so the other values are scaled according to that maximum. Hence, making the left and right steps more distinguishable. However, since our model has only been trained on the mixed dataset, where the values for standing and walking are closer to each other, the network might classify some of the standing frames in the walk-only sequence as turning, because the values for standing in the walk-only sequence might lie within range of the *turn* motion in the mixed dataset. To test this assumption, we scale the input walking sequence by multiplying it with $\frac{1}{1850}$, which gives us a more similar range to the walking parts in the mixed dataset. This number has been obtained through trial-and-error and could have been more precise by specifically calculating the range in which the walking parts lie within the mixed action sequence. However, already with this approximation we achieve an accuracy of 90.71% which is similar to the benchmark results:

Bi-LSTM	ED-TCN	DilatedTCN	Ours ($w = 3$, adjusted scale)
87.62%	90.73%	**92.03%**	90.71%

TABLE 7.8: Per-frame accuracy for benchmark models and our best model ($w = 3$) with adjusted scale on the walking dataset with *Walk 7-labels* granularity. The models were trained on the mixed dataset with *Mixed 10-labels* granularity.

Fig. 7.28 shows the result of the scaled input and the original input for our model.

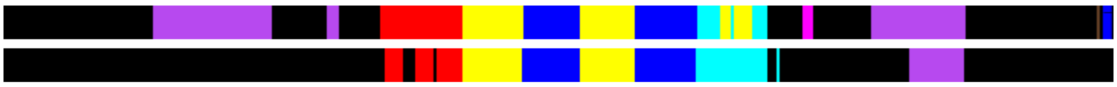


FIGURE 7.28: Sample test result on the *Walk 7-labels* from our model train on the *Mixed 10-labels*. *Top:* Without input scaling. *Bottom:* With input scaling.

Another solution to rescaling the input would be to compute a global maximum XYZ-value across all sequences and set that to 255 and rescale the other values according to that, instead of setting the maximum XYZ-value within one sequence to 255. To further improve the results, fine-tuning on the new data would also increase the models' accuracies.

7.6 Conclusion

In this chapter we have introduced various methods to increase a CNN's receptive field size. Furthermore, we have introduced a dilated temporal fully-convolutional network (DT-FCN) architecture for fine-grained semantic segmentation of motion data. We compare three different receptive field size (RFS) versions of our architecture and find that a receptive field of 243 frames, which defines a local structure in our sequences, performs best. This is in contrast to some claims [7, 129] that the RFS has to capture the whole sequence at once.

We analyze the robustness of our model against synthetic noise in the training labels. Our results show that even with 80% labeling noise, a performance of almost 90% accuracy in the test set is achieved for the random noise experiment. We compare our model against recent TCN-based baseline models for action segmentation, as well as a popular RNN model and outperform all of them in almost all experiments. This might be due to the combination of acausal and dilated convolutions in our models, coupled with the fast trainability of TCNs. Our model thus combines both advantages of the presented TCN benchmarks and is over a magnitude faster to train than the LSTM-based method. Compared to other dilated TCN architectures, our acausal convolutions with an increased dilation rate of $d = w^{l-1}$ instead of $d = 2^{l-1}$ [7, 72, 129, 135] account for an even larger receptive field size with fewer parameters without skipping inputs within a receptive field. Other advantages of our model compared to the benchmarks are that the output after each hidden layer has to be the same length as the input, which is unlike the encoder-decoder-based architecture of ED-TCN [72, 74]. In fact, the encoder-decoder architecture might even be a disadvantage when it comes to motion data: The encoder "compresses" its input into a lower-dimensional representation, possibly canceling important high-frequency features out, which might account for the lower accuracy compared to ours. Finally, RNN-based methods, such as the Bi-LSTM [41], are hard to parallelize due to their sequential nature, which gives them a major disadvantage compared to CNN-based methods. For a more realistic noise experiment, we add noise only in the boundary regions and compare the results to when no information is given in these regions. All models perform better when no information is given. Some of the models, including our's, perform even better when no information is given than when the clean ground truth information is given. This might be due to the inter- and intra-annotator disagreements in the ground truth data, which has been labeled by human "experts".

When it comes to generalizing to another dataset the causal DilatedTCN performs best, however; while our's performs worst. This experiment also shows the flaws of such an image representation. We counter this problem by scaling the input values to obtain a better segmentation output. Nevertheless, we believe that our method provides a fruitful tool for motion capture segmentation and has potential for further improvements.

Chapter 8

Summary and Outlook

8.1 Contribution

The thesis has presented the first approach for fine-grained semantic segmentation of motion data based on Convolutional Neural Networks. Compared to commonly used unsupervised methods [70, 140, 141], the presented (semi-)supervised methods are able to learn complex labels such as *begin left step* or *end right step*, while robustly handling labeling errors. Hence, we believe such (semi-)supervised deep learning methods are a fruitful direction to explore motion segmentation methods.

Our presented dilated temporal fully-convolutional neural network exceeds popular networks for sequence modeling, as well as, state-of-the-art networks for action segmentation on the presented dataset. While the key-ingredients for the success were already present in prior work [7, 71, 86], we believe it is the combination of these methods which accounts for the improved accuracy compared to other TCN-based methods. The combination of a VGG/FCN32-inspired [86, 118] model with acausal dilated convolutions and an increased dilation rate compared to other methods [72, 129] enables a large receptive field with even fewer parameters. Thus training time is reduced while the performance of the model is above or comparable to state-of-the-art competitors. Most of all, the model is very robust under label noise. Hence, even cheaply labeled data can be used for training. Even more, we found that a semi-supervised approach by removal of the boundary labels between classes can improve the performance further.

8.2 Limitations

The presented approach achieves a robust and accurate motion segmentation, although there are some limitations:

- **Truly unseen motions:** On one hand, such (semi-)supervised methods can output complex semantic labels and enhance controllability of the segmentation. On the other hand, this still requires these methods to require labeled motions. Hence, truly unseen motions can therefore not be handled.

- **Motion Image representation:** Processing the motion data to an image can have some disadvantages, as shown in the previous chapter. Due to the scaling between 0 and 255, nuances of different motions may be lost, and generalizing to various motions and datasets becomes difficult.
- **Testing on other skeletons:** As the networks are trained on a dataset with a specific skeleton and order of joints, it is hard to generalize a trained model on another testing set with a different configuration. To counter this one would need to include all possible combinations and orders of joints in the training set or extract the features of a trained model and re-order in accordance to the new dataset. However, this was not the focus of our thesis.

8.3 Future Work

There are multiple interesting directions for further research. Additional experiments to model inter- and intra-annotator disagreements better should be conducted to better prove the robustness against human error as well as the recording quality of the motion capturing solution. Furthermore, we found that per-frame accuracy does not necessarily capture time-continuities well, we therefore want to explore action detection-based metrics, such as mean Average Precision with midpoint hit criterion (mAP@mid) [108, 119] or mAP with a intersection over union (IoU) overlap criterion (mAP@k) [107]. Datasets with more complex actions should also be explored to further consolidate our assumptions, as well as improving our method to handle various skeletons on a trained model and thus a better data representation. Finally, although this approach does require considerably less human effort than manual segmentation, further research in semi- and un-supervised has to be conducted to make the model more automatic. In this direction domain transfer [33] and few-shot learning [26] can provide even less human effort, as well as self-supervised learning [4, 136, 137].

List of Figures

List of Tables

Bibliography

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: a system for large-scale machine learning. In *OSDI* (2016), vol. 16, pp. 265–283.

[2] Altman, N. S. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician 46*, 3 (1992), 175–185.

[3] Arikan, O., Forsyth, D. A., and O'Brien, J. F. Motion synthesis from annotations. In *ACM Transactions on Graphics (TOG)* (2003), vol. 22, ACM, pp. 402–408.

[4] Aristidou, A., Cohen-Or, D., Hodgins, J. K., Chrysanthou, Y., and Shamir, A. Deep motifs and motion signatures. In *SIGGRAPH Asia 2018 Technical Papers* (2018), ACM, p. 187.

[5] Aristidou, A., Cohen-Or, D., Hodgins, J. K., and Shamir, A. Self-similarity analysis for motion capture cleaning. In *Computer Graphics Forum* (2018), vol. 37, Wiley Online Library, pp. 297–309.

[6] Arjovsky, M., Chintala, S., and Bottou, L. Wasserstein gan. *arXiv preprint arXiv:1701.07875* (2017).

[7] Bai, S., Kolter, J. Z., and Koltun, V. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271* (2018).

[8] Barbič, J., Safonova, A., Pan, J.-Y., Faloutsos, C., Hodgins, J. K., and Pollard, N. S. Segmenting motion capture data into distinct behaviors. In *Proceedings of Graphics Interface 2004* (2004), Canadian Human-Computer Communications Society, pp. 185–194.

[9] Baxter, D. A., and Byrne, J. H. Simulator for neural networks and action potentials. In *Neuroinformatics*. Springer, 2007, pp. 127–154.

[10] Bottou, L. Online learning and stochastic approximations. *On-line learning in neural networks 17*, 9 (1998), 142.

[11] Bouchard, D. *Automated motion capture segmentation using Laban Movement Analysis*. PhD thesis, University of Pennsylvania, 2008.

[12] Bouchard, D., and Badler, N. Semantic segmentation of motion capture using laban movement analysis. In *International Workshop on Intelligent Virtual Agents* (2007), Springer, pp. 37–44.

[13] Bouchard, D., and Badler, N. I. Segmenting motion capture data using a qualitative analysis. In *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games* (2015), ACM, pp. 23–30.

[14] Castellano, G., Villalba, S. D., and Camurri, A. Recognising human emotions from body movement and gesture dynamics. In *International Conference on Affective Computing and Intelligent Interaction* (2007), Springer, pp. 71–82.

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

[16] CMU. Carnegie mellon university graphics lab: Motion capture database. `http://mocap.cs.cmu.edu/`, 2013. Carnegie Mellon University.

[17] Collobert, R., Kavukcuoglu, K., Farabet, C., et al. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS workshop* (2011), vol. 5, Granada, p. 10.

[18] Cortes, C., and Vapnik, V. Support-vector networks. *Machine learning 20*, 3 (1995), 273–297.

[19] Dauphin, Y. N., Fan, A., Auli, M., and Grangier, D. Language modeling with gated convolutional networks. *arXiv preprint arXiv:1612.08083* (2016).

[20] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database.

[21] Du, H., Manns, M., Herrmann, E., and Fischer, K. Joint angle data representation for data driven human motion synthesis. *Procedia CIRP 41* (2016), 746–751.

[22] Du, Y., Wang, W., and Wang, L. Hierarchical recurrent neural network for skeleton based action recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), pp. 1110–1118.

[23] Duda, R. O., Hart, P. E., and Stork, D. G. *Pattern classification.* John Wiley & Sons, 2012.

[24] Everingham, M., Van Gool, L., Williams, C. K. I., Winn, J., and Zisserman, A. The pascal visual object classes challenge 2012 (voc2012) results. `http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html`.

[25] Fathi, A., Ren, X., and Rehg, J. M. Learning to recognize objects in egocentric activities. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference On* (2011), IEEE, pp. 3281–3288.

[26] Fei-Fei, L., Fergus, R., and Perona, P. One-shot learning of object categories. *IEEE transactions on pattern analysis and machine intelligence 28*, 4 (2006), 594–611.

[27] Fod, A., Matarić, M. J., and Jenkins, O. C. Automated derivation of primitives for movement classification. *Autonomous robots 12*, 1 (2002), 39–54.

[28] for Education, G. F. M., and (BMBF), R. Hybri-it: Mensch-roboter kollaboration. `https://hybr-it-projekt.de/`, 2016. Grant Number: 01IS16026A.

[29] Fothergill, S., Mentis, H., Kohli, P., and Nowozin, S. Instructing people for training gestural interactive systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2012), ACM, pp. 1737–1746.

[30] (FP7), E. U. S. F. P. Interact. `http://www.interact-fp7.eu/`, 2013. Grant Number: 611007.

[31] Fragkiadaki, K., Levine, S., Felsen, P., and Malik, J. Recurrent network models for human dynamics. In *Proceedings of the IEEE International Conference on Computer Vision* (2015), pp. 4346–4354.

[32] Fukushima, K., and Miyake, S. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*. Springer, 1982, pp. 267–285.

[33] Ganin, Y., and Lempitsky, V. Unsupervised domain adaptation by backpropagation. *arXiv preprint arXiv:1409.7495* (2014).

[34] Gatys, L., Ecker, A. S., and Bethge, M. Texture synthesis using convolutional neural networks. In *Advances in Neural Information Processing Systems* (2015), pp. 262–270.

[35] Gatys, L. A., Ecker, A. S., and Bethge, M. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 2414–2423.

[36] Gehring, J., Auli, M., Grangier, D., Yarats, D., and Dauphin, Y. N. Convolutional sequence to sequence learning. *arXiv preprint arXiv:1705.03122* (2017).

[37] Gers, F. A., Schmidhuber, J., and Cummins, F. Learning to forget: Continual prediction with lstm.

[38] Gleicher, M. Cs838 - topics in computer animation. `https://research.cs.wisc.edu/graphics/Courses/cs-838-1999/Jeff/Example1.bvh`, 1999. University of WisconsinMadison.

[39] Glorot, X., Bordes, A., and Bengio, Y. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (2011), pp. 315–323.

[40] Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. *Deep learning*, vol. 1. 2016. MIT Press Cambridge.

[41] Graves, A., and Schmidhuber, J. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks 18*, 5-6 (2005), 602–610.

[42] Guo, S., Southern, R., Chang, J., Greer, D., and Zhang, J. J. Adaptive motion synthesis for virtual characters: a survey. *The Visual Computer 31*, 5 (2015), 497–512.

[43] Hahnloser, R. H., Sarpeshkar, R., Mahowald, M. A., Douglas, R. J., and Seung, H. S. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature 405*, 6789 (2000), 947.

[44] Hamilton, J. D. *Time series analysis*, vol. 2. Princeton university press Princeton, NJ, 1994.

[45] Hartline, H. K. The response of single optic nerve fibers of the vertebrate eye to illumination of the retina. *American Journal of Physiology-Legacy Content 121*, 2 (1938), 400–415.

[46] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[47] Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001. A field guide to dynamical recurrent neural networks. IEEE Press.

[48] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[49] Holden, D., Komura, T., and Saito, J. Phase-functioned neural networks for character control. *ACM Transactions on Graphics (TOG) 36*, 4 (2017), 42.

[50] Holden, D., Saito, J., and Komura, T. A deep learning framework for character motion synthesis and editing. *ACM Transactions on Graphics (TOG) 35*, 4 (2016), 138.

[51] Inc., N. Optitrack motion capture system. `https://optitrack.com/`, 2016.

[52] Inoue, M., Ogihara, M., Hanada, R., and Furuyama, N. Gestural cue analysis in automated semantic miscommunication annotation. *Multimedia tools and applications 61*, 1 (2012), 7–20.

[53] Isola, P., Zhu, J.-Y., Zhou, T., and Efros, A. A. Image-to-image translation with conditional adversarial networks. *arXiv preprint* (2017).

[54] Jain, A. K., Murty, M. N., and Flynn, P. J. Data clustering: a review. *ACM computing surveys (CSUR) 31*, 3 (1999), 264–323.

[55] Jenkins, O. C., and Mataric, M. J. Automated derivation of behavior vocabularies for autonomous humanoid motion. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems* (2003), ACM, pp. 225–232.

[56] Jenkins, O. C., and Matarić, M. J. A spatio-temporal extension to isomap nonlinear dimension reduction. In *Proceedings of the twenty-first international conference on Machine learning* (2004), ACM, p. 56.

[57] Ji, S., Xu, W., Yang, M., and Yu, K. 3d convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence 35*, 1 (2013), 221–231.

[58] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia* (2014), ACM, pp. 675–678.

[59] Jordan, J. Setting the learning rate of your neural network. `https://www.jeremyjordan.me/nn-learning-rate/`, 2018. JeremyJordan.me.

[60] Kahol, K., Tripathi, P., and Panchanathan, S. Automated gesture segmentation from dance sequences. In *Automatic Face and Gesture Recognition, 2004. Proceedings. Sixth IEEE International Conference on* (2004), IEEE, pp. 883–888.

[61] Kalchbrenner, N., Espeholt, L., Simonyan, K., Oord, A. v. d., Graves, A., and Kavukcuoglu, K. Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099* (2016).

[62] Karpathy, A. Cs231n convolutional neural networks for vision recognition, 2018. Stanford University.

[63] Karpathy, A., and Fei-Fei, L. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), pp. 3128–3137.

[64] Ke, Q., Bennamoun, M., An, S., Sohel, F., and Boussaid, F. A new representation of skeleton sequences for 3d action recognition. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on* (2017), IEEE, pp. 4570–4579.

[65] Kessy, A., Lewin, A., and Strimmer, K. Optimal whitening and decorrelation. *The American Statistician 72*, 4 (2018), 309–314.

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

[67] Kovar, L., and Gleicher, M. Flexible automatic motion blending with registration curves. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2003), Eurographics Association, pp. 214–224.

[68] Kovar, L., Gleicher, M., and Pighin, F. Motion graphs. In *ACM SIGGRAPH 2008 classes* (2008), ACM, p. 51.

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

[70] Krüger, B., Vögele, A., Willig, T., Yao, A., Klein, R., and Weber, A. Efficient unsupervised temporal segmentation of motion data. *IEEE Transactions on Multimedia 19*, 4 (2017), 797–812.

[71] Laraba, S., Brahimi, M., Tilmanne, J., and Dutoit, T. 3d skeleton-based action recognition by representing motion capture sequences as 2d-rgb images. *Computer Animation and Virtual Worlds 28*, 3-4 (2017), e1782.

[72] Lea, C., Flynn, M. D., Vidal, R., Reiter, A., and Hager, G. D. Temporal convolutional networks for action segmentation and detection. In *proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2017), pp. 156–165.

[73] Lea, C., Reiter, A., Vidal, R., and Hager, G. D. Segmental spatiotemporal cnns for fine-grained action segmentation. In *European Conference on Computer Vision* (2016), Springer, pp. 36–52.

[74] Lea, C., Vidal, R., Reiter, A., and Hager, G. D. Temporal convolutional networks: A unified approach to action segmentation. In *European Conference on Computer Vision* (2016), Springer, pp. 47–54.

[75] Lea, C. S., et al. *Multi-Modal Models for Fine-grained Action Segmentation in Situated Environments.* PhD thesis, Johns Hopkins University, 2017.

[76] LeCun, Y. In convolutional nets, there is no such thing as "fully-connected layers". `https://www.facebook.com/yann.lecun/posts/10152820758292143`, 2015. Facebook.

[77] LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *nature 521*, 7553 (2015), 436.

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

[79] Lee, C.-S., and Elgammal, A. Human motion synthesis by motion manifold learning and motion primitive segmentation. In *International Conference on Articulated Motion and Deformable Objects* (2006), Springer, pp. 464–473.

[80] Lee, J., Chai, J., Reitsma, P. S., Hodgins, J. K., and Pollard, N. S. Interactive control of avatars animated with human motion data. In *ACM Transactions on Graphics (ToG)* (2002), vol. 21, ACM, pp. 491–500.

[81] Lewis, D. D. Naive (bayes) at forty: The independence assumption in information retrieval. In *European conference on machine learning* (1998), Springer, pp. 4–15.

[82] Li, W., Zhang, Z., and Liu, Z. Action recognition based on a bag of 3d points. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on* (2010), IEEE, pp. 9–14.

[83] Lin, M., Chen, Q., and Yan, S. Network in network. *arXiv preprint arXiv:1312.4400* (2013).

[84] Lindsay, B. G. Mixture models: theory, geometry and applications. In *NSF-CBMS regional conference series in probability and statistics* (1995), JSTOR, pp. i–163.

[85] Liu, D. C., and Nocedal, J. On the limited memory bfgs method for large scale optimization. *Mathematical programming 45*, 1-3 (1989), 503–528.

[86] Long, J., Shelhamer, E., and Darrell, T. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), pp. 3431–3440.

[87] Lv, F., and Nevatia, R. Recognition and segmentation of 3-d human action using hmm and multi-class adaboost. In *European conference on computer vision* (2006), Springer, pp. 359–372.

[88] MacQueen, J., et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability* (1967), vol. 1, Oakland, CA, USA, pp. 281–297.

[89] Maron, M. E., and Kuhns, J. L. On relevance, probabilistic indexing and information retrieval. *Journal of the ACM (JACM) 7*, 3 (1960), 216–244.

[90] MathWorks®. Convolutional neural networks. `https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html`, 2017. MathWorks®.

[91] Meredith, M., Maddock, S., et al. Motion capture file formats explained. *Department of Computer Science, University of Sheffield 211* (2001), 241–244.

[92] Min, J., and Chai, J. Motion graphs++: a compact generative model for semantic motion analysis and synthesis. *ACM Transactions on Graphics (TOG) 31*, 6 (2012), 153.

[93] Min, J., Chen, Y.-L., and Chai, J. Interactive generation of human animation with deformable motion models. *ACM Transactions on Graphics (TOG) 29*, 1 (2009), 9.

[94] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (2016), pp. 1928–1937.

[95] Müller, M., and Röder, T. Motion templates for automatic classification and retrieval of motion capture data. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2006), Eurographics Association, pp. 137–146.

[96] Müller, M., Röder, T., and Clausen, M. Efficient content-based retrieval of motion capture data. In *ACM Transactions on Graphics (ToG)* (2005), vol. 24, ACM, pp. 677–685.

[97] Müller, M., Röder, T., Clausen, M., Eberhardt, B., Krüger, B., and Weber, A. Documentation mocap database hdm05. Tech. Rep. CG-2007-2, Universität Bonn, June 2007.

[98] Newlove, J. *Laban for actors and dancers: putting labanas movement theory into practice: a step-by-step guide.* 1993.

[99] Ng, A. Sequence models (course 5 of deep learning specialization), 2018. Coursera.

[100] Ofli, F., Chaudhry, R., Kurillo, G., Vidal, R., and Bajcsy, R. Berkeley mhad: A comprehensive multimodal human action database. In *Applications of Computer Vision (WACV), 2013 IEEE Workshop on* (2013), IEEE, pp. 53–60.

[101] Olah, C. Understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015. colah's blog.

[102] Patsadu, O., Nukoolkit, C., and Watanapa, B. Human gesture recognition using kinect camera. In *Computer Science and Software Engineering (JCSSE), 2012 International Joint Conference on* (2012), IEEE, pp. 28–32.

[103] Pearson, K. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 2*, 11 (1901), 559–572.

[104] Pullen, K., and Bregler, C. Motion capture assisted animation: Texturing and synthesis. In *ACM Transactions on Graphics (TOG)* (2002), vol. 21, ACM, pp. 501–508.

[105] Radford, A., Metz, L., and Chintala, S. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[106] René, C., and Hager, V. Temporal convolutional networks for action segmentation and detection. In *IEEE International Conference on Computer Vision (ICCV)* (2017), vol. 1, p. 3.

[107] Richard, A., and Gall, J. Temporal action detection using a statistical language model. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 3131–3140.

[108] Rohrbach, M., Rohrbach, A., Regneri, M., Amin, S., Andriluka, M., Pinkal, M., and Schiele, B. Recognizing fine-grained and composite activities using hand-centric features and script data. *International Journal of Computer Vision 119*, 3 (2016), 346–373.

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

[110] Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review 65*, 6 (1958), 386.

[111] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *nature 323*, 6088 (1986), 533.

[112] Russell, S. J., and Norvig, P. *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited,, 2016.

[113] Safonova, A., Hodgins, J. K., and Pollard, N. S. Synthesizing physically realistic human motion in low-dimensional, behavior-specific spaces. In *ACM Transactions on Graphics (ToG)* (2004), vol. 23, ACM, pp. 514–521.

[114] Schölkopf, B., Smola, A., and Müller, K.-R. Nonlinear component analysis as a kernel eigenvalue problem. *Neural computation 10*, 5 (1998), 1299–1319.

[115] Schroff, F., Kalenichenko, D., and Philbin, J. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), pp. 815–823.

[116] Schumacher, J., Sakič, D., Grumpe, A., Fink, G. A., and Wöhler, C. Active learning of ensemble classifiers for gesture recognition. In *Joint DAGM (German Association for Pattern Recognition) and OAGM Symposium* (2012), Springer, pp. 498–507.

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

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

[119] Singh, B., Marks, T. K., Jones, M., Tuzel, O., and Shao, M. A multi-stream bi-directional recurrent neural network for fine-grained action detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 1961–1970.

[120] Sivaramakrishnan, R., Antani, S., Xue, Z., Candemir, S., Jaeger, S., and Thoma, G. R. Visualizing abnormalities in chest radiographs through salient network activations in deep learning. In *2017 IEEE Life Sciences Conference (LSC)* (Dec 2017), pp. 71–74.

[121] Sobel, I., and Feldman, G. A 3x3 isotropic gradient operator for image processing. *a talk at the Stanford Artificial Project in* (1968), 271–272.

[122] Sola, J., and Sevilla, J. Importance of input data normalization for the application of neural networks to complex industrial problems. *IEEE Transactions on nuclear science 44*, 3 (1997), 1464–1468.

[123] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research 15*, 1 (2014), 1929–1958.

[124] Stein, S., and McKenna, S. J. Combining embedded accelerometers with computer vision for recognizing food preparation activities. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing* (2013), ACM, pp. 729–738.

[125] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), pp. 1–9.

[126] Taylor, G. W., Fergus, R., LeCun, Y., and Bregler, C. Convolutional learning of spatio-temporal features. In *European conference on computer vision* (2010), Springer, pp. 140–153.

[127] Tipping, M. E., and Bishop, C. M. Probabilistic principal component analysis. *Journal of the Royal Statistical Society: Series B (Statistical Methodology) 61*, 3 (1999), 611–622.

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

[129] Van Den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. Wavenet: A generative model for raw audio. *CoRR abs/1609.03499* (2016).

[130] Vemulapalli, R., Arrate, F., and Chellappa, R. Human action recognition by representing 3d skeletons as points in a lie group. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2014), pp. 588–595.

[131] Vögele, A., Krüger, B., and Klein, R. Efficient unsupervised temporal segmentation of human motion. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2014), Eurographics Association, pp. 167–176.

[132] Wu, D., and Shao, L. Leveraging hierarchical parametric networks for skeletal joints based action segmentation and recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2014), pp. 724–731.

[133] Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhudinov, R., Zemel, R., and Bengio, Y. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning* (2015), pp. 2048–2057.

[134] Yildirim, Ö. A novel wavelet sequence based on deep bidirectional lstm network model for ecg signal classification. *Computers in biology and medicine 96* (2018), 189–202.

[135] Yu, F., and Koltun, V. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122* (2015).

[136] Zhang, R., Isola, P., and Efros, A. A. Colorful image colorization. In *European conference on computer vision* (2016), Springer, pp. 649–666.

[137] Zhang, R., Isola, P., Efros, A. A., Shechtman, E., and Wang, O. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018), pp. 586–595.

[138] Zhao, L., and Badler, N. I. Acquiring and validating motion qualities from live limb gestures. *Graphical Models 67*, 1 (2005), 1–16.

[139] Zhou, B., Khosla, A., Lapedriza, A., Oliva, A., and Torralba, A. Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 2921–2929.

[140] Zhou, F., De la Torre, F., and Hodgins, J. K. Aligned cluster analysis for temporal segmentation of human motion. In *Automatic Face & Gesture Recognition, 2008. FG'08. 8th IEEE International Conference on* (2008), IEEE, pp. 1–7.

[141] Zhou, F., De la Torre, F., and Hodgins, J. K. Hierarchical aligned cluster analysis for temporal clustering of human motion. *IEEE Transactions on Pattern Analysis and Machine Intelligence 35*, 3 (2013), 582–596.

[142] Zhu, Y., and Gortler, S. J. 3d deformation using moving least squares.