A Transformer for Multi-Camera Multi-Object Tracking

Master's Thesis

presented by
Tobias Stenzel
Matriculation Number 1374389

submitted to the
Data and Web Science Group
Prof. Dr. Paul Swoboda
University of Mannheim

August 2023

Contents

List of Algorithms

List of Figures

List of Tables

Abstract

This thesis investigates Multiple Camera Multiple Object Tracking. While extensive research has been dedicated to single-camera Multiple Object Tracking (MOT), complexities in tracking densely populated scenes persist due to factors like occlusion and object indistinctness. To address these challenges, the potential of multi-camera tracking is explored, leveraging diverse viewpoints to enhance tracking accuracy.

The study introduces a comprehensive model that harnesses both temporal and spatial interactions among objects detected by multiple cameras. The model's core is a streamlined transformer-based framework. Its architecture encompasses distinct phases, including the extraction of image tokens from various cameras, object detection and tracking queries employing unique decoder input embeddings, and the representation of unified 3D detections using cylinders. Notably, this approach deviates from direct 2D bounding box predictions, utilizing camera calibration parameters for dynamic scene representation. The thesis highlights key components implemented within the model, such as the multi-camera tracking system, the application of a suitable object detector with tailored transformer dimensions, and 2D-3D transformations facilitating dataset creation and training evaluation.

Acknowledgments

I would like to express my heartfelt gratitude to Professor Paul Swoboda for his invaluable guidance, unwavering support, and insightful feedback throughout the journey of completing my Master's thesis. His expertise, encouragement, and mentorship have been instrumental in shaping my research and academic growth.

I am also deeply thankful to my parents, whose unwavering love, encouragement, and belief in my abilities have been my constant source of strength. Their endless support and sacrifices have played a pivotal role in enabling me to pursue my academic aspirations and reach this significant milestone.

Chapter 1

Introduction

1.1 Motivation and Problem

The extraction of object trajectories, a key element in comprehending high-level information within videos, forms the foundational building block of multiple object tracking (MOT). This process follows a tracking-by-assignment paradigm, involving the initial computation of detection boxes for objects of interest within each time frame, followed by data association linking detections with the same unique ID across time. A popular problem is single-camera MOT, where a solitary camera captures a scene, and the data association mechanism connects detections from the previous frame to the next frame. Despite the extensive research dedicated to single-camera MOT, the tracking of intricate and densely populated scenes remains a challenge, primarily attributed to errors that arise during data association. These errors are frequently instigated by conditions such as partial visibility or complete occlusion, along with the inherent indistinguishability of objects.

A less explored approach for enhancing tracking performance entails the utilization of multiple cameras, each positioned at different angles to observe the same scene (as illustrated in Figure 1.1). In this configuration, the issues of partial visibility and object indistinguishability are mitigated. Even if an object becomes obscured in one camera's view, another camera might retain a clear perspective. Similarly, objects that are nearly identical may be distinguished through the triangulation of their 3D positions across multiple cameras, adding a distinctive spatial dimension. However, this additional dimension makes the data association more complex because beyond the linkage of detections across temporal frames (temporal association), there arises the need to connect detections simultaneously observed by distinct cameras (spatial association). Furthermore, the resultant temporal and spatial associations must seamlessly yield coherent object tracks.

Figure 1.1: Occlusion in multi-camera tracking datasets. The green bounding box shows a pedestrian who is occluded in Camera 7 but not in Camera 1 in the same time period. The main purpose of multi-camera compared to single-camera methods and data is to improve object tracking through occlusions using multiple views of the same scene. Images taken from WILDTRACK dataset (Chavdarova et al. [14]).

This pursuit of robust multi-camera tracking is the central motivation behind our study, as we aim to address the intricate challenges presented by dynamic scenes and exploit the advantages offered by multiple camera perspectives.

1.2 Approach and Contribution

In this contribution, we present a model that considers both the temporal interactions of objects detected by a group of cameras and the spatial interactions between these cameras. Most importantly, it offers an end-to-end solution. Typically, tasks like detection, tracking, and particularly multi-camera tracking encompass numerous distinct model components. In contrast, our approach is streamlined and centered around a core transformer module.

Building on the ideas of the single-camera model Trackformer (Meinhardt et al. [64]) and the transformer-based object detector DETR (Carion et al. [12]), our model has three main parts:

1. **Extracting image tokens from multiple cameras.** For every camera, we input the image into a CNN backbone and include camera-specific learned embeddings. Subsequently, we pass the sequence of image tokens from all cameras through a transformer encoder.

2. **Object detection and tracking queries.** We train unique decoder input embeddings known as object queries. Post-decoding, each object query has the

potential to initiate a new object detection. The output embeddings, previously matched to ground truth detections serve as track queries to the decoder, with the goal of identifying the same object in the next set of images.

3. **Cylinders as unified 3D detections.** Instead of directly predicting 2D bounding boxes on the image plane for each camera, we model cylinders in the shared 3D world, utilizing camera calibration parameters. This approach allows us to create a dynamically consistent scene representation that can be projected onto different image planes for evaluation purposes.

A compilation of the most crucial components that we have implemented includes:

- The multi-camera tracking model.
- DETR as object detector with adjusted transformer dimensions to accommodate pre-trained detection embeddings.
- 2D-3D transformations that facilitate the creation of our 3D dataset, as well as the evaluation and monitoring of training using 2D image planes.
- Control over learning rates for various transformer layers to improve the learning process.
- The 3D multi-camera dataset and the associated dataloader.

We analyze the model from a conceptual perspective because the implementation does not contain the tracking evaluation part that is necessary for a comprehensive empirical analysis based on experiments.

1.3 Thesis Outline

We structure the thesis as follows:

- Chapter 2 introduces the MOT task, including metrics for model evaluation, and provides summaries of related literature and datasets for both single-camera and multi-camera MOT.
- Chapter 3 outlines the conceptual prerequisites and notation of deep neural networks with a focus on transformers. Furthermore, Chapter 4 offers a concise introduction to the conversion between 2D camera coordinates and 3D world coordinates.

- Chapter 5 presents our comprehensive multi-camera multi-object tracking model, building upon the foundations of Trackformer (Meinhardt et al. [64]) and DETR (Carion et al. [12]).
- Chapter 6 describes the data and briefly outlines the evaluation of Trackformer without multi-camera expansion. We then present the current status of our implementation and discuss conceptual advantages and limitations of our approach.
- Chapter 7 provides a summary of our contributions, followed by an exploration of areas that we believe merit further research.

Chapter 2

Related Literature

This chapter summarizes the related literature. The first section introduces the multi-object tracking task including the metrics for model evaluation. The second section summarizes the related literature and the datasets for single-camera and the third section for multiple camera multiple object tracking.

2.1 Introduction to Multiple Object Tracking

2.1.1 Overview

Multiple object tracking (MOT) models take video data and identify image parts that depict the same object over multiple frames. Furthermore, they classify the respective objects. MOT models are related to object detection models, which only locate and classify objects on a single frame. To be precise, the MOT objective is detecting objects on every single frame, i.e determining the bounding box coordinates and the class of every object, and to equip each detection with a unique ID that does not change between frames if the detection refers to the same object. Our references are the lecture CV3DST by Leal-Taixé [52] and the survey by Luo et al. [63] for more extensive introductions.

On a technical level, tracking is important for two reasons. First, a tracking model without the data association can better detect objects on video data than a usual frame-level detector because it uses additional information from other frames of the video. This helps, for instance, with overcoming occlusions and background clutter. Second, normal trackers that associate object IDs across frames allow us to reason about motion, for example, by predicting trajectories [52, Ch. 4 p. 3].

Applications. There are many applications such as video surveillance [24], autonomous driving [16], medical imaging [78], robotics [1], sports analysis [82]

and augmented reality [71]. For instance, tracking the position and movement of other vehicles, pedestrians, and obstacles on the road helps self-driving cars make informed decisions about acceleration and direction.

Challenges. Three key challenges are occlusions, object similarity and highly complex scenes [52, Ch. 5 p. 22]. If objects are occluded by other objects, it is hard to detect the occluded objects and to keep the identities distinguished in the subsequent frames. Occlusions are especially frequent in videos that show interacting objects.

Similar objects are also challenging. They make tracking more difficult as distinction between objects can only be made based on location and predicted motion. Especially demanding data are videos with many objects per frame, cluttered backgrounds with many irrelevant objects that should not be tracked, and changes to the camera position or picture quality.

Online tracking. We distinguish between online and offline tracking [52, Ch. 5 p. 24]. Online tracking processes only two frames at a time. The advantage is that we can use online models for real-time applications such as autonomous driving. The disadvantage is that it is hard to recover from errors or occlusions. This problem is called drifting. Offline tracking, on the other hand, processes a batch of frames at once. We can use it for deeper analyses of stored videos. The thesis focuses on online tracking.

Online-tracking components. Figure 2.1 shows the three online tracking subtasks. The first subtask is track initialization. This is usually achieved with an object detection model. Second, a motion model predicts the position of the previously tracked objects in the next frame using the information from the previous frame. Third, an appearance model matches the predictions and detections on every frame except of the first image [52, Ch. 5 pp. 25-36]. As an example, consider the following rough sketch of an online-tracking model: First, the model represents each detection on each frame by a vector embedding that encodes its location and appearance. From this embedding, it predicts the respective bounding boxes and classes. Second, we write the matching problem as maximum-weight matching in bipartite graphs to identify objects from the previous frame on the following frame. The model matches the pairs of detections with the highest similarity score (or smallest distance) for two subsequent frames from their respective embeddings with the Hungarian matching algorithm [50]. The right panel in Figure 2.1 depicts the adjacency matrix for the matching problem. In contrast to the figure, the example predicts the detections on the next frame only implicitly from the embeddings that are also used for the detections on the current frame.

Deep Learning. In the last years, deep learning models became the most successful solutions to the first two sub-tasks. The reasons are the high performance of modern detection models that allow us to learn discriminative features that can be

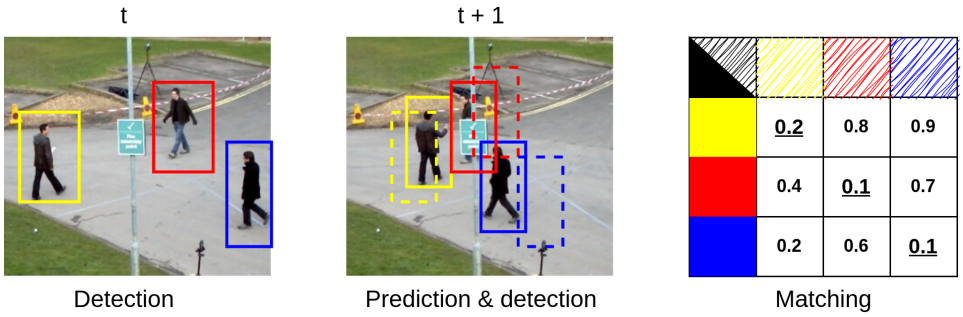

Figure 2.1: Online tracking. The three sub-tasks are track initialization (e.g. with detectors), prediction of the next position (with a motion model) and matching predictions with next detections (based on an appearance model) and a matching algorithm. The solid boxes are detections, the dashed boxes predictions and the right panel depicts the adjacency matrix for the bipartite matching problem between predictions and detections based on two adjacent frames. We match boxes with a high similarity score (small distance). The two images are from PETS-09 S2.L1 [25].

applied to various datasets [106], the ability to cope with high complexity in terms of motion and appearance patterns [52, Ch. 5 pp. 37], and larger datasets [22].

Multiple cameras. In many practical scenarios, it is necessary to detect objects in crowded and cluttered environments . However, traditional monocular tracking approaches fall short due to severe occlusions in such situations. Fortunately, real-world applications, especially for pedestrian tracking, provide an alternative, as image feeds from multiple cameras with overlapping fields of view are often available [33, p. 1]. These cameras are usually placed slightly higher than the average human height. By exploiting the scene's geometry from the cameras' calibration data and multiple perspectives, it is feasible to design tracking models that can offer dependable detection estimates in crowded scenes.

2.1.2 Evaluation

Various measures exist to assess the quality of a model, each focusing on certain tracking aspects. As a consequence, we have to rely on a multitude of measures and are usually unable to compare models in a clear ranking. We distinguish two approaches: Measures that follow the first approach focus on the ID association of detections between only two adjacent frames. The most relevant member of this group is MOT Accuracy. Measures that belong to the second approach take into

account the complete trajectory across multiple frames. Important measures in this group are the identification measures and the track quality measures.

MOT Accuracy (MOTA), as introduced by [9], is a measure based on frame-to-frame matching between track predictions and ground truth. Because it explicitly penalizes identity switches between adjacent frames, it is only a temporally local measure for tracking performance. It also tends to overemphasize object detection performance compared to the temporal continuity aspect of tracking. MOTA summarizes three different sources of errors. The measure is defined as

$$\text{MOTA} = 1 - \frac{\sum_t \text{FN}_t + \text{FP}_t + \text{IDSw}_t}{\sum_t \text{GT}_t}, \tag{2.1}$$

where t denotes the frame index and GT the number of ground truth objects on a frame. FN, the false negatives, is the number of undetected ground truth objects and and FP, the false positives, is the number of falsely detected objects that are not part of the ground truth. IDSw, the number of identity switches, counts how many times a trajectory changes from one ground-truth object to another [102]. MOTA can be negative if the tracker makes more errors than the number of all objects in the scene. Although MOTA does not properly balance the different sources of errors, it is considered the most expressive MOT measure.

Identification Precision, Recall and F1 Score. IDF1 [80] matches predictions to ground truth over the complete lengths of a trajectory. More precisely, it focuses on the ability to maintain track identity throughout the entire sequence. As a consequence, it overemphasizes the aspect of temporal continuity with respect to tracking performance.

To establish the predictions-to-ground-truth mapping, we solve a bipartite matching problem, connecting pairs with the greatest temporal overlap. Once the matching is established, True Positive IDs (IDTP), False Negative IDs (IDFN), and False Positive IDs (IDFP) can be computed, generalizing the concept of per-frame TPs, FNs, and FPs to tracks. Identification Precision (IDP) is expressed as

$$\text{IDP} = \frac{\text{IDTP}}{\text{IDTP} + \text{IDFP}}. \tag{2.2}$$

It measures the proportion of correctly identified object identities (IDTP - True Positive IDs) out of the total number of identities that the tracker has assigned (IDTP + IDFP - False Positive IDs). In other words, IDP indicates the accuracy of the tracker in maintaining consistent identities for the objects it detects.

On the other hand, Identification Recall (IDR) is given by

$$\text{IDR} = \frac{\text{IDTP}}{\text{IDTP} + \text{IDFN}}. \tag{2.3}$$

The recall measures the proportion of correctly identified object identities (IDTP) out of the total number of ground truth identities (IDTP + IDFN - False Negative IDs). IDR assesses the tracker's ability to capture all correct associations between object identities across frames.

We use both measures to calculate the F1 score for Identification (IDF1), which balances precision and recall and provides a single measure that reflects the overall accuracy of track identity maintenance:

$$\text{IDF1} = \frac{2 \cdot \text{IDTP}}{2 \cdot \text{IDTP} + \text{IDFP} + \text{IDFN}}. \tag{2.4}$$

Track Quality Measures. We can classify the ground-truth trajectory as mostly tracked (MT), partially tracked (PT), and mostly lost (ML). Based on the definition in [97], we consider a target mostly tracked if it is successfully tracked for at least 80% of its lifetime, and it is considered lost if it is covered for less than 20% of its total length. Partially tracked targets are those not in the above categories. We prefer a higher number of mostly tracked and fewer mostly lost targets. It is irrelevant whether the ID remains the same throughout the track for this measure. Mostly tracked and mostly lost targets are ratios of their total number to the total number of ground-truth trajectories.

The evaluation metrics apply to single camera and to multiple camera settings. However, for the latter we will rely on distributions and aggregates of these measures for all cameras of the same scene.

Our computations follow Appendix D in [22] and we will report IDF1, MOTA, MT and ML in percentage points and FP, FN and IDSw in absolute numbers.

2.2 Multiple Object Tracking

2.2.1 Related Work

We categorize the MOT literature in four tracking paradigms according to the recent trend to move past tracking-by-detection (Meinhardt et al. [64]).

Tracking-by-detection. Approaches that involve Tracking-by-detection create trajectories by linking a given set of detections over time. *Graphs* have been utilized to address track association and long-term re-identification by framing the problem as an optimization for maximum flow (minimum cost) [7]. The optimization is based on either distance-based costs [42, 74, 105] or learned costs [53]. Other methods involve the use of association graphs [86], learned models [48], and motion information [46], general-purpose solvers [104], multicuts [88], weighted graph labeling [39], edge lifting [41], or trainable graph neural networks [11, 93]. However, graph-based approaches can be computationally expensive, which limits

their use for online tracking. *Appearance*-driven methods rely on the increasing power of image recognition backbones to track objects by using similarity measures provided by twin neural networks [53], learned reID features [70, 79], detection candidate selection [15], or affinity estimation [18]. Appearance models have difficulty tracking objects in crowded scenarios where object-object occlusions are common and when objects reappear at a scene. *Motion* can be modeled to predict trajectories [2, 54, 81] by assuming constant velocity (CVA) [4, 17] or using the social force model [53, 73, 84, 102]. Learning a motion model from data [55] helps with track association between frames [105]. However, projecting nonlinear 3D motion [89] onto the 2D image domain remains a challenging issue for many models.

Tracking-by-regression. The Tracking-by-regression method does not link detections between frames but rather tracks objects by predicting their new positions in the current frame based on their past locations. Earlier attempts [8, 27] used regression heads on object features that were region-pooled. In [107], objects were depicted as center points that facilitated association through a distance-based greedy matching algorithm. To overcome the absence of object identity and global track reasoning, additional re-identification and motion models [8], as well as traditional [62] and learned [12] graph methods, have been required to achieve superior performance.

Tracking-by-segmentation. The Tracking-by-segmentation method not only predicts object masks but also utilizes pixel-level information to overcome problems with crowdedness and ambiguous backgrounds. Previous attempts have employed category-agnostic image segmentation [69], used Mask R-CNN [37] with 3D convolutions [92] and mask pooling layers [75], or represented objects as unordered point clouds [101] and cost volumes [98]. Nonetheless, due to the lack of annotated MOT segmentation data, current methods continue to depend on bounding boxes.

Tracking-by-attention. Attention is a technique used in image recognition to establish correlations among the input elements and is applied in Transformers (Vaswani et al. [91]) for tasks such as image generation [72] and object detection (Carion et al. [12] and Zhu et al. [109]). In the context of MOT, attention has thus far been utilized only to associate a given set of object detections [18, 109], without addressing the detection and tracking problem in a unified manner.

TrackFormer (Meinhardt et al. [64]), on the other hand, transforms the entire tracking objective into a single set prediction problem, employing attention not only for the association step but also for track initialization, identity, and spatio-temporal trajectories. This approach relies solely on feature-level attention, avoiding the need for additional graph optimization and appearance/motion models.

2.2.2 Datasets

Six notable single-camera MOT datasets are KITTI, DETRAC, and the four MOT Challenges. Note that PETS-09 [25] is a popular choice for single-camera MOT as well, although it is a multi-camera dataset.

KITTI [31] are benchmarks introduced for challenges in autonomous driving, including object detection, object tracking, scene flow and depth completion. The tracking benchmark consists of 21 training and 29 test sequences containing cars and pedestrians.

DETRAC [95] is a benchmark for vehicle tracking consisting of 100 sequences recorded from a high viewpoint.

The MOT Challenges' [22] objective is creating a common framework for testing tracking methods and gathering challenging sequences with different characteristics to develop more general tracking methods. Figure 2.2 gives an exemplary, visual overview of the different datasets.

MOT15 [56] includes 22 sequences from various sources including PETS and KITTI, and six new sequences collected by Dendorfer et al. [22]. The training data contains over 10 minutes of footage and 61,440 annotated bounding boxes to prevent over-fitting to specific sequences. Three sequences are particularly difficult due to low illumination, high pedestrian density, and motion blur from a moving camera.

MOT16 [65] has new and more challenging sequences than MOT15, with almost three times more bounding boxes for training and testing, filmed in high resolution and higher crowd density.

MOT17 contains the same sequences as MOT16, but with improved annotations and a different evaluation system that includes the evaluation of tracking methods using three different detectors. These detection sets are provided as public detections for MOT17 to challenge trackers further to be more general and work with detections of varying quality.

MOT20 [21] consists of eight sequences divided into training and testing sets, and filmed in three different scenes. The dataset contains approximately three times more bounding boxes for training and testing than MOT17, with high-resolution footage from an elevated viewpoint and a mean crowd density of 246 pedestrians per frame. Furthermore, the dataset includes annotations for other classes besides pedestrians, such as vehicles and bicycles.

Figure 2.2: One image from three example sequences for single-camera multiple object tracking from MOT15 [56], MOT16/17 [65] and MOT20 [21] with ground truth bounding boxes. ETH-Bahnhof depicts a street scene from a moving platform, MOT16/17 is filmed by a forward moving camera in a busy shopping mall, and MOT20-03 depicts people leaving an entrance of a stadium by night from an elevated viewpoint. The grey rectangles cover static objects that should not be tracked.

2.3 Multiple Camera Multiple Object Tracking

2.3.1 Related Work

Following Nguyen et al. [67], we categorize the literature in single view-based and centralized representation-based methods. The single view-based approaches focus on predicting tracklets directly on the image planes of the different cameras, whereas the centralized representation-based approaches aim track the objects one joint 3D world. Although the first category can be implemented in situations where cameras do not overlap, it does not make use of the camera parameters to explicitly facilitate cross-camera association and 3D space localization. The second category, however, utilizes camera calibration for merging tracklets and cross-camera association. Our work can be assigned to the second category.

Single view-based methods. In [100], a Hierarchical Composition of Tracklet (HCT) framework is introduced to match local tracklets using multiple object cues such as appearances and 3D positions. Another approach proposed in [99] uses a Bayesian formulation with a Spatio-Temporal Parsing (STP)-based tracking graph to prune matching candidates by exploiting semantic attribute targets. Similarly, [94] presents a dense sub-hypergraph search (SVTH) on the space-time-view graph using a sampling-based method. Recent methods include a semi-online Multi-Label Markov Random Field (MLMRF) approach [51], where the optimization problem over single detections is solved through alpha-expansion [10], and a non-negative matrix factorization technique (TRACTA) for grouping tracklets across cameras [38]. In contrast, DyGLIP [76] formulates the data association problem for multi-camera as a link prediction on a graph with tracklets as nodes. Although these techniques have demonstrated promising results on some datasets, they are prone to ID-switch errors in tracklet proposal generation, particularly in cluttered or crowded scenes as reported in [14].

Centralized representation-based methods. In order to estimate the occupancy map in 2D or occupancy volume in 3D, occlusion relationships between different detections have been taken into consideration. Explicit modeling is used in [30, 66] to construct occupancy maps by utilizing the foreground map obtained from background subtraction. Another technique, ground plane homographs, introduced in [47], generates a voting map from the foreground pixels in each view for constructing occupancy maps. A probabilistic approach is taken in [29], where GMLP [68] utilizes CNNs and Conditional Random Fields to model an occupancy volume map explicitly given detections from multiple cameras. More recently, the DMCT model [103] applied deep learning to directly compute the occupancy volume by fusing feature maps extracted from CNNs at multiple camera views.

At the intersection between both paradigms, Nguyen et al. [67] contribute a

novel multi-camera multi-object tracking approach using a spatial-temporal lifted multicut formulation. They refine tracklets from single-camera trackers to eliminate ID-Switch errors, leading to improved tracking graph quality and more accurate data association. The proposed method demonstrates near-perfect performance on the WILDTrack dataset, surpassing existing trackers on Campus and achieving comparable results on the PETS-09 dataset.

2.3.2 Datasets

In this section, we focus on multiple camera datasets with overlapping views. We use the term "overlapping" to describe multi-camera datasets where the fields of view of the cameras strictly overlap. The datasets are PETS-09, EPFL Campus, SALSA, CAMPUS, EPFL-RLC, WILDTRACK, and MMPTRACK. We do not consider the DukeMTMC dataset [80] to be in this category since only two of its cameras have a slight overlap in their fields of view.

PETS-09 [25] is a popular dataset for single and multiple camera models primarily for surveillance applications. It comprises three subsets: S1, targeting person count and density estimation; S2, targeting people tracking; and S3, targeting flow analysis and event recognition. The sequence S2.L1 is the most commonly used subset and features an overlapping camera setup. However, it has three disadvantages: First, a slope in the scene results in inconsistencies when projecting 3D points across the views. Second, the dataset is relatively small. And third, it was acquired in an actor set-up, which limits its ability to allow for good generalization and fair benchmarking of appearance-based methods.

EPFL Campus, SALSA and CAMPUS [29, 3, 100] are datasets that are not very crowded and that have only a small number of identities. Additionally, the coverage and image quality is limited. EPFL Campus contains the Laboratory, Terrace, and Passageway sequences that were shot at the university site. SALSA features a static cocktail party, making it less challenging for tracking. Campus does not provide annotations of the 3D locations of people.

EPFL-RLC [13] improves upon PETS-09 in terms of joint-calibration accuracy and synchronization. Yet, it is a collection of positive and negative multi-view annotations used for pedestrian classification and only provides full ground-truth annotations for a small subset of the frames. Furthermore, it is acquired using only three cameras with a much smaller field of view compared to the following datasets, resulting in significantly fewer detections per frame.

WILDTRACK [14] improves upon previous multi-camera person datasets due to its high-precision calibration and synchronization between cameras, and the large number of annotations that enable the development of deep learning-based multi-view detectors.

MMPTRACK [33] features a significant number of calibrated, overlapped cameras within indoor environments. This characteristic makes it well-suited for use in applications such as frictionless checkout. Compared to existing datasets, MMPTRACK stands out due to its larger size, encompassing both longer videos and a greater number of annotated frames.

Table 2.1 summarizes four important datasets with camera overlap and geometric calibration data. WILDTRACK provides the highest resolution frames and FPS rate, whereas MMPTRACK consists of sequence sets in five different environments, 23 different views and video lengths up to 172 minutes. Both datasets provide annotations for every frame. In the thesis, we focus on the WILDTRACK dataset because it gives us high quality image and calibration data, and manageable training time. Figure 2.3 depicts example images from 4 of 7 cameras and shows how we can track objects that are occluded in one view with the help of the calibration data. Without the calibration data, models have to rely solely on appearance similarities across the cameras.

	Resolution	# Envs.	# Cams.	FPS	Annot. (frames)	Lengths (min/cam)
PETS-09 S2.L1	720×576	1	7	7	$\sim$ 1/3	43−105
CAMPUS	1920×1080	3	3−4	30	1	3−4
WILDTRACK	1920×1080	1	7	2	1	$\sim$ 9
MMPTRACK	640×320	5	23	15	1	73−172

Table 2.1: Multiple camera datasets with camera overlap and calibration data. Features are camera resolution, number of different environments, number of cameras, FPS (frames per second), share of annotated frames, and video lengths in minutes per camera.

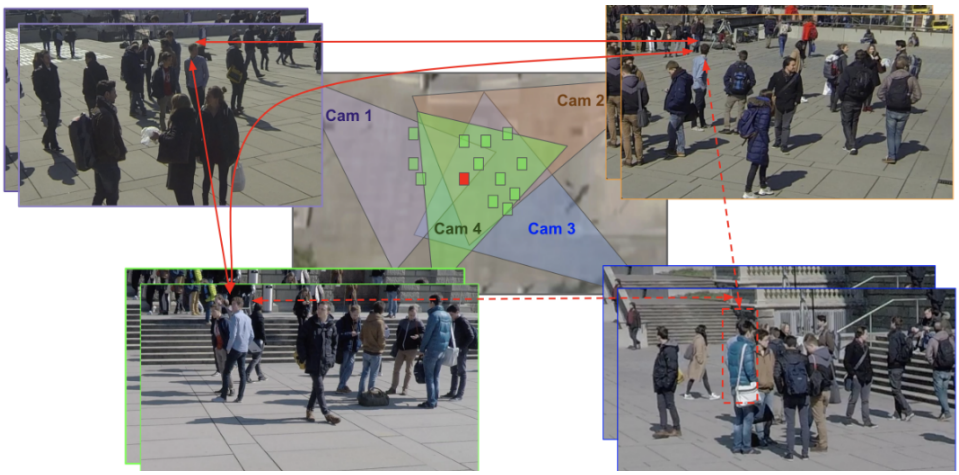

Figure 2.3: Multiple-camera tracking with overlapping cameras on WILDTRACK. The figure shows four of seven views (exterior images) of the same scenery with an overlap (central map). We can track a target pedestrian (red rectangle) in the intersection area that is occluded in Cam 4 (green) because it is captured by the other cameras from a different angle (dashed rectangle, Cam 3). In addition to the appearance similarity across the different views, the calibration data depicted on the map helps us find the correspondence between pedestrians from different views (red arrows). Image source: Nguyen et al. [67].

Chapter 3

Background: Deep Learning

This chapter introduces the foundational concepts and notation of deep neural networks within the context of machine learning. Our primary sources of reference include the "Deep Learning" book by Goodfellow, Bengio, and Courville [32], the CS231n lecture notes authored by Li, Wu, and Gao [58], and the dissertation by Karpathy [44]. Additionally, we draw insights from various other resources such as [6] for backpropagation, [43] for convolutional neural networks, [5] for recursive neural networks, and [96, 45, 90] for transformers.

3.1 Supervised Learning

Prediction rules based on some specific set of information can be written as a mapping $f : X \to Y$, where X is an input space and Y is an output space. To recognize a dog on a photo, for example, X would be the space of images and Y would be a probability interval $[0, 1]$ for the presence of a dog. However, it is oftentimes very difficult to find an explicit function f from theoretical considerations about the problem. For problems where it is easy to find many examples $(x, y) \in X \times Y$, the supervised learning approach is usually well-suited. In our example, the requirement would be a dataset that consists of a large number of images which are annotated with presence or absence of a dog.

Loss function. To be concrete, our dataset $\{(x_1, y_1), ...(x_n, y_n)\}$ includes n examples. Our theoretical assumption about the data is that these examples are drawn from a data generating distribution D with independent and identically distributed (i.i.d.) random variables, i.e. $(x_i, y_i) \overset{\text{iid}}{\sim} D$. In the supervised learning context, learning the mapping $f : X \to Y$ means selecting the function f from a set of candidate functions $\mathcal{F}$ so that f yields the best predictions for y given any x. i.e. we want to find the best approximation for $f(Y|X)$. We achieve this by

selecting a scalar-valued loss function $L(\hat{y}, y)$ which measures the difference between prediction $\hat{y}_i$ and actual outcome y_i. In theory, our objective is to find the function with the lowest expected loss for all examples from the data generating distribution D. In practice, however, we have to rely on our dataset. Therefore, we approximate our theoretical objective using the assumption that our data is drawn from an i.i.d. distribution. We search for function f^* with the lowest average loss over the data:

$$f^* \approx \arg\min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} L\big(\, f(x_i), y_i\big). \tag{3.1}$$

Regularization. Oftentimes, the i.i.d. assumption is too strong. In this case, we usually do not achieve the best result for predicting unseen outcomes with a function that is optimized solely with regards to our dataset. In short, this function does not necessarily generalize well. A further issue is how to choose between multiple functions with the same minimal loss. The approach that addresses both problems at once is regularization. The idea is to add a regularization penalty to our objective function from which we obtain our predictor. The penalization criterion $R(f)$ is function complexity. We thus search for the function that fits the data best *and* has a low complexity:

$$f^* \approx \arg\min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} L\big(f(x_i), y_i\big) + R(f), \tag{3.2}$$

where $R(f)$ is a scalar-valued function. With regularization, we can achieve a better generalization and choose between functions that achieve a similar loss.

Frequently, we have already selected our model but not its parameters. Hence, we take function f as given but we want to learn the best parameters θ for this function. In this situation, the concept of loss and regularization directly translates from finding the optimal function to finding the optimal parameters:

$$\theta^* \approx \arg\min_{\theta \in \Theta} \frac{1}{n} \sum_{i=1}^{n} L\big(f(x_i; \theta), y_i\big) + R(\theta), \tag{3.3}$$

where Θ denotes the parameter space. Common examples for $R(\theta)$ are multiples of vector norms. In this setting, choices like the model, the loss or the regularization are called hyperparameters. These are parameters that are set before the actual model parameters are learned. Hyperparameter choices are oftentimes critical to the quality of the learned model. There are several other ways to prevent overfitting the model besides regularization. Examples are choosing simpler models, stopping the optimization process early, changing or disabling some model units during training (dropout), and dataset augmentations.

Model validation. How do we test whether our model generalizes well to unseen data? The usual machine learning approach is as follows: at first, we split our examples in three groups: a large training, and smaller validation and test sets. Secondly, we learn the model parameters with the training data. The third step is to compute evaluation metrics for this model from the unseen test data. In general, we want to minimize the distance between the prediction and the target vector. For classification tasks, we can use evaluation metrics based on the confusion matrix. This enables us to identify single classes which are more difficult to predict for the model. Optionally, we can repeat the third step multiple times for different hyperparameter configurations and compare the generalization errors. Then, we evaluate the best model once again on the validation data and report its performance. We will briefly discuss hyperparameter selection in the end of the optimization section.

Example: Linear regression. Assume we have a dataset with 100 observations ($n = 100$) of two features each ($m = 2, X = \mathbb{R}^2$) and a scalar annotation ($Y = \mathbb{R}$). According to the source and target space, we choose to restrict the candidate class $\mathcal{F}$ to the family of linear functions, i.e. $\mathcal{F} = \{w^T x + b | w \in \mathbb{R}^2, b \in \mathbb{R}$. With this choice, we set our hypothesis space from a class of functions to the set of three parameters $\theta = \{w_1, w_2, b\}$ with $w = [w_1, w_2]$. Next, I list two common hyperparameter choices. First, the squared difference between predicted value and target, $L(\hat{y}, y) = (\hat{y} - y)^2$, is a common loss function. Second, the L2 norm of the weights multiplied with importance parameter λ is a typical regularizer choice, i.e. $R(w, b) = \lambda(w_1^2 + w_2^2)$. The L2 norm counteracts extreme weights and an excessive effect of one single weight on the prediction $\hat{y}$. Taken together, our objective is

$$\theta^* = \arg\min_{w,b} \underbrace{\left[\frac{1}{n}\sum_{i=1}^{n}(w^T x_i + b - y_i)^2\right]}_{\text{data fitting}} + \underbrace{\left[\lambda(w_1^2 + w_2^2)\right]}_{\text{regularization}}. \tag{3.4}$$

Example: Neural network regression. Perhaps the relationship between features x_1, x_2 and the scalar target y is not liner and there are interactions between the two features. A model class that is well-known for its theoretical capabilities of approximating continuous functions are feedforward neural networks (FNN). As a preview, we can extend the previous example to a specific FNN called neural network regression. It has the form $f(x; \theta) = w_2 \tanh(W_1^T x + b_1) + b_2$ with parameters $\theta = \{W_1, b_2, w_1, b_1\}$. W_1 is matrix with dimension $H \times 2$, b_1 and w_2 are both vectors of length H, and b_2 is a scalar. H is an integer-type hyperparameter. $W_1^T x + b_1$ is often called a hidden layer. Its elements, or neurons, are outputs of multiplicative interactions between elements from previous layers, in this case the two inputs x_1 and x_2. tanh denotes the hyperbolic tangent that squashes elements

from the real-valued domain to the interval [-1,1]. It is applied element-wise and introduces non-linearity to the model. The objective is given by:

$$\theta^* = \arg\min_{W_1,b_2,w_1,b_1} \underbrace{\left[\frac{1}{n}\sum_{i=1}^{n}\left(w_2 \tanh(W_1^T x_i + b_1) + b_2 - y_i\right)^2\right]}_{\text{data fitting}} + \underbrace{\left[\lambda\left(||W_1||_2^2 + ||w_2||_2^2\right)\right]}_{\text{regularization}}. \tag{3.5}$$

Example: Neural network classification. Oftentimes, we do not want to predict a real number but we want to predict whether an input corresponds to an output of a specific class k of K possible classes. For instance, we may want to predict whether a dog, a, cat, or a budgie is shown in a picture. To formalize this problem, we encode our classes as non-negative integers starting at 0. We also encode our output as a vector of $|K|$ zeros, where only the k^*-th element representing the actual class encoded as k^* is set to one. Using the previous order, a picture that shows a dog is encoded as $y = [1, 0, 0]$. We can achieve a model that predicts a vector of this form with two simple adjustments of the neural network regression model. The first adjustment is replacing vector w_2 by matrix W_2 with dimensions $K \times H$. This gives us a real-valued vector z with one real-valued element for every class. The second adjustment is that we compute class probabilities by applying the softmax function to z, i.e. $\hat{p}_k = e^{z_k} / \sum_{i=1}^{K} e^{z_i}$. Handy properties of the softmax function in the probability context are, first, that every element is mapped to $[0, 1]$, and second, that the sum of all elements is normalized to one. Hence, our class prediction would be the class that is represented by the largest element in vector $\hat{p}$. For instance, with $K = 3$ and $\hat{p} = [0.1, 0.7, 0.2]$, our prediction $\hat{y}$ equals $[0, 1, 0]$. The most common loss function for classification is the cross-entropy loss. It takes the predicted class probabilities $\hat{p}$ instead of the predicted class vector $\hat{y}$. We denote both options by output o:

$$L(o, y) = L(\hat{p}, y) = -\sum_{k=1}^{K} y_k \log \hat{p}_k = -\log \hat{p}_{k=k^*}. \tag{3.6}$$

The first equality is the cross-entropy definition for two distribution, i.e $H(q, r) = -\sum_x q(x) \log r(x)$. The second equality shows that the loss equals the negative log probability from vector p at the position of the actual class k^*. It follows from the fact that target distribution is degenerate. This means that only the true class has probability one. Additional motivation for choosing the cross-entropy for

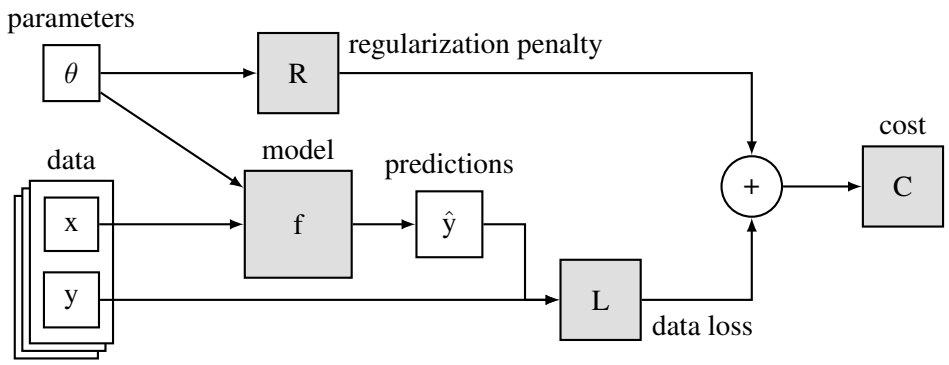

Figure 3.1: Computational graph for a general supervised learning approach. Examples $\{x_i\}_{i=1}^n$ and parameters θ are taken by model f to predict the targets by $\{\hat{y}_i\}_{i=1}^n$. Data loss L computes the difference between the predictions and the targets $\{y_i\}_{i=1}^n$. Regularization loss R penalizes extreme parameters. The sum of both penalties is given by cost C. Oftentimes we use predicted class probabilities $\hat{p}$ instead of (rounded) predictions.

classification problems is that minimizing this loss is equivalent to maximizing the likelihood of observing model parameters θ conditional on predicting the true class label.

Summary. Supervised learning requires a dataset of n examples $\{(x_1, y_1)$, ..., $(x_n, y_n)\}$, where $(x_i, y_i) \in X \times Y$. y_i represents the annotation that we want to predict based on features x_i. Finding a mapping $f : X \to Y$ is a two-step process. First, we have to formalize the problem by choosing

1. The search space of functions $\mathcal{F}$ with $f \in \mathcal{F}$.
2. The scalar-valued loss function $L(\hat{y}, y)$ that measures the difference between the network's predictions $\hat{y} = f_\theta(x)$ and the target y.
3. The scalar term $R(f)$ that penalizes overly complex functions.

In deep learning without architecture search, the space of functions is one neural network f_θ with some parameters $\theta \in \Theta$. Putting these parts together, the second step is to find these parameters from the optimization problem $\theta^* = \arg\min_{\theta \in \Theta} C(\theta)$ with $C(\theta) = \frac{1}{n} \sum_{i=1}^n L\big(f_\theta), y_i\big) + R(f_\theta)$. We call $C(\theta)$ the cost function.

3.2 Optimization

In the previous section we formalized supervised learning of a predictive model as solving the optimization problem $\theta^* = \arg\min_{\theta \in \Theta} C(\theta)$, where θ denotes the model parameters and cost function C represents the average loss of all examples plus a regularization penalty. In realistic settings, there is no closed form solution to this problem. Therefore, we have to rely on schemes that iteratively proposes new parameters given the previous choice or the initial guess. We want to choose a method that proposes new candidates with a high likelihood of reducing the loss compared to the current parameters, so we can replace them.

First order methods. In practice, a neural network is a composition of many small functions that are easy to differentiate. Therefore, we can compute the gradient $\nabla_\theta C$ from our network via the backpropagation technique. We will discuss this method in detail in the next section. The gradient of our cost function is a vector of first-order partial derivatives. It contains the direction of its fastest increase and its magnitude is the rate of increase in that direction. The gradient at a minimum of the loss function equals zero. Therefore, we can use the negative gradient as search direction for selecting the next proposal θ_{i+1} from current candidate θ_i. This is the basic idea of the gradient descent algorithm. The method alternates between two steps: 1.) Compute the gradient $\nabla_\theta C(\theta)$. 2.) Update θ by subtracting a small multiple of the gradient. Many applications use very large datasets. For instance, the number of training images in ImageNet is about 1 million. Therefore, it is handy to approximate the loss gradient from a small minibatch of examples (e.g. 100). With that, we can update θ many times for every epoch. An epoch is one iteration over the complete training set. In practice, a large number of approximate updates works better than a small number of exact updates. This algorithm is called Stochastic Gradient Descent (SGD). It is summarized in Algorithm 1.

Algorithm 1: Stochastic Gradient Descent

```
// Initialize parameters and learning rate
```
Let $\theta \in \Theta, \eta \in \mathbb{R}^+$
repeat
1. Sample a minibatch from the training data
2. Estimate gradient $\nabla_\theta C(\theta)$ only from minibatch data with backpropagation
3. $\Delta\theta = -\eta \nabla_\theta C(\theta)$
4. $\theta := \theta + \Delta\theta$

until convergence

A crucial parameter for gradient descent algorithms is the learning rate, or step

size, η. If we set it too small, the optimization requires too many steps. If we set it too large, the algorithm may not converge or even diverge. As an illustration, consider the following toy example with convex objective function: Let $f = x^2$ with gradient $\frac{df}{dx} = 2x$ and initial parameter value $x = 1$. With $\eta < 1$ the algorithm finds $x^* = 0$. However, with $\eta = 1$, it oscillates between $x = -1$ and $x = 1$, and with $\eta > 1$ it diverges to $x = \infty$. Generally, finding a useful learning rate depends on the objective function.

Advanced first-order methods. Oftentimes, we can achieve faster convergence speed with modified formulations of the update direction $\Delta\theta$ from Step 3 in Algorithm 1. The two main ideas are to use weighted moving averages of all gradients so far and to compute parameter-specific learning rates. Common variants are Momentum [87], Adagrad [23], RMSProp [40] and Adam [49]. The **Momentum** update is inspired by the physics notion of momentum. The current update direction is not determined by the current gradient but also by the previous gradients. The impact of the previous gradients, however, decays exponentially for every previous iteration. Let α be the decay parameter and $g := \nabla_\theta C(\theta)$. We then replace Step 3 by two steps: First, we compute the "velocity" $v = \alpha v + g$ (initialized at zero), and second, we compute the parameter update with $\Delta\theta = -\eta v$. **Adagrad** introduces element-wise learning rates for the gradient. I.e. we weigh every partial derivative with the moving average of its squared sum. The motivation is two-fold: first the shape of the objective function can vary between different dimensions. Therefore, dimension-specific learning rates may improve the algorithm. Second, we can achieve a more equal exploration of each dimension by equipping dimensions with a history of small gradients with larger step sizes and vice versa. Formally, Adagrad first introduces the intermediate variable $r = r + g \odot g$ where $\odot$ denotes elementwise multiplication. Second, it computes the gradient update with $\Delta\theta = -\frac{\epsilon}{\delta+\sqrt{r}} \odot g$. In the last expression, δ is a small number to avoid division by zero. **RMSProp** introduces an additional hyperparameter to weigh Adagrad's running average, i.e. $rs = \rho r + (1-\rho) g \odot g$. Lastly, the **Adam** update combines gradient momentum and RMSProp's dimension-specific learning rates.

Hyperparamter optimization. In the discussion of supervised learning and optimization, we encountered many configurations that we did not specify. Examples are the number of hidden units H in our neural network, the regularization strength λ, or the learning rate η in gradient descent. These parameters either change the composition of the model parameters λ or impact their selection during training. Therefore, they have to be set before a model is trained. To this end, we have to select our hyperparameter space $\mathcal{H}$. Due to restrictions of time and computational power, this choice is often based on experiences with similar models from previous research. Principled approaches range from model-free

methods like random search to global optimization frameworks like Bayesian optimization. Model-free methods are simple and do not use the evaluation history whereas global optimization techniques additionally consider the uncertain trade-off between exploring new values and exploiting good values that have already been found. I recommend the textbook chapter by Feurer and Hutter [28] for more explanations and concrete examples.

3.3 Backpropagation

We learned that we can find a mapping $f \in \mathcal{F}$ that maps features X to outcome Y consistent with the data by minimizing the cost function with repeated gradient evaluations using a gradient descent optimizer.

We compute the gradient with backpropagation. This algorithm allows us to efficiently compute gradients of functions that 1.) are scalar-valued, 2.) have many input parameters, and 3.) can be decomposed into simpler intermediate functions that are differentiable. The mathematical formula of backpropagation is inspired by the third property. It is the gradient computation via recursive applications of the chain rule from calculus. The algorithmic order is inspired by the first two properties. The idea is to efficiently compute the resulting derivative starting from the intermediate functions on the parameter instead of the cost function side.

Toy example. Recall that we would like to compute the gradient of cost function $C(\theta)$, which takes not only parameters θ but also multiple examples (x_i, y_i) as input. Specifically, our first-order optimizer requires the gradient of the cost function with respect to the model parameters $\nabla_\theta C(\theta)$ in order to update the parameters θ. Let us consider the following example. Let $C(\theta_1, \theta_2) = \theta_1\theta_2 + \tanh(\theta_1)$ be the cost function of a neural network with parameters θ_1 and θ_2. Figure 3.2 shows that we can view this equation as a computational graph with cost function C as root and parameters as leaf nodes. We introduce intermediate variables to write C as a sequence of the intermediate functions $z_3 = \tanh(\theta_1)$, $z_4 = \theta_1\theta_2$, $z_5 = z_3 + z_4$, and $C(\theta) = z_5$. For ease of notation, we will later also write θ_1, θ_2 and $C(\theta)$ as z_1, z_2 and z_6. We can write the gradient of the cost function with respect to each parameter as a combination of the gradients of its parent nodes using the chain rule from calculus and the fact that multiple occurrences of a term add up in its derivative. This is shown in Equation Equation 3.7 and Equation 3.8:

$$\frac{\partial C}{\partial \theta_1} = \frac{\partial C}{\partial z_5}\left(\frac{\partial z_5}{\partial z_3}\frac{\partial z_3}{\partial \theta_1} + \frac{\partial z_5}{\partial z_4}\frac{\partial z_4}{\partial \theta_1}\right) = 1 - \tanh(\theta_1)^2 + \theta_2, \tag{3.7}$$

$$\frac{\partial C}{\partial \theta_2} = \frac{\partial C}{\partial z_5}\frac{\partial z_5}{\partial z_4}\frac{\partial z_4}{\partial \theta_2} = \theta_1. \tag{3.8}$$

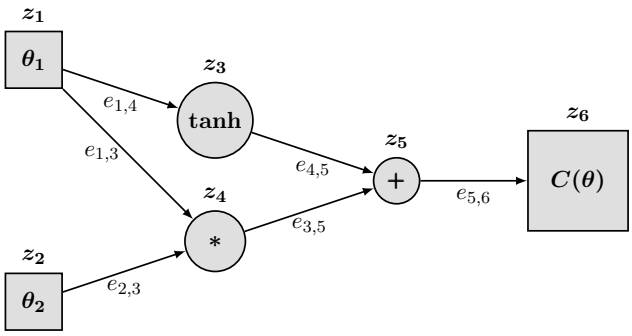

Figure 3.2: Computational graph for a toy example of a neural network's forward pass without vector-valued intermediate functions, data, regularization and loss. The function given by the graph is $C(\theta_1, \theta_2) = \theta_1\theta_2 + \tanh(\theta_1)$. Intermediate functions that are relevant for deriving the gradient with the chain rule are denoted by z_i with $i \in \{1, ..., 6\}$. The edges along which the intermediate evaluations move forth during the forward pass and along which the gradients move back during the backward pass are denoted by $e_{i,j}$ with $i, j \in \{1, ..., 6\}$. We can view two edges leaving one node as a a shortcut depiction for applying the `duplicate` or fork operation along both edges.

Vector-valued intermediate functions. The toy example deviates from a realistic neural network application in a few aspects. One of these aspects is that we usually consider vectors of large parameter groups. Another aspect is that the intermediate functions that transform the parameter vector θ step-by-step are usually vector-valued (unlike the final cost function). Apart of confluence operators like "+" and "*" that usually combine different parameter groups, the final gradients can be written as a sequence of "local" gradient computations. Due to vector-valued functions and the number of parameters, these computations are written as matrix multiplications. We look at this extension more formally. To this end, let z_0 be the input vector which we transform through a series of functions be $z_i = f_i(z_{i-1})$ where $i = 1, ..., k$ and only the last z_i can be scalar. Assuming that the functions f_i are once differentiable, we can compute the Jacobian matrix $\frac{\partial z_i}{\partial z_{i-1}}$ for all intermediate functions. This will give us the values of the first derivative of every output dimension of z_i depending on each single input dimension of z_{i-1}. From the multivariable chain rule, we obtain the result that the gradient of our final function with respect to input vector equals the product of all intermediate Jacobians: $\frac{\partial z_k}{\partial z_0} = \prod_{i=1}^{k} \frac{\partial z_i}{\partial z_{i-1}}$.

Reverse accumulation. We can compute $\frac{\partial C}{\partial \theta}$ in two different ways. In this section we learn that one approach is much more efficient. The reason is the the

structure of our problem: we minimize a scalar-valued function with a large number of parameters. One approach is to compute the gradient from parameters to output: $\frac{\partial C}{\partial \theta} = \frac{\partial z_k}{\partial z_{k-1}} \cdot ... \cdot \frac{\partial z_2}{\partial z_1}$. This is called forward accumulation. The other approach, reverse accumulation, is to compute the gradients from output to parameters: $\frac{\partial C}{\partial \theta} = \frac{\partial z_2}{\partial z_1} \cdot ... \cdot \frac{\partial z_k}{\partial z_{k-1}}$. The first approach is less efficient but more intuitive. It is more intuitive because the derivatives can be computed in sync with the evaluation steps. This makes it easier to think about how confluence operations like "$+$", "$*$", the fork operation `duplicate`, or filter operations like `max` or `average` transform the gradients. We will later see that it is crucial to use these operations in a careful way in the architecture design of neural networks because they can have large effects on the model performance. In reverse accumulation, we fix the dependent variable to be differentiated and compute the derivative with respect to each intermediate function recursively. Figure 3.3 shows how we can calculate the two gradients of our toy example step-by-step with this method. There are two important things to note. First, computing the backward operations corresponding to the forward operations is prone to error. This is one reason why this should be done automatically by a graph-based computer program. Second, we can re-use our intermediate computations $\bar{z}_3$ to $\bar{z}_5$ for both gradients based only on one evaluation of the cost function $C(\theta) = z_6$. In realistic neural network applications, we are able to re-use a large number of intermediate results based on only one forward evaluation for an even larger number of input parameters. In contrast, with forward-mode accumulation, computing the gradient requires to evaluate each intermediate function with the whole parameter vector. Although re-using the intermediate gradients requires more storage, the reverse-mode accumulation strategy is much more efficient for functions like neural networks where the number of output values is much smaller than the number of input values.

Implementation. In our toy example, we have already discovered that it is easier to think of the evaluation and differentiation of a neural *network* in terms of a directed graph of operations instead of a linear sequence of function applications. In this graph, the nodes stand for differentiable operations that take a number of vectors from its incoming edges, transforms and potentially interrelates them, and sends the result to the next nodes along its outgoing edges. The graph abstraction translates to most implementations of the backpropagation algorithm. A *Graph* object keeps the connections (e.g. `duplicate` or $+$) between the nodes and the collection of operations (e.g. `tanh`). Both Node and Graph objects implement a `forward()` and a `backward()` method. The Graph's `forward()` calls the Nodes' `forward()` methods in their topological order. With that, every Node computes its operation on its input, and the Graph sends it to the next Node. The Graph's `backward()` iterates over the nodes in reverse topological order and calls their `backward()` methods. In the backward pass, each Node is given the

Evaluation steps (forward pass)	Derivation steps (backward pass)
$z_1 = \theta_1$	$\bar{z}_5 = 1$ (assumption)
$z_2 = \theta_2$	$\bar{z}_4 = \bar{z}_5$
$z_3 = z_1 * z_2$	$\bar{z}_3 = \bar{z}_5$
$z_4 = \tanh z_1$	$\bar{z}_2 = \bar{z}_4 * z_1 = \theta_1$
$z_5 = z_3 + z_4$	$\bar{z}_1 = \bar{z}_4 * z_2 + \bar{z}_4 * (1 - \tanh(z_1)^2)$
	$= \theta_1 + (1 - \tanh(\theta_1)^2)$

Table 3.1: Gradient computations in reverse accumulation mode for toy example $C(\theta_1, \theta_2) = \theta_1\theta_2 + \tanh\theta_1$. The left column shows the evaluation and the right column depicts the derivation steps. $\bar{z}$ denotes the derivative of the cost function with respect to an intermediate expression, i.e. $\frac{\partial C}{\partial z}$. In the backward pass, we first evaluate the scalar-cost function and use its derivative with respect to z_5 at the top of the graph. Here we assume $\bar{z}_5$ equals 1. In the next step, we combine the chained gradients of the intermediate expressions according to the respective backward functions of the `duplicate` and the confluence operations "$+$" and "$*$" from top to bottom. Addition distributes the upstream gradient down to all its inputs. Multiplication passes the upstream gradient multiplied with the *other* input back. I.e. $\frac{\partial(z_3+z_4)}{\partial z_4} = 1$, $\frac{\partial(z_1*z_2)}{\partial z_2} = z_1$, and the backward function of $(z_{1,3}, z_{1,4}) =$ `duplicate`(z_1) equals $\frac{\partial z_3}{\partial z_1} + \frac{\partial z_4}{\partial z_1}$, where $z_{1,3}$ and $z_{1,4}$ denote abbreviated nodes from the implicit duplication operation in the first and third evaluation step.

gradient of the cost function with respect to its output and it returns the "chained" gradients with respect to all its inputs. "Chained" means that the Node multiplies the Jacobian that it received from its parent nodes with its own local Jacobian. The Graph sends the resulting product to the Nodes' children and the process repeats until the recursion ends at the last computation which includes the data and the current parameter values. At last, the optimizer updates the parameter vector based on the final gradient (Step 3 in Algorithm 1). Note that the implemented compute graph for Figure 3.2 would in practice be larger and rudimentary because we can decompose many operations, like divide or subtract in tanh, into more basic operations.

3.4 Neural Networks

In the last sections we learned that we can compute any differentiable loss function between an arbitrary differentiable function f that takes input x and outputs predictions $\hat{y}$ and the data (x, y), and optimize the model f with respect to its parameters θ with stochastic gradient descent. In this section, we look at how to construct f as a neural network.

3.4.1 Vanilla Neural Networks.

In the two examples from Section 3.1, neural network regression and neural network classification, we have already discovered one main idea of vanilla neural networks: combining matrix multiplications and element-wise non-linearities. The other idea is that we can repeat, or layer, these two transformations multiple times. For instance, abstracting from the concrete structure of the source and target data, we would write a neural network with two fully connected layers as $f(x) = W_1\phi(W_1x)$, where ϕ represents an an element-wise non-linearity like tanh and W_1, W_2 are matrices that interact, scale, and shift the inputs. A 3-layer neural networks would be implemented as $f(x) = W_3\phi(W_1\phi(W_1x))$, and so forth. The outputs from the intermediate functions are called hidden layers, and one output a hidden unit. We can think of hidden units as feature abstractions from the previous layer, or latent features for the next layer. During training, the neural network learns which feature abstractions are useful to the next layer. A network is called deep if it has more than one hidden layer. Common choices for non-linearity ϕ are tanh, the rectified linear unit (ReLU) $\max(0, x)$, and the logistic function $1/(1 + e^{-x})$. Usually, we add an additional element $x_0 = 1$ to the input vector. The corresponding weight, or bias, $b := w_0$ shifts the output. The choice of the last layer depends on the type of output data y. For instance, we could select

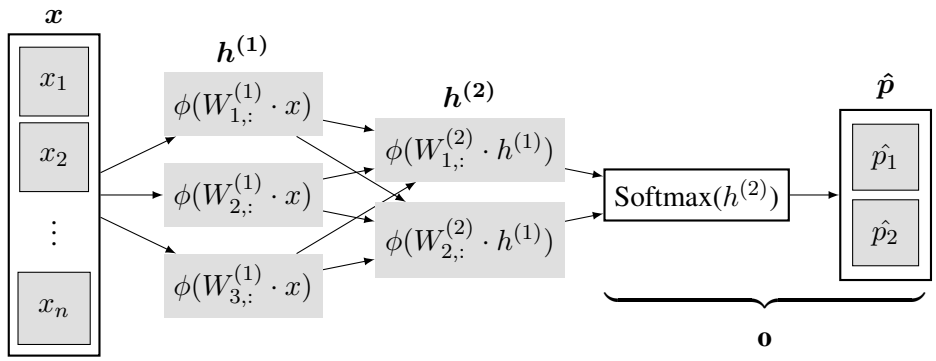

Figure 3.3: Vanilla neural network with two fully-connected hidden layers for binary classification. The first hidden layer, $h^{(1)}$, is composed of three neurons. Each neuron takes the input vector x and compute the dot product with its weight vector from its respective column of the layer's weight matrix $W^{(1)}$. Moreover, $h^{(1)}$ introduces non-linearity via an elementwise non-linear operation ϕ. The second layer, $h^{(2)}$, repeats the same process with the previous hidden layer as its input but reduces the number of hidden features. The output layer o has the same size as the last hidden layer and computes the Softmax probabilities for classes one and two.

the logistic function for binary classification, the softmax function for multi-class classification, and a linear layer to predict natural numbers. Similar to our toy example in Figure 3.2, we can depict a vanilla neural network as a directed acyclical graph and compute its gradient via backpropagation in a supervised learning setting. Figure 3.3 depicts a two-layer example architecture for a binary classification task. Note the similarity between this network and the second example from Section Section 3.1. Figure 3.3 clarifies how we aggregate dot product operations on the neuron level to matrix operations on the layer level. In the last decade, the neural network approach has led to state-of-the-art results in areas with large amounts of high-quality data such as computer vision, natural language processing, speech recognition and others. A theoretical reason for this success is that neural networks can achieve universal approximation. This means that they can approximate any continuous function, either via sufficient depth (number of layers; e.g. [19]) or width (number of columns in weight matrices; e.g. [34]). A practical reason is that we can optimize these functions very efficiently with many parallel computations on modern hardware.

Backward pass. To improve our understanding about backpropagation of neural networks, we look at the partial derivatives of typical layers. Note that we usually consider the derivatives with respect to the loss instead of the cost function

because the regularization terms are not complex and simply add up to the more complicated loss derivative at the end of each update computation.

ReLU: the derivative of the element-wise ReLU is given by $\frac{\partial ReLU}{\partial z} = 0$ if $x \leq 0$ and 1 if x >0. As a consequence, we propagate only the gradient for the neurons with positive activations back to the previous layer.

Linear layer: Let $z = Wx$ be a linear layer with one input channel and n elements, $x \in \mathbb{R}^{n\times 1}$, with $W \in \mathbb{R}^{m\times n}$ and let $z, \delta := \frac{\partial L}{\partial z} \in \mathbb{R}^{m\times 1}$. Then $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial W} = \delta x^T$. The same result, δX^T, holds for linear layers with k features or input channels, i.e. with $X \in \mathbb{R}^{n\times k}$ and $Z, \delta \in \mathbb{R}^{m\times k}$. Hence, *each* input receives the input-specic respective weighted sum of the upstream gradient of *all* neurons from the next layer.

Softmax: Let $\hat{p} = \text{softmax}(z)$ denote the softmax probabiltities with $\hat{p}, z \in \mathbb{R}^{n\times m}$ and let $L(\hat{p}, y)$ with $y \in \mathbb{R}^{n\times 1}$ be the scalar cross-entropy loss. We get $\frac{\partial L}{\partial z} = \frac{1}{n}(\hat{p} - y \otimes \vec{1}^m)$, where $\otimes$ denotes the outer product between two vectors. If we add another logit layer $l \in \mathbb{R}^{n\times m}$ below the softmax, we obtain the following simple result:

$$\frac{\partial L}{\partial l_{i,j}} = \begin{cases} \hat{p}_{i,j} & \text{if i} \neq \text{j} \\ \hat{p}_{i,j} - 1 & \text{if i = j.} \end{cases} \tag{3.9}$$

Intuitively, for each example $i \in 1, ..., n$, the gradients for each parameter that flow from each predicted probability is increased by the amount that the predicted probability differs from the actual label (times $1/n$). Therefore, gradient descent in particular updates parameters θ towards the direction of the gradients from the bad predictions.

3.4.2 Convolutional Neural Networks

A Convolutional Neural Networks (CNN) [57] is type of neural network architecture that is specifically designed for source data where the tabular arrangement includes a spatial meaning. Moreover, the features of one example are not stored in a vector of different property values but x is a multi-dimensional array (i.e. a *tensor*). For instance, the data for a color image could have dimensions $32 \times 32 \times 3$. The first two dimensions represent the height and the width of the pixels and the third dimension stands for three color channels red, green and blue. We can find these spatial relationships in many data, not only in images. Further examples are sound spectrogram or sentence data. The main motivation for CNNs is to reduce

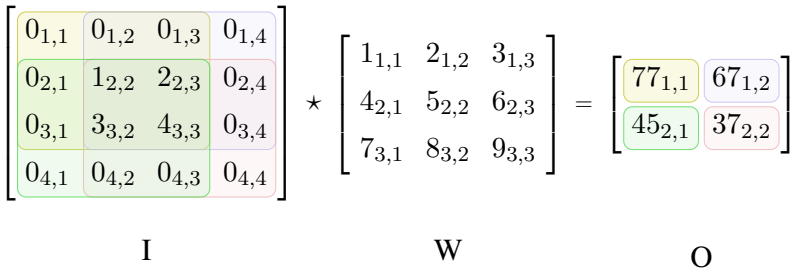

Figure 3.4: Cross-correlation between two matrices. Input matrix I has shape 4×4 and kernel matrix W has 3×3. The colored areas in I show the receptive field for each output in O. The matrix elements are numbered by their matrix indices.

the complexity with parameter sharing by focusing on relations between spatially close data points.

Cross-correlation. The core building block of a CNN is the *Convolutional Layer* (or CONV layer). The main operation in this layer is the cross-product between tensors. We denote the cross-product between two tensors I and W as $I \star W = O$. I is the input and W is the kernel or the filter. If the two matrices are three dimensional tensors, then the third dimension of both tensors must have equal length. Let i_1, i_2 and f_1, f_2 be the width and the height of the input and the filter, respectively. The output, activation map O, has dimension $(i_1 - f_1 + 1) \times (i_2 - f_2 + 1)$. Accordingly, the cross-product for one output element is given by the following formula :

$$O_{i,j} = (I \star W)_{i,j} = \sum_{m=1}^{k_1} \sum_{n=1}^{k_2} W_{m,n} \cdot I_{i+m-1,j+n-1} + b. \tag{3.10}$$

Figure 3.4 illustrates Equation 3.10 without adding the filter-specific bias b. We fill the activation map row-wise starting from the first element in the top-left corner. To compute this element, we place the filter on top of the input so that the feature and the top-left element of the filter overlap. Then, we calculate the dot product between the elements at the same position, i.e. we obtain $O_{1,1}$ by $5 \cdot 1 + 6 \cdot 2 + 3 \cdot 8 + 4 \cdot 9 = 94$. The cross-correlation operation is equivalent to the convolution operation with horizontally and vertically flipped filter.

Concrete example. Let us continue with another example in three dimensions. Our input I is a $32 \times 32 \times 3$ tensor that represents an image with red, green and blue channels. Our filter W is a $5 \times 5 \times 3$ tensor. We have one filter channel for each color channel. Now we *convolve* this filter by sliding it across the whole image. With that, we interact the color channels dimensions because we compute the dot product over three dimensions. The result is an *activation map* with dimensions

28×28 because we can only place a 5×5 filter only 28 times over a 32×32 tensor. It is common to pad the input with a frame of zeros to control the first two dimension lengths of the output. For instance, a frame of zeros with thickness 2 maintains the first two dimension lengths of the input. Another option is to slide the filter with some stride to reduce the impact of the relations between close pixels on the filter weights. For instance, convolving the $5 \times 5 \times 3$ filter over the $32 \times 32 \times 3$ image with no padding and stride 2 results in an activation map of size $16 \times 16 \times 3$ instead. Lastly, the CONV layer does not only use a single filter but a set of filters. E.g. with a set of seven filter, we obtain the same number of 16×16 activation maps. We stack these activation maps along the third dimension of the resulting output tensor. Thus, we have transformed a $32 \times 32 \times 3$ into a $16 \times 16 \times 7$ stack of activation maps. Intuitively, each single filter has the capacity to detect specific local features in the input tensor that may be of importance to later layers. The weights in this filter tensor are parameters that we train with backpropagation.

General definition. A convolutional layer for images that are represented by a three dimensional input tensor is given by the following five components:

> **Input:** a tensor I of size $W_1 \times H_1 \times D_1$
>
> **Hyperparameters:** the number of filters K, the filter's width or height F (assuming both are equal), the stride S , and the amount of zero padding, P.
>
> **Output:** D_2 different activation maps stored in a volume of size $W_2 \times H_2 \times D_2$, where $W_2 = (W_1 - F + 2P)/S + 1$, $H_2 = (H_1 - F + 2P)S + 1$, and $D_2 = K$.
>
> **Complexity:** the number of parameters in each filter is $F \times F \times D_1$. This gives a total of $K \times (F \times F \times D_1) + K$ parameters for the whole layer. Note that the filter depth always equals the input depth. Moreover, the last K represents the bias terms that we add to the respective filter after each dot product computation with the data.
>
> **Operation:** Each d-th slice of the output tensor (of size $W_2 \times H2$) is the result of computing the cross-correlation between the d-th filter over the input tensor with a stride of S and offsetting the result by d-th bias afterwards.

Parameter sharing. Cross-correlation slides each filter over the input with the same weights at every position. As a consequence, the size of the receptive field, i.e. the set of inputs that impact one output, is much smaller compared to fully-connected layers (cf. Figure 3.3). In particular, a convolutional layer is a special case of a fully connected layer, where many neurons have the same (re-arranged) set of weights and where most weights are set to zero except of a small

neighborhood. Hence, the convolutional layer has much less parameters and is less prone to overfitting. To illustrate this point, let us consider the example of a $128 \times 128 \times 3$ input image that is taken by a convolutional layer with 32 $5 \times 5 \times 3$ filters, padding of 2 and a stride of 1. The output is a $128 \times 128 \times 32$ volume consisting of $524,288$ elements. We compute this volume with only $32 \cdot (5 \cdot 5 \cdot 3) + 32 = 2,432$ total parameters. In contrast, if this was a fully connect layer that computes every output element based on its own specific weights, we would use $524,288 \cdot (128 \cdot 128 \cdot 3) = 25,769,803,776$ parameters. This number is not only gigantic but it would also be difficult not to overfit the data even if we could store the parameters and compute the result.

Pooling layers. Another building block of CNNs are pooling layers. These layers are used to further reduce overfitting by downsampling the convolutions output with a fixed scheme and without any parameters. Specifically, these pooling operations are applied to each activation map separately and preserve the depth of the output volumes but not their height and width. As with cross-correlation, we slide the pooling filter over its input. However, we have to do this for each input channel separately because the pooling operation has no depth. A common setting is a 2×2 filter with stride 2 where the filter represents a max operation over four numbers. This filter gives us an output tensor that is downsampled by 2×2 along the first two dimensions.

Backward pass. The Jacobian of a convolution layer $O = I \star W$ is given by $I \star J^O$, where J^O is the Jacobian that contains the upstream gradients δ^o with respect to the activation map parameters. This is illustrated by Figure 3.5. In comparison to the Jacobian for a fully-connected linear layer, the smaller, shared downstream gradient is only multiplied with the activations of its adjacent elements from the previous layer. The first derivatives of the pooling operations average and max are simple. The derivative of the average with respect to one element is 1 divided by the number of elements. The derivative of the max is the indicator function of maximum element's index.

CNN architectures. We build complete CNNs by stacking convolutional and pooling layers. A classical architecture is LeNet-5 for digit classification from black & white images of hand-written digits [57]. A slightly simplified version has the form SOURCE, [[CONV, POOL] $\cdot$ 2], CONV, FC, FC, SOFTMAX. In this notation, SOURCE stands for a tensor of a batch of images ($[100 \times 32 \times 32 \times 1]$ for a batch of 100 32×32 black & white images), CONV denotes six, sixteen, and 120 5×5 filters with stride 1 and tanh activation, POOL denotes an average pooling layer with a 2×2 filter and a stride of 2, and FC represents fully-connected layers. The first layer has tanh activations and the last layer calculates the softmax probabilities for ten different digits. The FC layers are used to extract features not only locally but globally and because this type of layer is cheaper after

$$\begin{bmatrix} \delta^w_{1,1} & \delta^w_{1,2} & \delta^w_{1,3} \\ \delta^w_{2,1} & \delta^w_{2,2} & \delta^w_{2,3} \\ \delta^w_{3,1} & \delta^w_{3,2} & \delta^w_{3,3} \end{bmatrix} = \begin{bmatrix} i_{1,1} & i_{1,2} & i_{1,3} & i_{1,4} \\ i_{2,1} & i_{2,2} & i_{2,3} & i_{2,4} \\ i_{3,1} & i_{3,2} & i_{3,3} & i_{3,4} \\ i_{4,1} & i_{4,2} & i_{4,3} & i_{4,4} \end{bmatrix} \star \begin{bmatrix} \delta^o_{1,1} & \delta^o_{1,2} \\ \delta^o_{2,1} & \delta^o_{2,2} \end{bmatrix}$$

J^W I J^O

Figure 3.5: Backward pass through cross-correlation. The Jacobian J^W for the weight parameters of the cross-correlation $O = I \star W$ is given by the cross correlation between the Jacobian J^O that contains the downstream gradients δ^o with respect to the activation map parameters. The shaded area in I shows the elements that are used in the dot product with J^O to compute the shaded element in J^W.

multiple rounds of downsampling. A receptive field of a (hidden) feature is the set of inputs that influence this feature. In a fully connected layer, the receptive field of every hidden feature is always the complete input vector or tensor. By stacking multiple convolutional layers, we can achieve the same receptive field with much less parameters. The outputs from higher layers have larger receptive fields and thus represent higher-level features. One example for these type of features could be far-reaching edges.

Lengthy network paths. In the last paragraph, we learned that classic CNN architectures are essentially a stack of functions. In Section 3.3, however, we saw that a sequence of function applications results in a long and linear backpropagation graph given by a multiplication sequence of partial derivatives. If a number of these partial derivatives are either very small or very large, their multiplicative effect can cause either too small or too large gradient updates during optimization. Especially layers with sigmoid activations (e.g. logistic, tanh) with derivatives that are flat or extremely steep for large parts of the domain are problematic. If parameters have once reached these parts, learning oftentimes stops for larger chunks of the network for two reasons. First, for these parameters, it requires a number of unusually large steps to leave these extreme areas. And second, other parameters with gradients that include multiplications with the extreme gradients are set to zero or infinity, too.

Skip connections. We can alleviate the problem by connecting earlier (or bottom) layers, $h^{(i)}$, with later (or top) layers, $h^{(i+k)}$, via the duplication operation followed by the "+" operator. With that, we open up a new path past the majority of the stacked functions. We call this type of link a *skip connection*. The effect is that, in the backward pass, $l^{(i)}$ receives another downstream Jacobian $J^{(i+k)} \cdot ... \cdot J^{(o)}$

that we add to the more complex Jacobian $J^{(i+1)} \cdot J^{(i+2)} \cdot ... \cdot J^{(i+k)} \cdot ... \cdot J^{(o)}$. Intuitively, the updates from the more complex Jacobian are used to learn the difference between the bottom layer and the top layer. As a result, we can learn a simple representation of the model without exposing the gradient to further multiplicative transformations and, in addition, we can learn another representation for more complex relations between the input features. An illustrative toy model similar to Figure 3.2 is $C(\theta) = \tanh(\theta)^n$ where $n \in \mathbb{N}^+$ represents the number of subsequent tanh operations. The model with skip connection is $C_{res}(\theta) = C(\theta) + \theta$. We can observe the described technical aspects by comparing $\partial C/\partial\theta$ with $\partial C_{res}/\partial\theta$ and the respective computational graphs. An early implementation of this idea is Microsoft's ResNet [36]. This architecture uses skip connections that only skip one layer at a time. We will return to the problem of vanishing and exploding gradients from lenthy network parths in our discussion of recurrent neural networks.

Inductive bias. In the previous paragraph, we have seen how we can design neural network architectures to form sensible predictions on a domain-specific type of source data. We have also learned how to exploit the peculiar spatial relations in this data in order to save parameters and training time compared to fully-connected FNNs. The assumptions that we pose on the relations in the data in our architecture design is called the *inductive bias*. In summary, there are three inductive biases in the convolution layers of CNNs:

> **Translation invariance:** the convolution operation is translation invariant, i.e. $f(x) = f(T(x))$ with $x, a \in \mathbb{R}^n$ and $T : \mathbb{R} : x \rightarrow \mathbb{R}^n$, where f denotes the convolution and T a transformation of the input. For images, $n = 2$. The motivation is that we want to identify an object independent of changes to its position. For other transformations, such as rotations and change in color, however, we need to train on additional augmented images.
>
> **Locality of features:** the filter sizes are much smaller than the image because we assume that local relations between the pixels are more important than global relations.
>
> **Universality of feature extractors:** we can reuse the same filter for all regions of the input because we assume that the hidden features which we extract are similarly important at each position.

In order to improve our results, we soften the inductive bias regarding the locality of features with two additional layers: at the beginning of the network, we include cheap pooling layers and towards the end we add fully connected layers.

3.4.3 Recurrent Neural Networks.

Many tasks require source or target spaces that contain sequences. For instance, in translation programs we oftentimes encode words as sequences of one-hot vectors. These vectors are index vectors with a one at the position of an integer that maps to a word in a fixed vocabulary. A simple recurrent neural network (RNN) processes a sequence of vectors $\{x_1, ..., x_T\}$ with a recurrence formula $h_t = f_\theta(h_{t-1}, x_t)$. The function f that we will describe in more detail below takes the same parameters θ at every time step to process an arbitrary number of vectors (cf. parameter sharing and inductive bias). An interpretation of the hidden vector h is that of a summary of all previous x vectors. A common initialization is $h_0 = \vec{0}$. We can define f according to three criteria. The first criterion is the source and target space. We would model f differently for spaces from one to many, many to one or many to many vectors. The second criterion is the order in which we process source vectors and predict target vectors. This depends on the nature of the data. For example, we may either want to predict y_t directly after processing x_t, or predict the complete output sequence $\{y_1, ..., y_T\}$ after we have processed the complete source sequence $\{x_i\}_{i=1}^T$. Another option is to not only compute a summary of the past but also a summary of the future and use both vectors in our predictions. The third criterion is how we want to improve upon drawbacks of the model formulation for longer sequences. We will come to this point later in this section. To illustrate how RNNs work, we will look at two examples.

Vanilla Recurrent Neural Networks use a recurrence defined by

$$h_t = \phi\big(W \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix}\big). \tag{3.11}$$

Here, we concatenate the current input and the previous hidden states, transform both linearly and pass it to a non-linear activation function. This vector notation is equivalent to $h_t = \phi(W_{xh}x_t + W_{hh}h_{t-1})$. The two matrices W_{xh} and W_{hh} are concatenated horizontally to W. If the input vectors x_t have dimension $1 \times D$ and the hidden vectors dimension $1 \times H$, then $W_{xh} \in \mathbb{R}^{H \times D}$, $W_{hh} \in \mathbb{R}^{H \times H}$, and weight matrix W is a matrix with dimensions $[H \times (D + H)]$. Vanilla RNN models the current hidden states h_t at each time step as a linear function of the elements in the previous hidden states h_{t-1} and the current input x_t, transformed by a non-linearity. In a classification task, e.g. where we want to predict the next written letter in a prompt by the previously types letters, we would apply the Softmax function to a linear transformation of the hidden state at each time step, $o_t = W_{ho}h_t$, in order to predict the next character's one-hot encoding. This is illustrated by Figure 3.6.

Encoder-Decoder RNNs use the complete input history $\{x_i\}_1^{T_x}$ to predict the

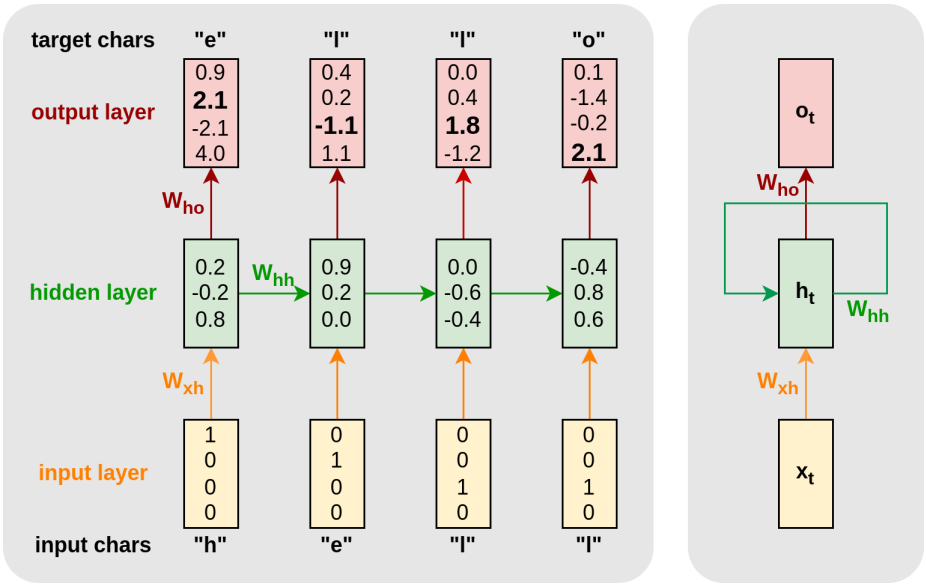

Figure 3.6: Vanilla RNN as character-level language model. The left side shows the *unrolled RNN*. The vocabulary has four characters and the training sequence is "hello". Each letter is represented by a 1-hot encoding (yellow) and the RNN predicts the encodings of the next letter (green) at each time step. The RNN has a hidden state with three dimensions (red). The output has four dimensions. The dimensions are the logits for the next character. They are the softmax of a linear transformation of the hidden states. During supervised learning, the model will be trained to increase (decrease) the logits of the correct (false) characters. The right side shows the *rolled-up RNN*. The graph has a cycle that shows that the same hidden states are shared across time and that the architecture is the same for each step.

first output y_1. Then, it additionally uses the complete prediction history $\{\hat{y}_i\}_1^{t-1}$ to predict the next y_t for $t = 2, ..., T_y$. The model is able to generate sequences of arbitrary length that can be unequal to the length of the input sequence. An examplary task is translating sentences from English to German. Here, we work with one-hot encodings of words from a fixed vocabulary instead of letters from an alphabet. To build this RNN, we use the same recursion from the previous example as an encoder RNN. However, do not classify the output at each timestep directly. Instead, we use the last hidden state from the encoder h_T as a context vector c_0. Intuitively, the context vector is an abstract representation of the entire input sentence. Then, we use another RNN, the decoder RNN to process the information from the context and the output from the previous period to generate the hidden states s_t for the current period:

$$s_t = \phi(W_{os}o_{t-1} + W_{ss}s_{t-1}) \tag{3.12}$$

with $s_0 = c_0$ and, for instance $y_0 = \vec{0}$. We then predict the next word's one-hot encoding from the hidden states like in the last example with the softmax of $o_t = W_{so}s_t$. To model the fact that input and output sequences can have different length, we require special start of sentence, <sos>, and end of sentence, <eos>, tokens.

The encoder uses the <eos> as x_1. The decoder takes <sos> as y_1 and stops the recursion when it returns <eos>.

The simplicity of the RNN's formulation has two drawbacks. First, the connections between inputs and hidden states through linear layers and element-wise non-linearities is not flexible enough for some tasks. Second, the recurrent graph structure leads to problematic dynamics during the backward pass.

Exploding and Vanishing Gradient Problem. We now explore the second problem more formally. The vanilla RNN's loss with respect to the weight matrix in Equation 3.11 is given by:

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T}\sum_{k=1}^{t+1} \frac{\partial L_{t+1}}{\partial o_{t+1}} \frac{\partial o_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_k} \frac{\partial h_k}{\partial W}. \tag{3.13}$$

The crucial part is the derivative of the hidden layer in the next period with respect to some past layer k. It is given by the recursive product

$$\frac{\partial h_{t+1}}{\partial h_k} = \prod_{j=k}^{t} \frac{\partial h_{t+1}}{\partial h_k} = \frac{\partial h_{t+1}}{\partial h_t} \frac{\partial h_t}{\partial h_{t-1}} ... \frac{\partial h_{k+1}}{\partial h_k} = \prod_{j=k}^{t} \text{diag}\big(W_{hh}\phi'(W[x_{j+1}; h_j])\big). \tag{3.14}$$

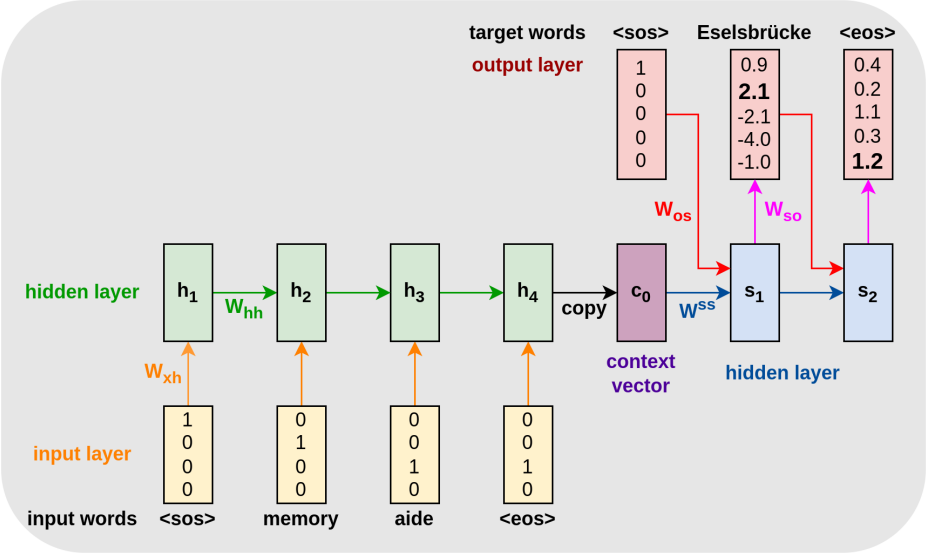

Figure 3.7: Encoder-Decoder RNN as word-level language model. The input language has two and the output language has three words. Every word, the start and the end of a sentence are represented by a one-hot encoding (yellow) and the RNN predicts the encodings of the translation of the input sentence. The encoder RNN has a hidden state (green) that is updated by one word step-by-step to produce the final context vector (purple). The decoder RNN takes linear functions of the final encoding and the start embedding to compute the hidden states for the next output embedding (blue). The one-hot encoding of the output words (red) are a softmax of a linar function of these hidden states. The decoder RNN iterates the prediction until it returns the end token. With this architecture, we can predict sequences of arbitrary length using the embedded information of a whole input sequence.

Equation 3.14 shows that, in order to compute the gradients for W_{xh} and W_{hh}, we have to multiply a large number of Jacobians depending on the size of the input sequences. The reason is that the derivative of a hidden layer h_{t+1} with respect to some previous layer h_k equals the product of the derivatives from $t+1$ to $k+1$ with respect to their previous layers. For simplicity, let us assume that these derivatives are constant. Further, let us compute the eigendecomposition of the fixed Jacobian matrix $\frac{\partial h_{t+1}}{\partial h_t}$ to analyze the problem more formally. We obtain the eigenvalues $\lambda_1, \lambda_2, ..., \lambda_n$, with $|\lambda_1| > |\lambda_2| > ... > |\lambda_n|$, and the respective eigenvectors, $v_1, v_2, ..., v_n$. With these components, we can write the constant update of the hidden state in the direction of an eigenvector v_i as $\lambda_i \Delta$. During the backward pass from $t+1$ to k, this change becomes $\lambda_i^{t-k} \Delta h$. As a result, if the largest eigenvalue λ_1 is smaller than one, the gradient will vanish, and if it is larger than one, the gradient explodes. In practice, the gradients oftentimes rather vanishes due the contribution of the activation derivative which cannot exceed one. In this case, the earlier hidden states are barely updated. The effect is that the front parts of the input do not impact the prediction. In other words, our model has a weak long-term memory.

There are many approaches to alleviate this problem. Examples are regularization, careful weight initializations, using ReLU activations only, and gated recurrent networks like Long Sort-Term Memory (LSTM) or Gated Recurrent Unit (GRU). We can view these gating mechanisms as much more sophisticated extensions of the skip connections from subsection 3.4.2. Today, the most common approach is the transformer architecture. It is not only less vulnerable to vanishing gradients but also provides a more expressive coupling of current inputs and previous states compared to vanilla RNNs.

3.4.4 Transformer

In the last section we learned that RNNs have problems with learning relations between the first parts of a source sequence with the target and later parts of the source sequence. An architecture without this structural problem is the transformer. The architecture was published by Vaswani et al. [91] and applied to machine translation. We will develop this model step-by-step, starting with its core component, attention.

Attention. Let us introduce this concept with an example. Figure 3.8 shows an encoder-decoder network with attention similar to the previous encoder-decoder RNN (Figure 3.7). The core idea of attention is defining the hidden state of the decoder-RNN as a function of every hidden state from the encoder-RNN for every time period without recursion. The result of the attention function is the context vector. We will use this vector for every output element. In the specific network

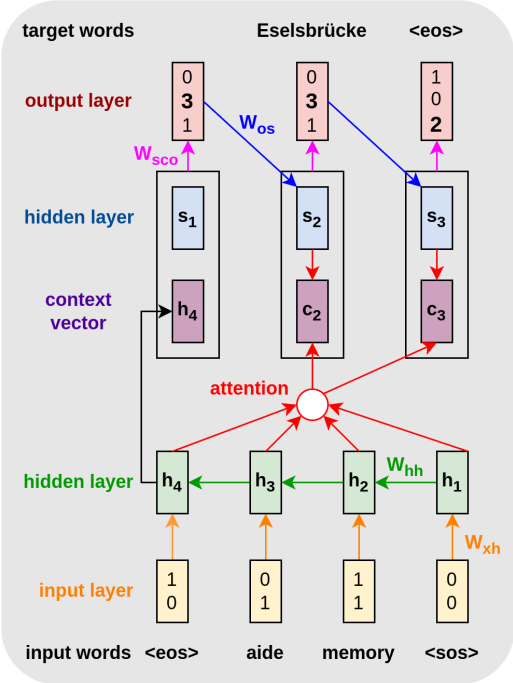

Figure 3.8: Encoder-Decoder with attention. In contrast to the encoder-decoder RNN, the output layer is a function of the concatenation of the hidden states and a time-dependent context vector (black boxes). The main idea is that the context vector c_t is a function of all hidden states $\{h\}_{t=1}^{T}$ instead of only the last one (and the previous state s_{t-1}). This function is called *attention* (red). The black box represents vector concatenation. c_1 is initialized with h_4, s_1 with arbitrary values, and o_1 is discarded.

in Figure 3.8, the context vector is a function of both the decoder states s and the encoder states h. Further, it is additionally concatenated with s to predict the output layer.

The attention function that returns the context vector for output y_t with attention scores for each input is given by the following expression:

$$c_t = \sum_{i=1}^{n} \alpha_{t,i} h_i \tag{3.15}$$

$\alpha_{t,i}$ is a softmax function of another function *score* that measure how well output y_t and input x_i are aligned through the encoder state h_i:

$$\alpha_{t,i} = \text{align}(y_t, x_i) = \frac{\exp \text{score}(s_{t-1}, h_i)}{\sum_{j=1}^{n} \exp \text{score}(s_{t-1}, h_j)}. \tag{3.16}$$

There are many different scoring functions [96]. A common choice is the scaled dot-product score $(s_t, h_i) = \frac{s_t^T h_i}{\sqrt{d}}$, where d is the hidden state dimension of both encoder and decoder states. Here, the alignment score for one sequence element is given by a relative score of the dot-product between the respective encoder

hidden state and the current decoder hidden state. We scale down the dot product to prevent vanishing gradients from a pass to a softmax layer. After training the model, we can analyze how much each output element depends on, or *attends* to, each input. We do this by assembling a table with outputs as columns and the output-specific alignment scores for each input as rows.

Another option for an encoder-decoder with attention is using a self-attention mechanism to compute the context vector, for example, with score $(h_j, h_i) = \frac{h_j^T h_i}{\sqrt{d}}$. We can use the scores, for instance, in machine translation, to model how important the previous words are for translating the current word in a sentence. Note that we can execute many of these operations in parallel for the whole input and output sequences using matrix operations. To distinguish attention between outputs and inputs from self-attention, we also introduce the term *cross-attention*.

Next, we will discuss an expansion of the attention mechanism and how it is applied to the transformer's encoder and decoder, separately. Finally, we will look at the complete model, and how it replaces the positional information from the encoder RNN in a simple way without any recurrence.

Key, value and query. As we do not use recurrence of single sequence elements anymore, let us denote the whole sequence of input embeddings by $X \in \mathbb{R}^{L \times D^{(x)}}$. L can either be the complete input length T_x or later only a fraction of it. $D^{(x)}$ is the input embedding's length. Let us denoted the sequence of output embeddings by $Y \in \mathbb{R}^{M \times D^{(y)}}$. The transformer uses an extension of the attention mechanism, the multi-head attention, as its core building block. The first step is that, instead of using the softmax of the scaled dot-product between encoder states h and decoder states s directly as in the last section, it uses the scaled dot-product with two different input encodings, $K = XW^k \in \mathbb{R}^{L \times D_k}$ and $V = XW^v \in \mathbb{R}^{L \times D_v}$, and an output encoding $Q = YW^q \in \mathbb{R}^{M \times D_k}$, with $W^k \in \mathbb{R}^{D^{(x)} \times D_k}$, $W^q \in \mathbb{R}^{D^{(y)} \times D_k}$ and $W^v \in \mathbb{R}^{D^{(x)} \times D_v}$. Note that source and target embeddings are projected linearly into the same space. We compute attention with

$$c(Q, K, V) = \text{Softmax}\big(\frac{QK^T}{\sqrt{n}}\big)V. \tag{3.17}$$

We call (K,V) key-value pairs and Q the query. Using the interpretation of the dot product as a similarity measure, the context matrix shows the (self-)similarity between the input and a representation of the input that is weighted by its similarity to the output (so far). $c(Q, K, V)$ is a matrix because we now compute the attention scores for every target (query) dimension at once. However, we mask embeddings for unseen target elements in every period. In the transformer encoder, there is an important layer where the queries are also source representations and in the

decoder, there is a layer where keys and values are also target representation.

Multi-head attention. Instead of computing the attention once, the multi-head approach splits the three input matrices into smaller parts and then computes the scaled dot-product attention for each part in parallel. The independent attention outputs are then concatenated and linearly transformed into the next layer's input dimension. This allows us to learn from different representations of the current information simultaneously with high efficiency.

$$\text{MultiHead}(X_q, X_k, X_v) = [\text{head}_1; ...; \text{head}_h]W^o, \tag{3.18}$$

where head_i =Attention$(X_qW_i^q, X_kW_i^k, X_vW_i^v)$ and $W_i^q \in \mathbb{R}^{D^{(y)}\times D_v/H}, W_i^k \in \mathbb{R}^{D^{(x)}\times D_k/H}, W_i^v \in \mathbb{R}^{D^{(x)}\times D_v/H}$ are matrices to map input embeddings of chunk size $L \times D$ into query, key and value matrices. $W^o \in \mathbb{R}^{D_v\times D}$ is the linear transformation in the output dimensions. These four weight matrices are learned during training. Target self-attention and cross attention layers compute outputs in $\mathbb{R}^{M\times D}$, and source self-attention calculates outputs in $\mathbb{R}^{L\times D}$.

Transformer encoder. Figure 3.9 depicts the encoder network. It computes an input representation based on the self-attention mechanism that allows it to locate particular pieces of information from a large context at all positions.

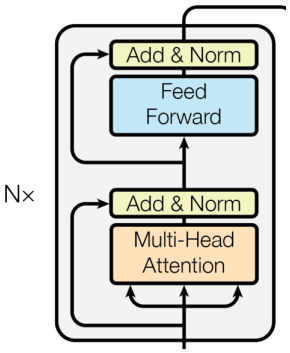

Figure 3.9: Transformer encoder. In the original form, the encoder is a stack of $N = 6$ identical layers but with different parameters. It consists of two similar components. The first sublayer starts with a multi-head *self*-attention layer (orange) and the second with a *point-wise* fully-connected feed forward network (blue). Point-wise means that the same weights are applied to each input element. Afterwards, the respective previous input vector is added to both outputs and the results are normalized by the normalized residual layers (yellow). Crucially, in the self-attention layer, the queries are also functions of the input embeddings. Image source: Vaswani et al. [91].

Transformer decoder. Figure 3.10 shows the decoder network. It is able to retrieve relevant information from the encoded source representation to compute feature representations for generating the new target sequence in autoregressive

fashion. The key component is the multi-head *cross*-attention layer (in contrast to the other multi-head *self*-attention blocks).

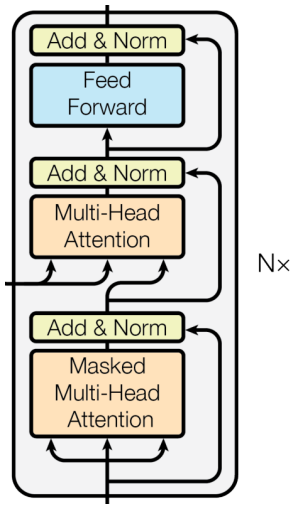

Figure 3.10: Transformer decoder. In the original form, the encoder is a stack of $N = 6$ identical layers. It first encodes the output sequence in the masked multi-head *self*-attention layer (orange). The masked elements are the target representations that are not generated so far. Next, it passes these target representations as queries to the multi-head attention layer (orange) together with the output input representations as keys and values. Finally, the results pass through a fully-connected feed forward network (blue). Each of these three layers is subsequently transformed by a normalized residual layer (yellow). Image source: Vaswani et al. [91].

The complete transformer architecture. Figure 3.10 shows the complete architecture. It has the following properties:

> **Inductive bias for self-similarity:** The self-attention layers allow the model to detect reoccuring themes independent of their distances from each other. This works well in many applications becaue reoccurence is an important pattern in numerous real-world domains.
>
> **Expressive forward pass:** The transformer interacts all elements of the input and output with themselves and each other in relatively simple and direct connections. This allows the model to learn many input-output relations in just a few steps.
>
> **Wide and shallow compute graph:** Due to the residual layers and the matrix products in the attention layers the compute graph is wide and shallow. This makes forward and backward passes fast on parallel hardware. Furthermore, together with layer normalization and dot product scaling, this mitigates vanishing or exploding gradients in backpropagation.

Complexity comparison. Let us conclude this chapter by comparing the complexities of the three main architectures in deep learning. **??** shows the differences in the number of FLOPS (floating point operations) for the main units in the tree main architectures that we have discussed so far. We observe that attention scales

Layer	Number of floating point operations
Self-Attention	$\mathcal{O}(\text{length}^2 \cdot \text{dim})$
RNN (LSTM)	$\mathcal{O}(\text{length} \cdot \text{dim}^2)$
Convolution	$\mathcal{O}(\text{length} \cdot \text{dim}^2 \cdot \text{kernel\_width})$
Fully-connected	$\mathcal{O}(\text{length}^2 \cdot \text{dim}^2)$

Table 3.2: Comparison of computation complexity between models. length refers to the number of elements in the source sequence and dim to the embedding depth for each element.

better than the other units if the length of the source sequence is much smaller than the depth of each elements embeddings. This applies to tasks like machine translation in [91] or question-answering. However, for direct applications to image data, this property does not hold. For instance, the length from a CIFAR image equals $32 \cdot 32 \cdot 3 = 3072$. Therefore, applying a transformer to image data in order to profit from its advantages requires architectures that reduce the input length beforehand.

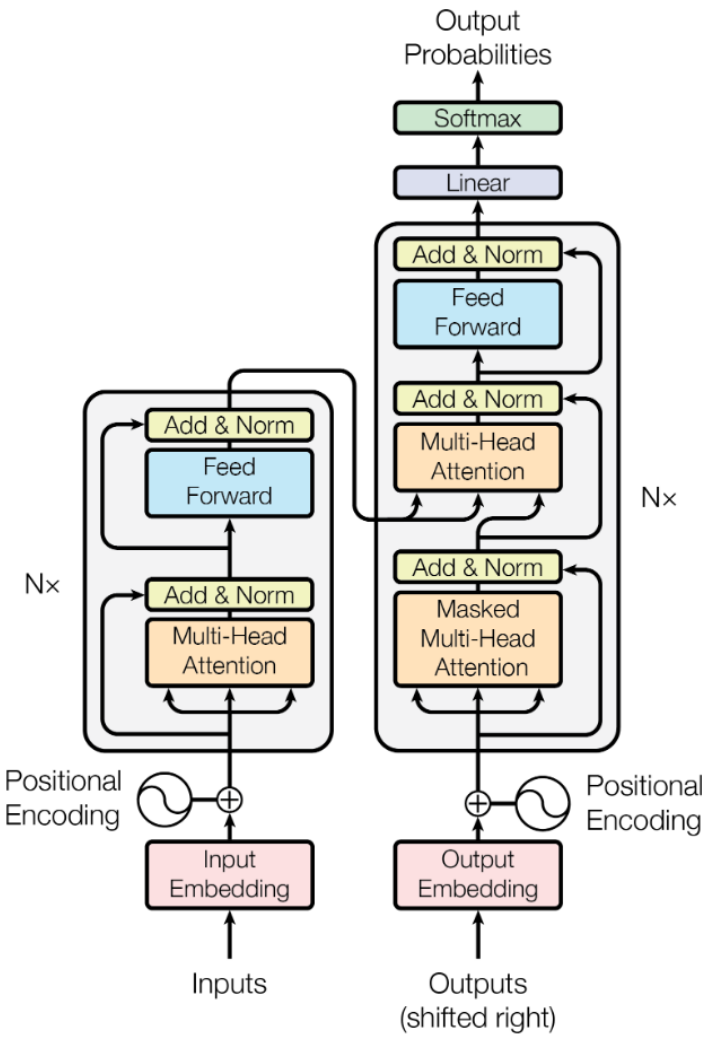

Figure 3.11: The complete transformer. In the original form, both the source and target sequence are passed to embedding layers to produce a vector of length $D = 512$ for every element. To preserve the ordering information of the inputs, we add a respective sinusoidal positional encoding vector to every embedding. To compute the probabilities for each element in the output space at every position, we pass the decoder output through a linear and a softmax layer. Image source: Vaswani et al. [91].

Chapter 4

Background: 2D-3D Projections

This chapter provides a short introduction to conversion from 2D camera coordinates to 3D world coordinates and back given camera calibration parameters based on the pinhole camera model as discussed in Sagerer, Savas, and Schmidt [83] and Hartley and Zisserman [35]. We will later use these transformations for predicting 3D cylinders instead of 2D bounding boxes to prevent occlusions using the information from our multi-camera tracking dataset.

4.1 Fundamentals: Homogenous Coordinates

In computer vision, homogeneous coordinates are used to represent points in a projective space, extending the traditional Cartesian coordinate system. A point in 2D homogeneous coordinates is represented as a 3D vector $\tilde{x} = (\tilde{x_1}, \tilde{x_2}, w)$, where w is a nonzero scalar, known as the *homogeneous coordinate*. The actual 2D Cartesian point (x_1, x_2) can be recovered from the homogeneous coordinates by dividing $\tilde{x}_1$ and $\tilde{x}_2$ by w:

$$x_1 = \frac{\tilde{x}_1}{w}, \quad x_2 = \frac{\tilde{x}_2}{w} \tag{4.1}$$

Homogeneous coordinates extend the representation to include a third coordinate (usually denoted as w), which enables the representation of points at infinity. By using these extended coordinates, parallel lines can be represented as intersecting at a point at infinity, which simplifies various geometric calculations and transformations. In perspective projection, for example, this allows for a more accurate representation of how parallel lines converge at a single point on the image plane, which is a characteristic of three-dimensional scenes projected onto a two-dimensional surface.

When performing geometric transformations, such as rotations and translations, homogeneous coordinates facilitate the use of matrix multiplication, making the calculations more efficient and elegant. The use of homogeneous coordinates is particularly important in camera transformations, where it simplifies the representation of 3D points and camera projection matrices.

The homogeneous representation of a 3D point $X = (X_1, X_2, X_3)$ can be obtained by appending a homogeneous coordinate of 1 to the 3D point:

$$\tilde{X} = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ 1 \end{bmatrix} \tag{4.2}$$

For 2D points in homogeneous coordinates, w is typically also set to 1, resulting in a 3D vector of the form $\tilde{x} = (x_1, x_2, 1)$.

The use of homogeneous coordinates allows us to represent both points and transformations in a unified manner, making it an essential concept in computer vision and graphics applications.

4.2 The Pinhole Camera Model

In the pinhole camera model, we have several components that define the mapping between 3D points in the world and their corresponding 2D points in the image plane as shown by Figure 4.1. These components include the extrinsic parameters and the intrinsic parameters.

Intrinsic Parameters. The intrinsic parameters are specific to the camera itself and define its internal characteristics. They describe how the camera captures light and projects the scene onto the image plane These parameters are constant for a given camera and remain the same regardless of the camera's position and orientation in the world. Here, we consider use case without the need for distortion and skew parameters. The intrinsic parameters include:

a. Focal Length (f_x and f_y): The focal length determines the magnification of the camera and how much the scene is "zoomed in" or "zoomed out" in the image. It is usually represented in pixels and can be different for the horizontal (f_x) and vertical (f_y) axes.

b. Principal Point (c_x and c_y): The principal point is the image coordinates of the camera's optical center. It defines the location of the camera's focal point in the image plane. The principal point is typically denoted as (c_x, c_y),

where c_x is the horizontal coordinate, and c_y is the vertical coordinate, both measured in pixels.

These intrinsic parameters are essential for determining how the camera projects 3D points onto the 2D image plane and for correcting any distortions introduced by the camera lens. This transformation is given by camera matrix K:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}.$$

Extrinsic Parameters.: The extrinsic parameters describe the position and orientation of the camera in the 3D world coordinate system. They represent the camera's external relationship with respect to the 3D scene being captured. The extrinsic parameters include:

a. Rotation Matrix or Rotation Vector: The rotation matrix R represents the orientation of the camera in 3D space. It describes how the camera's coordinate system is aligned with the world coordinate system. Alternatively, the rotation can be represented using a rotation vector r, which is a compact 3D vector that encodes the rotation axis and angle. We can convert r to R using Rodrigues formula [26].

b. Translation Vector: The translation vector t represents the position of the camera center (also called the camera origin) in the world coordinate system. It describes the 3D displacement of the camera with respect to a reference point.

Combining the intrinsic and extrinsic parameters allows us to transform 3D points in world coordinates to their corresponding 2D points in the image plane. This process is commonly referred to as the camera projection equation and involves applying the intrinsic parameters to the 3D points, followed by a transformation using the extrinsic parameters. We can also use the parameters to transform 2D points back to 3D.

4.3 Projecting a 3D Point onto the Camera Image Plane

Now, let us understand the steps involved in projecting a 3D point onto the camera image plane:

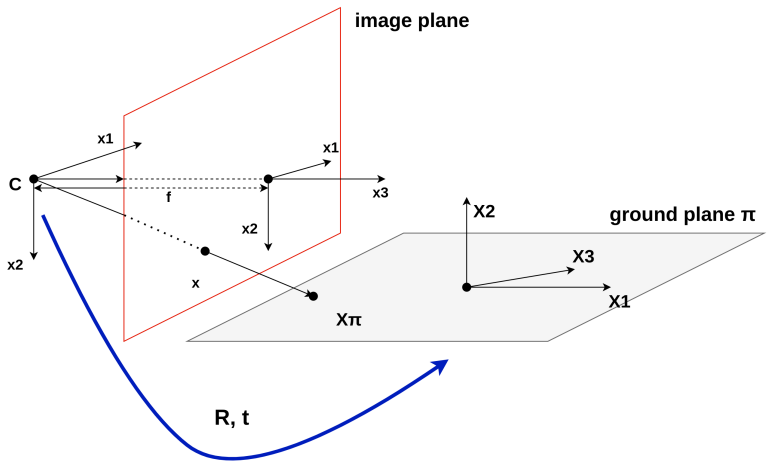

Figure 4.1: Pinhole camera model. The model describes the relationship between the 3D world (above grey area) and the 2D image (red area) captured by a camera. In this model, the world coordinates $(X1, X2, X3)$ represent points in the 3D space around the camera. The image coordinates $(x1, x2)$ correspond to the 2D points on the camera's image sensor. The camera center C is the point where all lines of sight converge within the camera, while the focal length f determines the distance between the camera center and the image plane.

Extrinsic parameters R and t involve the camera's position and orientation in the world, while intrinsic parameters encompass properties like focal length and distortion coefficients. The transformation from the 2D image plane to world space involves a process of back-projection, where rays from each image point are cast through the camera center (using the intrinsic paramters) onto the scene (with the extrinsic paramters, blue arrow). This mapping enables the conversion of 2D image features back to their 3D positions in the real world. The projection simplifies if we project 2D image points to the ground plane π. Image adapted from Hartley and Zisserman [35].

1. **Extrinsic Matrix**: Concatenate the rotation matrix R and translation vector t to form the $3 \cdot 4$ extrinsic matrix $[R|t]$:

$$[R|t] = \begin{bmatrix} R & t \end{bmatrix}$$

2. **Projection Matrix**: Compute the projection matrix P by multiplying the camera matrix K with the extrinsic matrix $[\mathbf{R}|\mathbf{t}]$:

$$P = K \cdot [R|t]$$

When we multiply the camera matrix K by the extrinsic matrix $[R|t]$, we are effectively combining the intrinsic and extrinsic properties of the camera into a single transformation. This combined transformation represents the complete process of projecting 3D points onto the 2D image plane, accounting for both the camera's internal properties (intrinsic) and its position and orientation in the world (extrinsic).

3. **Projection**: Project the 3D point onto the camera image plane by multiplying the homogeneous point $\tilde{X}$ with the transpose of the projection matrix P:

$$\tilde{x} = \tilde{X} \cdot P^T$$

4. **Dehomogenization**: Normalize the projected point to obtain the 2D coordinates $\mathbf{x}$ by dividing the first two elements of $\tilde{x}$ by the third element:

$$x = \frac{\tilde{x}_{\{1,2\}}}{\tilde{x}_{\{3\}}}$$

4.4 Projecting a 2D Point onto the 3D World Ground Plane

1. **Projection Matrix**: We combine the camera matrix K with the rotation matrix R to form the projection matrix M:

$$M = K \cdot R$$

The projection matrix represents the complete transformation from the 3D world to the 2D image plane.

2. **Camera Center in World Coordinates**: To find the camera center's 3D coordinates in the world coordinate system, we calculate $\tilde{C}$ by transforming the negative of the camera translation vector t_{vec} using the transpose of the rotation matrix R:

$$\tilde{C} = -R^T \cdot t_{\text{vec}}$$

3. **Projection to 3D World Ground Plane**: We project the 2D point $\tilde{x}$ back to 3D world coordinates using the inverse of the projection matrix M:

$$\tilde{X} = M^{-1} \cdot \tilde{x}$$

4. **Normalization**: To obtain the 3D coordinates on the ground plane, we normalize $\tilde{X}$ by adjusting its depth based on the camera center's distance from the 3D point:

$$\tilde{X}^{\pi} = -\frac{\tilde{C}_{\{3\}}}{\tilde{X}_{\{3\}}} \cdot \tilde{X} + \tilde{C}$$

where $\tilde{C}_{\{3\}}$ is the third element of $\tilde{C}$, and $\tilde{X}_{\{3\}}$ is the third element of $\tilde{X}$.

5. **Dehomogenization**: The resulting $\tilde{X}^{\pi}$ represents the 3D projection of the image point on the ground plane. To obtain the final 3D world ground plane coordinates $X = (X_1, X_2, X_3)$, we divide the first three elements of $\tilde{X}^{\pi}$ by its fourth element:

$$X = \frac{\tilde{X}^{\pi}_{\{1,2,3\}}}{\tilde{X}^{\pi}_{\{4\}}}.$$

Chapter 5

Model

In this chapter, we present a comprehensive multi-camera multi-object tracking model that comprises three key components. The first component utilizes the powerful object detector DETR (Carion et al. [12]) to identify diverse objects within an image. Chapter 3 provides the necessary background knowledge about deep learning, with a particular focus on Convolutional Neural Networks (CNNs) and transformers.

The second component introduces the tracking model Trackformer (Meinhardt et al. [64]), which leverages the object detector's output embeddings and assigns consistent identities to objects across consecutive frames. By building upon the object detector's capabilities, Trackformer achieves reliable multi-object tracking.

For the third component, we propose Trackformer's extension to handle multiple camera recordings with overlapping views of the same scene, aiming to enhance object tracking performance, especially amidst occlusions. We extend the tracker's functionality to incorporate multiple video inputs, each capturing various perspectives of the same scene. Unlike the first two sections about the detection and the tracking model, where we primarily focus on 2D detections, encompassing bounding boxes and object classes, in this section, we shift our attention to 3D detections represented as cylinders. To this end, we harness our understanding of 2D-3D transformations that we gained in Chapter 4.

It is important to highlight that our approach is tailored explicitly for the multi-camera tracking scenario. Consequently, we have fine-tuned and adapted certain aspects of the original model descriptions from the first two sections to align with the multi-camera setting. At the end of this chapter, we conduct a thorough comparison between our proposed multi-camera tracker and Trackformer (Meinhardt et al. [64]), focusing on architectural differences and distinct training processes.

5.1 DETR as Object Detector

Previous object detection systems incorporate various manually designed elements, such as anchor generation, rule-based training target assignment, and non-maximum suppression (NMS) post-processing [61]. These components do not constitute fully end-to-end solutions. In a recent study, Carion et al. [12] introduced DETR, an object detection approach that eliminates the need for such manual components. DETR represents the first fully end-to-end object detector and achieved highly competitive results on the COCO 2017 detection dataset [60]. The two main components are a set-based global loss that enforces a subset of unique class predictions via bipartite matching, and a transformer encoder-decoder architecture with learned embeddings called *object queries*. In contrast to Vaswani et al. [91], these queries are learned encodings of potential objects on the image instead of rule-based encoded representations of the target sequence. This section explains first the transformer at inference time, and second the bipartite matching loss with which we train the model.

Complete transformer-based DETR architecture. The overall model has three components as depicted by Figure 5.1. The first component is a conventional CNN backbone that generates a lower-resolution activation map many channels. The transformation from large embedding length (image height $\cdot$ width) to large embedding depth is crucial for the efficiency of the attention modules (see Figure 3.4.4). The second component is the transformer encoder-decoder. In contrast to the original decoder by Vaswani et al. [91], DETR decodes N objects at once at each decoder layer instead of predicting the output sequence in an autoregressive manner by masking later elements. Because the decoder is permutation-invariant like the encoder, we also require N different decoder input embeddings to generate different results. We achieve this by feeding learned embeddings with dimension d that we call *object queries* to the decoder. In contrast to that, the original transformer in Vaswani et al. [91] takes the decoder outputs from the previous iteration as decoder inputs. The decoder transforms the N object queries into N output embeddings. The third component is a two-headed prediction network that is shared for all output embeddings. It is defined as an FFN, i.e. a a 3-layer vanilla neural net with ReLU activations for the normalized bounding box values and hidden dimension d. We normalize the input to the prediction FFNs with a shared layer-norm. The one head is a linear projection layer to predict the bounding box values. The other head is a softmax layer that predicts the class labels.

The transformer allows the model to use self- and cross-attention over the object queries to include the information about all potential objects with pair-wise relations in their prediction of only one potential object, while using the whole image as context.

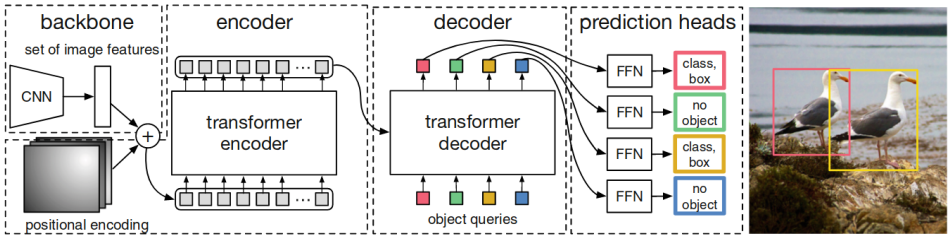

Figure 5.1: High-level DETR architecture with CNN backbone, transformer-encoder decoder and FFN prediction heads. In contrast to the autoregressive sequence generation in Vaswani et al. [91], the outputs are computed at once by feeding different learned positional encodings, called *object queries*, into the transformer decoder. Image source: Carion et al. [12].

Object detection set prediction loss. During training, we want to learn predictions that have the right number of no classes and the right number of classes with the right class label and bounding box. Formally, we achieve this the following way: From the transformer decoder prediction head, DETR infers N predictions for the tuples of bounding box coordinates and the object class. N has to be at least as large as the maximal number of objects on one image. The first tuple element are the bounding box coordinates, denoted by $b \in [0, 1]^4$. These are four values: the image-size normalized center coordinates, and the normalized height and width of the box w.r.t. the input image's borders. Using center coordinates and normalization help us to deal with images of different sizes. Examples for the second tuple element, class c, are "person A", for person detection, or "car" for object detection. The remaining class is the "no object" class, denoted by $\emptyset$. For every image, we pad the target class vector c of length N with $\emptyset$ if the image contains less than N objects. To score predicted tuples $\hat{y} = \{\hat{y}_i\}_{i=1}^{N} = \{(\hat{c}_i, \hat{b}_i)\}_{i=1}^{N}$ with respect to the targets y, Carion et al. [12] apply a loss function that we apply to a class permutation based on an *optimal* bipartite matching between the predicted and target tuples. With this loss function, we jointly maximize the log likelihood of the class permutation and minimize the bounding box losses. Figure 5.2 depicts an example of bipartite matching between predictions and ground truth for a picture of two seagulls at a shore during training time. We observe that the matching procedure selects a unique permutation that directly maps exactly one prediction to one target. Thus, with that approach we do not have to handle mappings of many similar predicted bounding boxes and classes to one target, for instance, to the same seagull.

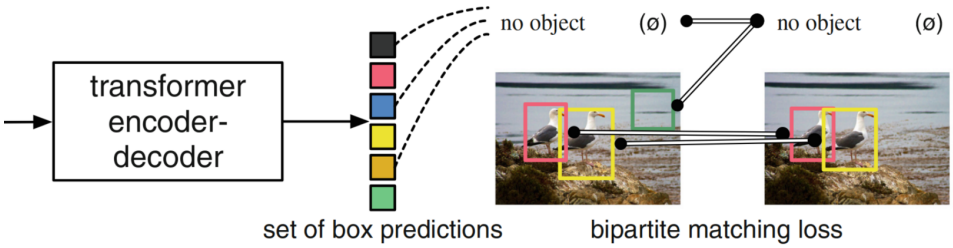

Figure 5.2: DETR training pipeline with bipartite matching loss. The loss function generates an optimal one-to-one mapping between predictions and targets according to bounding box and object class similarity (colors). In this unique permutation of N classes, class predictions with no match among the class targets are regarded as "no object" predictions, too (green). Image source: Carion et al. [12] .

The next section describes the loss function required for the bipartite matching between predicted and target detections that is optimal in terms of bounding box and class similarity. We find the optimal permutation of predicted detections $\hat{\sigma} \in \mathfrak{S}_N$ of N elements from Equation 5.1:

$$\hat{\sigma} = \arg\min_{\sigma \in \mathfrak{S}_N} \sum_{i}^{N} \mathrm{L}_{\text{match}}(y_i, \hat{y}_{\sigma(i)}), \tag{5.1}$$

where $\mathrm{L}_{\text{match}}$ is a pair-wise matching cost between target y_i and prediction $\hat{y}$ with index $\sigma(i)$. We compute the assignment efficiently with the Hungarian matching algorithm [50] instead of brute force.

We want to assign predictions and targets that are close in terms of class and bounding box. Thus, the matching considers both the class score for every target and the similarity of predicted and target box coordinates. We reward a high class score for the target class and a small bounding box discrepancy. To this end, let us denote the probability of predicting class c_i for detection with permutation index $\sigma(i)$ by $\hat{p}_{\sigma(i)}(c_i)$ and the predicted box by $\hat{b}_{\sigma(i)}$. With that, we define the pair-wise matching loss as

$$\mathrm{L}_{\text{match}}(y_i, \hat{y}_{\sigma(i)}) = -\mathbb{1}_{\{c_i \neq \emptyset\}} \hat{p}_{\sigma(i)}(c_i) + \mathbb{1}_{\{c_i \neq \emptyset\}} \mathrm{L}_{\text{box}}(b_i, \hat{b}_{\sigma(i)}). \tag{5.2}$$

Here, our objective is to find the best matching for the "real" classes $c \neq \emptyset$ by ignoring class predictions that are directly mapped to "no class" by the prediction. However, the model still has to learn not to predict too many real classes. Given our optimal matching $\hat{\sigma}$, we achieve this by minimizing the *Hungarian loss*. The

function is given by

$$\mathrm{L}_{\text{Hungarian}}(y, \hat{y}) = \sum_{i=1}^{N} \left[-\log \hat{p}_{\hat{\sigma}(i)}(c_i) + \mathbb{1}_{\{c_i \neq \emptyset\}} \mathrm{L}_{\text{box}}\left(b_i, \hat{b}_{\hat{\sigma}}(i)\right) \right] \tag{5.3}$$

The difference to the matching loss is two-fold. First, we now penalize wrong assignments to all classes including "no class" with the first class-specific term. We do not include the "no class" instances to the box loss L_{box} because they are not matched to a bounding box anyway. Second, we scale the class importance compared to the bounding boxes by taking the log of predicted class probabilities. In practice, the class term is further reduced for "no class" objects to take class imbalance into account. The last expression is the box loss. It is the L1 loss of the bounding box vector b_i. Note, however, that the L1 loss penalizes larger boxes. Equation 5.4 shows the expression:

$$\mathrm{L}_{\text{box}}(b_i, \hat{b}_{\sigma(i)}) = \lambda_{\text{box}} ||b_i - \hat{b}_{\sigma(i)}||_1, \tag{5.4}$$

where $\lambda_{\text{box}} \in \mathbb{R}$ is a hyperparameter. We normalize the loss by the number of objects in each image.

DETR's drawbacks. In spite of its intriguing design and commendable performance, DETR encounters certain challenges. Firstly, it requires significantly more training epochs to reach convergence compared to existing object detectors. For instance, when evaluated on the COCO benchmark [60], DETR necessitates 500 epochs for convergence, making it approximately 10 to 20 times slower than Faster R-CNN [77]. Secondly, DETR exhibits relatively lower proficiency in detecting small objects (Zhu et al. [109]). Contemporary object detectors typically utilize multi-scale features, employing high-resolution feature maps for the detection of small objects. However, employing high-resolution feature maps leads to impractical complexities for DETR. These aforementioned issues primarily stem from the deficiency of Transformer components in processing image feature maps. During initialization, the attention modules distribute nearly uniform attention weights to all pixels in the feature maps. It requires many training epochs for the attention weights to learn to concentrate on sparse meaningful locations. Additionally, the computation of attention weights in the Transformer encoder is quadratic in relation to the number of pixels. Consequently, processing high-resolution feature maps becomes highly computationally and memory intensive.

5.2 Trackformer as Multi-object Tracker

Trackformer (Meinhardt et al. [64]) extends DETR to multi-object tracking. To be precise, it uses the variant Deformable DETR ([109]), presumably because the results were better. Trackformer not only achieved state of the art results for online tracking but also presented an end-to-end architecture that solves the three subtasks of track initialization (detection), prediction of next positions, and matching predictions with detections. Thereby, it bypasses intermediate layers that are usually present in previous pipeline designs, similar to how DETR facilitated object detection. The main idea is depicted by Figure 5.3. It is to re-use DETR's *decoded* object queries that have been matched to an actual object in one frame and use them as additional object queries for the next frame as *autoregressive track queries*. Accordingly, we dynamically adjust the transformer decoder sequence length. The static object queries are responsible for initializing new tracks and the taken over track queries allow tracking objects across frames. In contrast do DETR, besides the bounding box quadruple b and the object class c, we additionally predict predict the track ID across frames in an implicit way from by enumerating the track queries.

With the described approach, we train the model to not only decode learned object queries into representations that can detect objects but also to use the decoded queries again as decoder input to detect the same object when possible. If we match the decoder output from an object query in frame $t-1$, it is re-used as an additional track query as decoder input for frame t. If its output is matched again, we assume that both detections belong to the same object with ID k.

Set prediction loss. We now want to formulate a loss that allows the model to learn the bipartite matching $j = \pi(i)$ between target objects y_i to the set of both object and track query predictions $\hat{y}_j$. For this purpose, let us denote the subset of target track identities at frame t with $K_t \subset K$. This is different from DETR as K contains all object identities for all images in the video sequence. These object or track identities can be present in multiple frames, i.e. they can intersect from frame to frame. Trackformer takes three steps to associate queries with targets. The last step corresponds to DETR's method. The steps to obtain the mapping of predicted detections to target detections $\hat{\sigma}$ for one frame are the following: first, we match $K_{\text{track}} = K_{t-1} \cap K_t$ (target objects in the current frame that were also present in the previous frame) by track identity. This means, we associate these targets with the output from the previous query. Second, we match $K_{\text{leaving}} = K_{t-1} \setminus K_t$ (objects leaving the scene between two frames) with background class $\emptyset$. And third, we match $K_{\text{init}} = K_t \setminus K_{t-1}$ (objects entering the scene) with the N_{object} object queries by minimum cost mapping based on object class and bounding box similarity the same ways as DETR assigned its targets to object queries.

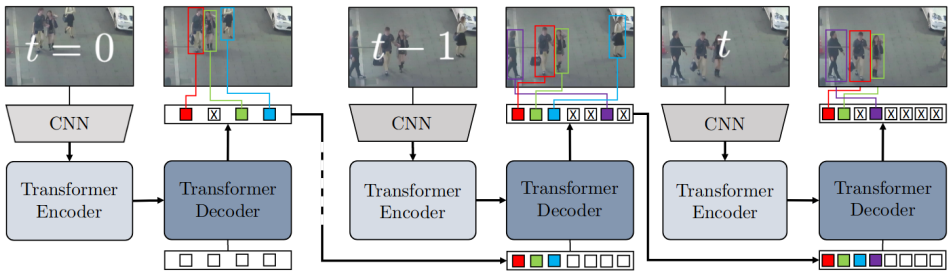

Figure 5.3: Trackformer architecture. Trackformer extends DETR to tracking on video data by feeding the decoded object queries that are matched to actual objects as additional *autoregressive tracking queries* (colored detection squares) next to the object queries for the next image into the transformer decoder (dark blue). The decoder processes the set of $N_{\text{track}} + N_{\text{object}}$ queries to further track or remove existing tracks (light blue) and to initialize new tracks (purple). Image source: Meinhardt et al. [64] .

Output embeddings which were not matched, i.e. 1) proposals with worse class and bounding box similarity than others or 2) track queries without corresponding ground truth object, are assigned to background class $\emptyset$.

With this order, we prioritize matching track queries from the last frame even if object queries from the current frame yield more similar bounding boxes and classes. This is necessary because, in order to assign the same object ID to detections in multiple frames, we have to train the object queries to not only initialize a track after on pass through the decoder but also to detect the same object after two passes through the decoder with given the respective interactions from the image encodings.

The final set prediction loss for one frame is computed over all $N = N_{\text{object}} + N_{\text{track}}$ model outputs. Because $K_{t-1} \setminus K_t$ (objects that left the scene) are not contained in the current-frame permutation, we write the loss as

$$\mathrm{L}_{\text{MOT}}(y, \hat{y}, \pi) = \sum_{i=1}^{N} \mathrm{L}_{\text{query}}(y, \hat{y}, \pi). \tag{5.5}$$

Further, we define the loss per query and differentiate two categories. First, we have the object query loss L_0 for outputs from unmatched embeddings. And second, we have the track query loss L_1 for outputs from matched embeddings that will be overtaken to the next time period as track queries. Formally, the query loss is given by

$$\mathrm{L}_{\text{query}} = \begin{cases} L_0 & = -\lambda_{\text{cls}} \log \hat{p}_i(\emptyset) & \text{if } i \notin \pi \\ L_1 & = -\lambda_{\text{cls}} \log \hat{p}_i(c_{\pi=i}) + \mathrm{L}_{\text{box}}(b_{\pi=i}, \hat{b}_i) & \text{if } i \in \pi. \end{cases} \tag{5.6}$$

The expression captures two features: first, L_1 rewards track queries that find the right bounding box for objects that are still present on the current frame and it rewards object queries with similar outputs to new objects. Second, L_0 not only rewards track queries that predict the background class if their object has left the scene but also object queries that predict the background class if their bounding box prediction is off. The discussed details about track and object queries, and the matching rules with examples for assigned bounding boxes and losses are depicted in Figure 5.4.

Track query re-identification. What happens if objects are occluded or re-enter the scene? To deal with such cases, we keep feeding previously removed track queries for a *patience window* of $T_{track-reid}$ frames into the decoder. During this window, predictions from track ids are only considered if a classification score higher than $\sigma_{track-reid}$ is reached.

Results. TransformTrack achieved state-of-the-art performance in multi-object tracking on MOT17 and MOT20 datasets.

5.3 A Transformer for Multi-Camera Multi-Object Tracking

This section is organized into three parts. First, we outline the process of transforming our dataset, comprising multiple image sequences and sets of camera calibration parameters, into a unified 3D dataset. Next, we present the architectural modifications applied to Trackformer to adapt it for multi-camera tracking. Finally, we discuss the differences between our proposed model and Trackformer, encompassing not only architectural changes but also modifications to the training pipeline.

3D Tracking Data. To establish a cohesive representation of the scene across all camera perspectives, we introduce two transformations: $\mathcal{T}_{3D \mapsto 2D}$ for the 2D image plane to 3D world mapping and $\mathcal{T}_{2D \mapsto 3D}$ for the inverse mapping (as discussed in Chapter 4). By employing $\mathcal{T}_{2D \mapsto 3D}$, we derive a 3D cylinder associated with each 2D bounding box. The four values that define each cylinder are computed as follows:

Base Point: We project the bottom-middle point of the bounding box onto the 3D ground plane.

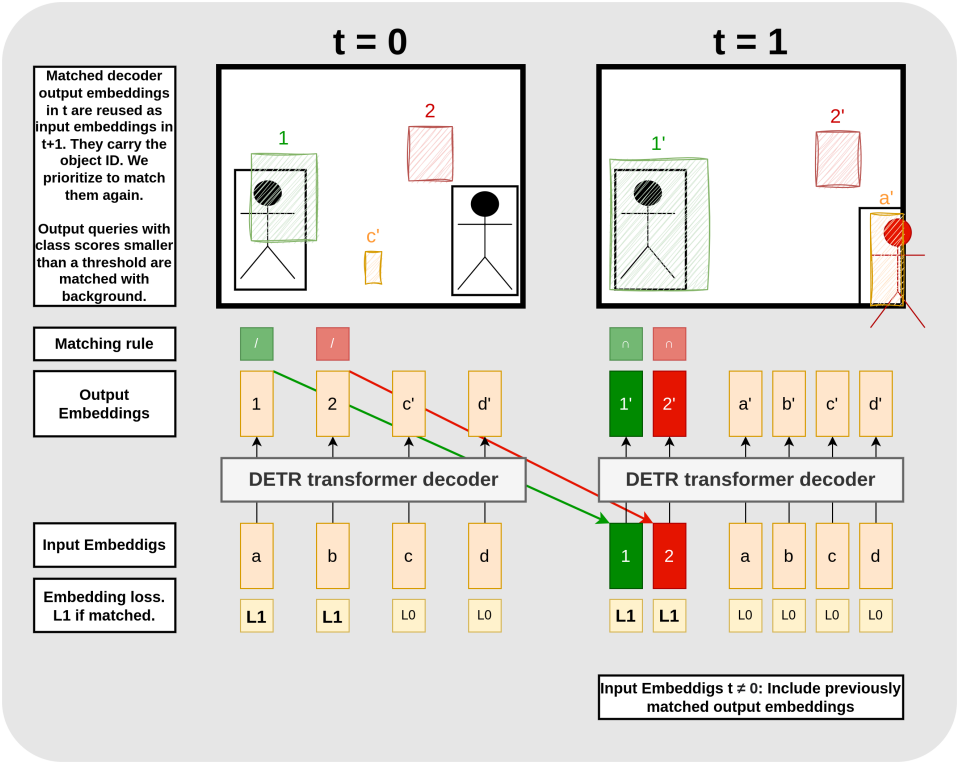

Figure 5.4: Training track and object queries. The black boxes are ground truth detections and the colorful, annotated boxes are predictions from the respective output embedding. Embeddings that do not spawn a box predict a class score smaller than the threshold. In t=0, the most similar boxes (green and red) are matched with the two target boxes according to matching step 2: K_{init} (symbolized by "/"). For these boxes, we compute L_1 based on boxes and class scores, and for the unmatched boxes, we compute L_0 solely based on the class scores. Then we update the model parameters accordingly. The matched output embeddings are taken over as additional input embeddings, carrying the object IDs from the objects on the previous image. They are matched with priority according to matching step 1: K_{track} (symbolized by "∩"). Embedding 2' is matched although the bounding boxes from output embedding a' is more similar to the target. We feed embeddings $1'$ and $2'$ to the decoder in period $t = 2$ (see Figure 5.5).

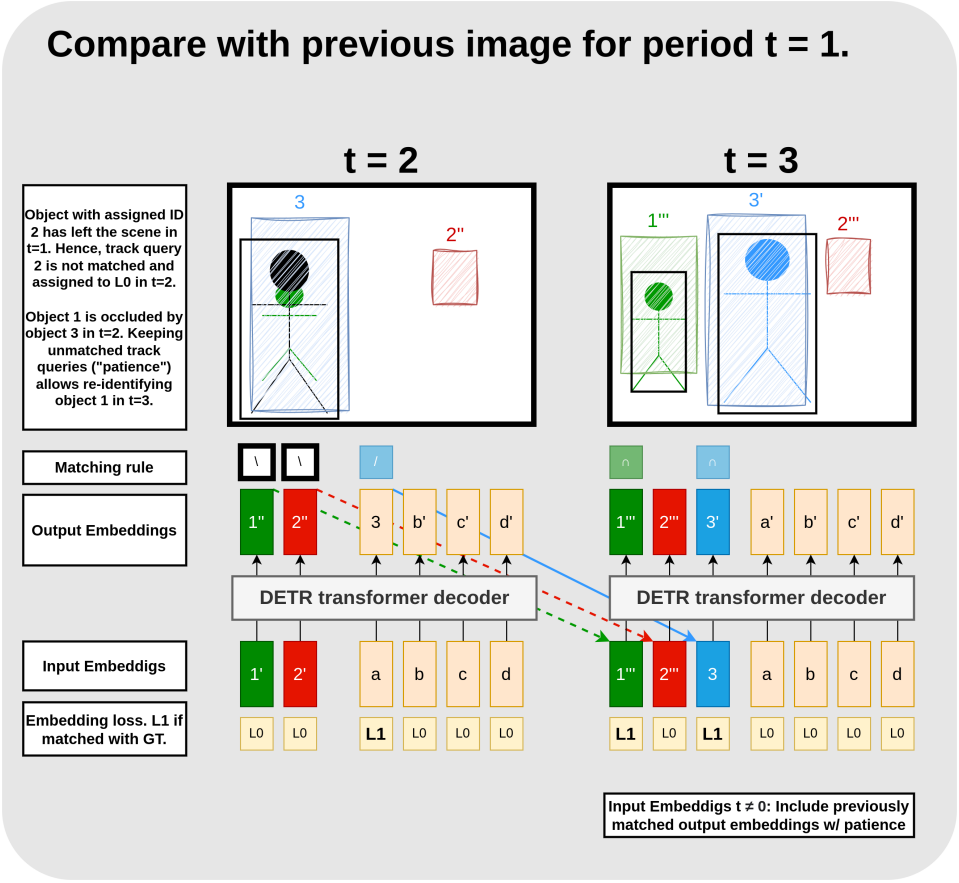

Figure 5.5: Training track queries with re-identification. Make sure to compare this Figure to Figure 5.4. From $t = 1$, we additionally feed output embeddings $1'$ and $2'$, carrying the object identities for pedestrian 1 (green) and pedestrian 2 (blue) to the encoder. However, pedestrian 2 has left the scene between $t = 1$ and $t = 2$ and pedestrian 1 is occluded by a news pedestrian with ID 3. Here, we depict the case where we have no ground truth annotation for the occluded pedestrian.

Note that the prediction from embedding $1'$ is quite reasonable given that pedestrian 1 is almost invisible. In contrast, embedding $2'$ keeps predicting a bounding box in the close to the upper right corner independent of the image.

Since there are no detections with IDs previously matched to 1 or 2, we cannot apply matching step 1: K_{track} (symbolized by "$\cap$") to output embeddings $1''$ and $2''$. Instead, we match both outputs with the background class according to step 2 K_{leaving} (symbolized by "$\setminus$"). Assuming the green pedestrian would be annotated, however, we would have apply step 1: K_{track} (symbolized by "$\cap$") to embedding $1''$ instead. To track the new pedestrian with ID 3, we apply matching step 3: K_{init} (symbolized by "$/$") to embedding 3 and update the model according to the respective losses. Since we keep unmatched embeddings with patience, we take over embeddings $1''$ and $2''$ to the next frame in addition to embedding 3. This allows the model to re-identify object 1 (green) in period $t = 3$.

Radius: A bottom corner of the bounding box is projected onto the 3D ground plane, and the distance between this point and the base point determines the cylinder's radius.

Height: The top-middle point of the bounding box is projected onto the ground plane. Through the intersection of the vertical line passing through the base point and the line connecting the projection to the world-coordinate camera center, we compute the cylinder's height.

The inverse procedure of computing bounding boxes from cylinders using $\mathcal{T}_{3D \mapsto 2D}$ is comparatively straightforward. It entails projecting the relevant point from the cylinder shell onto the image plane, perpendicular to the line pass through camera center and the 3D cylinder base.

With the mapping from 2D bounding boxes to 3D cylinders, we calculate a unified 3D training dataset using the information from all cameras. Specifically, we take the mean of all cylinders corresponding to the same object at a given moment across all cameras. Aggregating the cylinders from all views is necessary due to small variations arising from calibration inaccuracies. The resulting dataset allows us to train a model that is capable of predicting 3D cylinder targets.

During evaluation, we project the model's 3D outputs back onto each 2D image plane, filter objects based on visibility in the respective view, and measure the performance using the original 2D ground truth data and our standard tracking metrics.

The multi-camera tracker architecture. We expand Trackformer to handle datasets with $v > 1$ cameras and overlapping views of the same scene, along with camera calibration parameters,´ with the following three components:

Append image tokens from different cameras. We feed the v images into the same CNN and obtain v feature maps with dimensions $[W, H, D]$. Each feature map receives a camera-specific positional encoding, initialized as a vector of length equal to the number of feature channels D, expanded to the width and height dimensions, and added to its respective feature map. These encodings are crucial for the permutation invariant transformer to distinguish between cameras. We then pass $v \times W \times H$ image tokens, each with dimension D, to the transformer encoder.

Cylinder prediction head. Given that both 3D cylinders and 2D bounding boxes are defined by four values, we repurpose DETR's bounding box prediction head to predict (x_c, y_c, h, r). Here, x_c and y_c represent the first two coordinates of the cylinder's foot point on the 3D ground plane, while h and r denote the cylinder's height and radius, respectively.

DETR instead Deformable DETR. Unlike Trackformer, we cannot employ Deformable DETR (Zhu et al. [109]; summary in Appendix **??**) as our transformer-based object detector. The reason is that Deformable DETR associates the center point of an object detection with each object query. It uses this center point to select a section of the next period's feature map around the query for cross-attention and to predict the next center point by adding an offset to the current center point. This concept does not extend naturally to multiple images of the same scene. Consequently, foregoing Deformable DETR increases the model's reliance on implicitly encoding positional information of previous detections into the track queries, making it more challenging for all queries to attend correctly to the relevant parts of the feature map.

Figure 5.6 illustrates the transformer module, inputs, and outputs for one time period and two cameras, providing an example of the model's operation. We can extend the architecture for a full image sequence as shown in Figure 5.3. In essence, we leverage the transformer's ability to learn complex functions with minimal additional guidance. Our expectation is that it will learn the mapping $\mathcal{T}_{2D \mapsto 3D}$ for every camera without relying on calibration parameters, aggregate different feature maps into a unified scene, identify identical objects across views, and track all individual objects over time.

Differences from Trackformer. This section outlines the distinctions between the multi-camera tracking transformer and its training methodology in comparison to Trackformer (Meinhardt et al. [64]). There are seven primary points of contrast:

Training with tokens from multiple feature maps and camera embeddings. To enhance tracking accuracy amidst occlusions, we leverage data from multiple cameras. As a related detail, our baseline does not entail 1) merging the current feature maps with the previous one or 2) a 3D instead of a 2D sinusoidal embedding because [64] do not ablate the impact of these components.

Training data with target 3D cylinder detections. The approach involves predicting a unified 3D representation as opposed to separate 2D representations for each camera. However, the 2D representation can be retrieved by projecting the 3D cylinders as bounding boxes onto the image plane.

Impact of camera calibration errors. Minor calibration errors introduce a degree of imprecision in our 3D training data. Consequently, there is a reduction in evaluation performance when projecting cylinders onto the image plane.

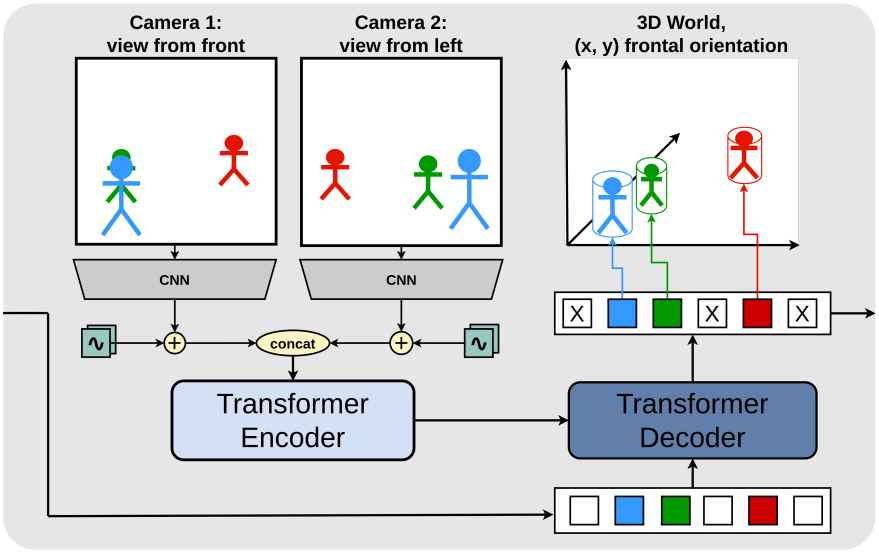

Figure 5.6: Rolled up multi-camera multi-object transformer at inference time. Compare this Figure to Figure 5.3. In contrast to Trackformer (Meinhardt et al. [64]), we pass multiple images (here two images) to the same CNN. In addition to the previous positional encoding, we add a learned positional embedding that differs across cameras to every feature map.This allows the transformer module to distinguish tokens from different cameras. We then pass these tokens together to the transformer encoder. The second difference is that we now predict the four parameters of a cylinder in a joint 3D space for all images instead of the bounding boxes for one image. This allows us to better follow objects through occlusions, especially if there is no annotation for an occluded object on one camera. As an example, we see that the green pedestrian is occluded in the view of camera 1 but not from the view of camera 2. With the additional information from camera 2, the model can track the green pedestrian in 3D space. We can further use these predictions to project the cylinders back onto the image plane of each camera as bounding boxes and evaluate our results in this space instead.

Absence of GIoU loss. Because we are not predicting bounding boxes, we exclude the Generalized Intersection over Union (GIoU) loss from our bounding box loss function and complete solely the L1 loss.

DETR as detector instead of Deformable DETR. Our model relies on the DETR detector, which is less efficient than Trackformer's. Consequently, we are unable to initialize our model weights using those from Trackformer.

Limited availability of data. The availability of multi-camera tracking data is comparatively restricted in contrast to single-camera tracking. Additionally, we are unable to leverage the extensive CrowdHuman dataset [85] with simulated adjacent frames, a resource that significantly contributed to the performance of [64].

Constraints on image augmentations. Certain image augmentations that obscure the projection relationship between images and the unified 3D space are not viable for our approach. For instance, operations such as cropping, flipping, and resizing cannot be applied. However, other transformations like erasing, blurring, and colorization remain applicable.

Chapter 6

Results

The chapter is organized as follows: firstly, we briefly recapitulate the data used in this study and outline its division into training, testing, and validation sets.

Secondly, we evaluate Trackformer (Meinhardt et al. [64]) without the multi-camera expansion, examining its performance on the WILDTRACK dataset and comparing it against other multi-camera models.This allows two things: 1) Trackformer's performance provides a baseline against which we can compare the multi-camera extension. 2) We will be able to measure our multi-camera model's performance against other multi-camera models.

In the third section of this chapter, we present the current status of our implementation.

In the fourth section, we discuss unresolved elements of the implementation.

Lastly, we address potential advantages and drawbacks of our approach from a conceptual standpoint.

6.1 Data

We use the WILDTRACK dataset that depicts seven videos from cameras with overlapping view of the ETH Zürich Campus. Each camera sequence corresponds to 400 images. Figure 6.1 shows that the annotations lack quality. Randomly omitted annotations like the man with blue shirt in the upper left picture prevents models to achieve perfect predictions because, in contrast to the people in the far background, these omissions are not a function of the image data. Apart from this, we will use a 320/40/40 training/test/validation split.

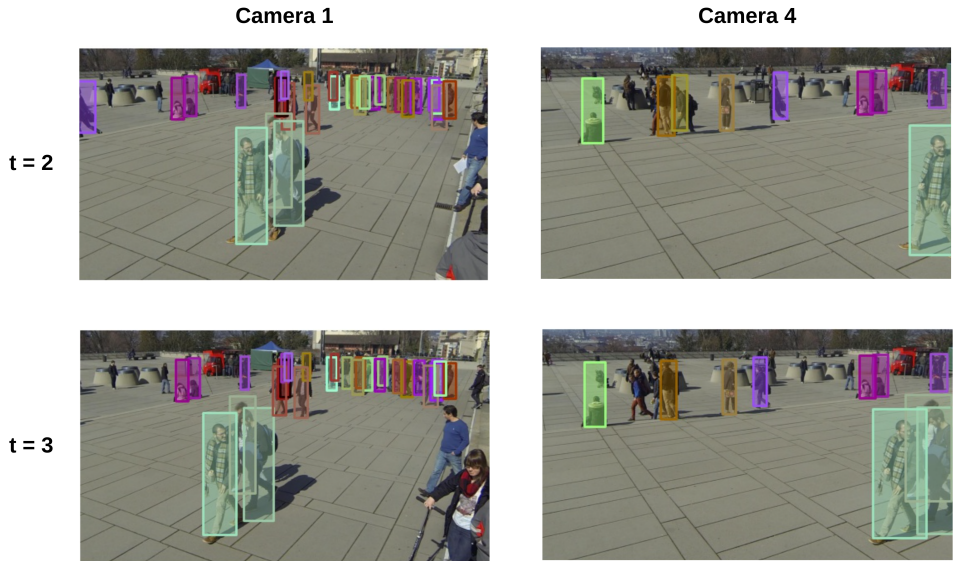

Figure 6.1: WILDTRACK annotations. Annotations in the WILDTRACK dataset are characterized by distinct bounding box colors corresponding to different track IDs. The figure showcases camera views, with the left column presenting camera 1 and the right column depicting camera 4. Notably, camera 4 offers a leftward perspective relative to camera 1. The upper row corresponds to time period 2, while the lower row pertains to time period 3. It's worth noting that certain annotations are absent, as evident in the unmarked individual in blue clothes captured by camera 1 at $t = 3$. Furthermore, annotations exclusively encompass the foreground, exemplified by the depiction from camera 4 at $t = 3$. Another observation is the nearly uniform bounding box height, exemplified by the seated women in the pink boxes in the background of each image.

6.2 Trackformer Evaluation

In this section, we conduct an evaluation of Trackformer (Meinhardt et al. [64]) in its original form on the WILDTRACK dataset. This approach serves a dual purpose: firstly, the Trackformer results establish a baseline to gauge the influence of the multi-camera extension in the future; secondly, it enables a comparative assessment of the multi-camera model against other existing multi-camera models.

Table 6.1 presents a comparison of the attained evaluation metrics for single-camera Trackformer, computed by us, and eight multi-camera models or model variants, using results extracted from the respective publications. We assess them across eight metrics, which we reiterate here for improved clarity:

1. **IDF1** balances precision and recall to provide a comprehensive evaluation of track identity maintenance over the complete trajectory.

2. **MOTA** (MOT Accuracy) quantifies frame-to-frame matching between predicted tracks and ground truth, penalizing identity switches and emphasizing detection accuracy. It summarizes false positives (**FP**), false negatives (**FN**), and identity switches (**IDSw**) as a percentage of ground truth objects.

3. Track Quality Measures: These classify trajectories as mostly tracked (**MT**), or mostly lost (**ML**) based on tracking success ratios. MT indicates targets tracked for at least 80% of their lifespan, and ML denotes targets tracked for less than 20%,

We observe that Trackformer can compete with earlier models KSP-DO and GLMB in terms of detection-based measures such as MOTA, MT, ML, FP, and FN. Specifically, Trackformer achieves 71.1% MOTA, 69.1% MT, 5.1% ML, 555 FP, and 683 FN. Apart from MOTA, Trackformer's results are superior to the better KSP-DO variant, KSP-DO-ptrack, and partially better and partially worse than the two DMCT and GLMB variants.

However, concerning metrics that are more sensitive to accurate ID association between frames, namely IDF1 and ID Switches, Trackformer performs significantly worse than all multi-camera models. For instance, Trackformer achieves an IDF1 of 68.1% and 243 ID switches, while the worst multi-camera model, KSP-DO, achieves a 73.2% IDF1 and only 85 ID Switches.

Furthermore, the best multi-camera model, LMGP, significantly outperforms Trackformer.

This outcome is anticipated since Trackformer doesn't utilize information from all cameras collectively, making it particularly challenging to maintain consistent track IDs through occlusions. Consequently, to harness Trackformer's robust

single-camera tracking capabilities in a multi-camera context, appropriate modifications are necessary. Our endeavors in this regard are detailed in the subsequent section.

Method	IDF1 (%) ↑	MOTA (%) ↑	MT (%) ↑	ML (%) ↓	FP ↓	FN ↓	IDSw ↓
LMGP [67]	98.2	97.1	97.6	1.3	71	7	12
KSP-DO [14]	73.2	69.6	28.7	25.1	1095	7503	85
KSP-DO-ptrack [14]	78.4	72.2	42.1	14.6	2007	5830	103
GLMB-YOLOv3 [68]	74.3	69.7	79.5	21.6	424	1333	104
GLMB-DO [68]	72.5	70.1	93.6	22.8	960	990	107
DMCT [103]	77.8	72.8	61.0	4.9	91	126	42
DMCT Stack [103]	81.9	74.6	65.9	4.9	114	107	21
Trackformer [64] (single-camera baseline)	68.1	71.1	69.1	5.1	555	683	243

Table 6.1: Results achieved on the WILDTRACK dataset by a selection of tracking models. We compare evaluation metrics for single-camera Trackformer, calculated by us, and eight multi-camera models or model variants, utilizing results extracted from the respective publications. We assess these models using eight metrics. In terms of metrics highly responsive to ID association, namely IDF1 and ID switches, Trackformer demonstrates significantly inferior performance compared to the multi-camera models. This discrepancy is likely due to Trackformer's inability to harness the additional spatial dimension resulting from the joint analysis of images from multiple viewpoints.

6.3 Multi-Camera Implementation

Leveraging the PyTorch-based research code by Meinhardt et al. [64] as a foundation, we introduce a series of enhancements to improve the model's adaptability to multi-camera data. The key achievements of our implementation include:

1. **Multi-camera tracking model.** We have successfully integrated our multi-camera tracking model.

2. **DETR as the object detector.** We have transitioned from Deformable DETR to vanilla DETR. We also enable the user to use pre-trained detection embeddings by adjusting the transformer's parameter dimensions.

3. **Dataloader and dataset for multi-camera tracking.** We provide the multi-camera dataset in COCO [60] format, alongside a dedicated dataloader for iterating over images from multiple cameras within the same time period.

4. **Dataloaders and intermediate datasets for single-camera tracking.** Our implementation generates two intermediate WILDTRACK datasets. The first dataset is used by a dataloader tailored for training and detection evaluation in COCO format. Additionally, we have written a custom dataloader for intermediate data in MOT format for tracking evaluation.

5. **Preprocessing.** We min-max scale the cylinder target annotations to train the model and map its outputs back to the respective image size to evaluate the model's performance.

6. **2D-3D transformations.** We have implemented the functions $\mathcal{T}_{3D \mapsto 2D}$ and $\mathcal{T}_{2D \mapsto 3D}$ for converting annotations from 3D cylinders to 2D bounding boxes and back. These methods are utilized to generate, learn, and evaluate our 3D training data. To ensure their accuracy, we rigorously tested these methods using unit tests, integration tests, and visual inspections. Additionally, we have provided vectorized PyTorch implementations for efficient evaluation of projected 3D model outputs on the original 2D data.

7. **Visualization.** The vectorized functions are also used to visualize the bounding boxes during training and evaluation.

8. **Flexible learning rate configurations.** We have incorporated fine-grained learning rate configurations, allowing users to set distinct learning rates for different transformer layers, thereby enhancing model convergence and performance. We also have the option to monitor their change during the training process.

In summary, our multi-camera implementation builds upon the PyTorch-based research code developed by Meinhardt et al. (2022) and introduces a comprehensive array of improvements aimed at enhancing adaptability for multi-camera data. These enhancements encompass the integration of a multi-camera tracking model, a transition from Deformable DETR to vanilla DETR for object detection, dedicated dataloaders and datasets tailored for both multi-camera and single-camera tracking, advanced preprocessing strategies, meticulously validated 2D-3D annotation transformations, visualization capabilities, and the incorporation of flexible learning rate configurations.

The code is available at https://github.com/tostenzel/mcmot-transformer.

6.4 Unresolved Implementation Elements

The tracking evaluation could not be finalized during the processing time of this thesis due to the high complexity of the Trackformer (Meinhardt et al. [64]), Deformable DETR (Zhu et al. [109]), and DETR (Carion et al. [12]) codebases.

6.5 Discussion

In this section, we critically examine the model architecture from a conceptual standpoint, identifying three potential advantages and three drawbacks.

The potential upsides are:

1. **Comprehensive contextual integration and adaptation:** Transformers excel at capturing long-range dependencies, contextual relationships, efficient information fusion, and adaptability to varying environments. These capabilities could empower the architecture to integrate contextual information from diverse viewpoints, dynamically adapt to changing conditions, flexibly tailor its model, and reduce engineering complexity, possibly leading to enhanced accuracy and adaptability in multi-camera multi-object tracking.

2. **Scalability to numerous cameras:** Transformers handle large-scale data processing effectively, conceivably making them suitable for scaling to numerous cameras while maintaining efficiency.

3. **End-to-end learning with reduced engineering complexity:** The architecture could possibly facilitate joint learning of feature extraction, object detection, and tracking, fostering synergy, potentially improving overall tracking performance, and simplifying complex engineering tasks such as data synchronization, feature extraction, and fusion in multi-camera setups.

The disadvantages are:

1. **Scarcity of multi-camera tracking data:** The concern stems from the scarcity of multi-camera tracking data in comparison to the single-camera tracking data required by Trackformer. Even combining all multi-camera datasets still falls short of the extensive training data utilized by Trackformer. Consequently, the full potential of the multi-camera transformer may not be realized when applied to real-world data.

2. **Lack of guidance in learning camera-specific transformation functions:** Firstly, the model lacks guidance in learning camera-specific transformation functions from the image plane to 3D space, even though relevant calibration data is given. Consequently, there exists untapped potential for enhancing the model's performance by incorporating this information. A second issue is the reliance on imprecise calibration information during the creation of our 3D training set, resulting in the model learning the associated errors. Moreover, during model evaluation, the projection of cylinders onto camera-specific image planes exacerbates the impact of calibration errors.

3. **Drawbacks Linked to Reliance on DETR:** Two points are linked to our reliance on DETR instead of Deformable DETR. Notably, DETR exhibits slower training times. The utilization of potentially subpar detections in tracking could negatively impact the tracking outcomes. Additionally, we cannot reuse the transformer weights from Trackformer because of the different transformer module and dimensionality. This makes the optimization problem even harder.

In this analysis, we assess the model architecture's conceptual aspects, highlighting potential advantages and drawbacks. Notably, the model's primary challenge revolves around the scarcity of multi-camera tracking data compared to the single-camera data used by Trackformer, potentially hindering the realization of the multi-camera transformer's full potential for real-world applications. The model's strengths include its capacity for contextual integration, scalability to multiple cameras, and streamlined end-to-end learning with reduced engineering complexity. Conversely, challenges encompass the lack of guidance in learning camera-specific transformations, reliance on limited calibration data, and issues arising from using DETR instead of Deformable DETR, which impact training efficiency and detection quality.

Chapter 7

Conclusion

In conclusion of this thesis, we first summarize our contributions, and second, we expound upon areas that we believe warrant further research.

7.1 Summary

In this study, we introduced a transformer model tailored for multi-camera multi-object tracking. Our framework is applicable to datasets comprising multiple image sequences of the same scene, captured from diverse viewpoints and characterized by distinct camera calibration parameters. Building upon the foundations of the single-camera model Trackformer (Meinhardt et al. [64]) and the transformer-based object detector DETR (Carion et al. [12]), our model and the implementation comprises three key components:

1. **Image tokens from multiple cameras.** For each camera, we feed the image through a CNN backbone and incorporate camera-specific learned embeddings. Subsequently, we pass the sequence of image tokens from all cameras through a transformer encoder.

2. **Object and track queries.** We train distinct decoder input embeddings called object queries. After decoding, each object query has the potential to initiate a new object detection. Output embeddings previously matched to ground truth detections, serve as track queries to the decoder, aiming to identify the same object from the next set of images.

3. **Cylinders as joint 3D detections.** Departing from direct prediction of 2D bounding boxes on the image plane for each camera, we instead model cylinders in the 3D joint world, leveraging camera calibration parameters. This

approach enables us to acquire a dynamically coherent scene representation that can be projected back onto various image planes for evaluation purposes.

Conceptually, we posit that the primary challenge encountered by our model is the small size and low number of available datasets. Notably, Meinhardt et al. [64] trained their single-camera model on a substantially larger dataset compared to what is available for the multi-camera scenario.

7.2 Future Work

We propose the following points for future work:

1. The first step is to complete and test the tracking evaluation for our multi-camera model as the final part of model implementation. With that, we can optimize and train the model parameters, run ablations, and transition from conceptual considerations to empirically based statements about the model.

2. Our current multi-camera model's definition necessitates the learning of functions that map points on the 2D image plane of one image to the joint 3D world from scratch, despite the availability of definitions from the camera calibrations. Exploring ways to equip the model with this information could enhance its convergence and performance.

3. A transformer-based multi-camera approach would likely benefit significantly from additional available data, in addition to, for instance, WILDTRACK (Chavdarova et al. [14]) and MMPTRACK (Han et al. [33]). This should be a top priority for future work.

Bibliography

[1] Imran Ahmed et al. “Towards Collaborative Robotics in Top View Surveillance: A Framework for Multiple Object Tracking by Detection Using Deep Learning”. en. In: *IEEE/CAA Journal of Automatica Sinica* 8.7 (July 2021), pp. 1253–1270 (Cited on page 5).

[2] Alexandre Alahi et al. “Social lstm: Human trajectory prediction in crowded spaces”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 961–971 (Cited on page 10).

[3] Xavier Alameda-Pineda et al. “SALSA: A Novel Dataset for Multimodal Group Behavior Analysis”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38.8 (Aug. 2016). Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 1707–1720 (Cited on page 14).

[4] Anton Andriyenko and Konrad Schindler. “Multi-target tracking by continuous energy minimization”. In: *CVPR 2011*. IEEE, 2011, pp. 1265–1272 (Cited on page 10).

[5] Mustafa Murat Arat. *Backpropagation Through Time for Recurrent Neural Network*. Feb. 2019. URL: https://mmuratarat.github.io/2019-02-07/bptt-of-rnn (visited on 02/28/2023) (Cited on page 17).

[6] *Automatic differentiation*. en. Page Version ID: 1143708979. Mar. 2023. URL: https://en.wikipedia.org/w/index.php?title=Automatic_differentiation&oldid=1143708979 (visited on 03/21/2023) (Cited on page 17).

[7] Jerome Berclaz et al. “Multiple object tracking using k-shortest paths optimization”. In: *IEEE transactions on pattern analysis and machine intelligence* 33.9 (2011). Publisher: IEEE, pp. 1806–1819 (Cited on page 9).

[8] Philipp Bergmann, Tim Meinhardt, and Laura Leal-Taixe. "Tracking without bells and whistles". In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. arXiv:1903.05625 [cs]. Oct. 2019, pp. 941–951 (Cited on page 10).

[9] Keni Bernardin and Rainer Stiefelhagen. "Evaluating multiple object tracking performance: the clear mot metrics". In: *EURASIP Journal on Image and Video Processing* 2008 (2008). Publisher: Springer, pp. 1–10 (Cited on page 8).

[10] Yuri Boykov, Olga Veksler, and Ramin Zabih. "Fast approximate energy minimization via graph cuts". In: *IEEE Transactions on pattern analysis and machine intelligence* 23.11 (2001). Publisher: IEEE, pp. 1222–1239 (Cited on page 13).

[11] Guillem Brasó and Laura Leal-Taixé. "Learning a neural solver for multiple object tracking". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, pp. 6247–6257 (Cited on page 9).

[12] Nicolas Carion et al. "End-to-End Object Detection with Transformers". en. In: *Computer Vision – ECCV 2020*. Ed. by Andrea Vedaldi et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 213–229 (Cited on pages 2, 4, 10, 53–56, 72, 74).

[13] Tatjana Chavdarova and Francois Fleuret. "Deep Multi-camera People Detection". In: *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*. Dec. 2017, pp. 848–853 (Cited on page 14).

[14] Tatjana Chavdarova et al. "WILDTRACK: A Multi-Camera HD Dataset for Dense Unscripted Pedestrian Detection". In: 2018, pp. 5030–5039 (Cited on pages 2, 13, 15, 70, 75).

[15] Long Chen et al. "Real-time multiple people tracking with deeply learned candidate selection and person re-identification". In: *2018 IEEE international conference on multimedia and expo (ICME)*. IEEE, 2018, pp. 1–6 (Cited on page 10).

[16] Hsu-kuang Chiu et al. *Probabilistic 3D Multi-Object Tracking for Autonomous Driving*. arXiv:2001.05673 [cs]. Jan. 2020. URL: http://arxiv.org/abs/2001.05673 (visited on 05/05/2023) (Cited on page 5).

[17] Wongun Choi and Silvio Savarese. “Multiple target tracking in world coordinate with single, minimally calibrated camera”. In: *Computer Vision–ECCV 2010: 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part IV 11*. Springer, 2010, pp. 553–567 (Cited on page 10).

[18] Peng Chu and Haibin Ling. “Famnet: Joint learning of feature, affinity and multi-dimensional assignment for online multiple object tracking”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 6172–6181 (Cited on page 10).

[19] G. Cybenko. “Approximation by superpositions of a sigmoidal function”. en. In: *Mathematics of Control, Signals and Systems* 2.4 (Dec. 1989), pp. 303–314 (Cited on page 29).

[20] Jifeng Dai et al. “Deformable Convolutional Networks”. In: 2017, pp. 764–773 (Cited on page 88).

[21] Patrick Dendorfer et al. *MOT20: A benchmark for multi object tracking in crowded scenes*. arXiv:2003.09003 [cs]. Mar. 2020. URL: http://arxiv.org/abs/2003.09003 (visited on 05/06/2023) (Cited on pages 11, 12).

[22] Patrick Dendorfer et al. “MOTChallenge: A Benchmark for Single-Camera Multiple Target Tracking”. en. In: *International Journal of Computer Vision* 129.4 (Apr. 2021), pp. 845–881 (Cited on pages 7, 9, 11).

[23] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159 (Cited on page 23).

[24] M. Elhoseny. “Multi-object Detection and Tracking (MODT) Machine Learning Model for Real-Time Video Surveillance Systems”. en. In: *Circuits, Systems, and Signal Processing* 39.2 (Feb. 2020), pp. 611–630 (Cited on page 5).

[25] A. Ellis and J. Ferryman. “PETS2010 and PETS2009 Evaluation of Results Using Individual Ground Truthed Single Views”. In: *2010 7th IEEE International Conference on Advanced Video and Signal Based Surveillance*. Aug. 2010, pp. 135–142 (Cited on pages 7, 11, 14).

[26] L. Euler. “Problema algebraicum ob affectiones prorsus singulares”. In: *Opera omnia 1st series* 6 (1770), pp. 287–315 (Cited on page 49).

[27] Christoph Feichtenhofer, Axel Pinz, and Andrew Zisserman. “Detect to track and track to detect”. In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 3038–3046 (Cited on page 10).

[28] Matthias Feurer and Frank Hutter. “Hyperparameter Optimization”. en. In: *Automated Machine Learning: Methods, Systems, Challenges*. Ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. The Springer Series on Challenges in Machine Learning. Cham: Springer International Publishing, 2019, pp. 3–33 (Cited on page 24).

[29] Francois Fleuret et al. “Multicamera people tracking with a probabilistic occupancy map”. In: *IEEE transactions on pattern analysis and machine intelligence* 30.2 (2007). Publisher: IEEE, pp. 267–282 (Cited on pages 13, 14).

[30] Dirk Focken and Rainer Stiefelhagen. “Towards vision-based 3-d people tracking in a smart room”. In: *Proceedings. fourth ieee international conference on multimodal interfaces*. IEEE, 2002, pp. 400–405 (Cited on page 13).

[31] A Geiger et al. “Vision meets robotics: The KITTI dataset”. In: *The International Journal of Robotics Research* 32.11 (Sept. 2013). Publisher: SAGE Publications Ltd STM, pp. 1231–1237 (Cited on page 11).

[32] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016 (Cited on page 17).

[33] Xiaotian Han et al. “MMPTRACK: Large-Scale Densely Annotated Multi-Camera Multiple People Tracking Benchmark”. en. In: 2023, pp. 4860–4869 (Cited on pages 7, 15, 75).

[34] Boris Hanin and Mark Sellke. *Approximating Continuous Functions by ReLU Nets of Minimal Width*. arXiv:1710.11278 [cs, math, stat]. Mar. 2018. URL: http://arxiv.org/abs/1710.11278 (visited on 03/21/2023) (Cited on page 29).

[35] Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003 (Cited on pages 47, 50).

[36] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: 2016, pp. 770–778 (Cited on page 35).

[37] Kaiming He et al. *Mask R-CNN*. arXiv:1703.06870 [cs]. Jan. 2018. URL: http://arxiv.org/abs/1703.06870 (visited on 04/24/2023) (Cited on page 10).

[38] Yuhang He et al. “Multi-target multi-camera tracking by tracklet-to-target assignment”. In: *IEEE Transactions on Image Processing* 29 (2020). Publisher: IEEE, pp. 5191–5205 (Cited on page 13).

[39] Roberto Henschel et al. “Improvements to frank-wolfe optimization for multi-detector multi-object tracking”. In: *arXiv preprint arXiv:1705.08314* 8 (2017) (Cited on page 9).

[40] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. *Lecture 6a Overview of mini-batch gradient descent*. 2012. URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (visited on 02/03/2023) (Cited on page 23).

[41] Andrea Hornakova et al. "Lifted disjoint paths with application in multiple object tracking". In: *International Conference on Machine Learning*. PMLR, 2020, pp. 4364–4375 (Cited on page 9).

[42] Hao Jiang, Sidney Fels, and James J. Little. "A linear programming approach for multiple object tracking". In: *2007 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2007, pp. 1–8 (Cited on page 9).

[43] Jefkine Kafunah. *Backpropagation In Convolutional Neural Networks*. May 2016. URL: https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/ (visited on 02/28/2023) (Cited on page 17).

[44] Andrej Karpathy. "Connecting Images and Natural Language". Dissertation. Stanford University, Aug. 2016 (Cited on page 17).

[45] Andrej Karpathy. *The Transformer is a magnificient neural network architecture*. Oct. 2022. URL: https://twitter.com/karpathy/status/1582807367988654081?lang=en (visited on 03/03/2023) (Cited on page 17).

[46] Margret Keuper et al. "Motion segmentation & multiple object tracking by correlation co-clustering". In: *IEEE transactions on pattern analysis and machine intelligence* 42.1 (2018). Publisher: IEEE, pp. 140–153 (Cited on page 9).

[47] Saad M. Khan and Mubarak Shah. "A multiview approach to tracking people in crowded scenes using a planar homography constraint". In: *Computer Vision–ECCV 2006: 9th European Conference on Computer Vision, Graz, Austria, May 7-13, 2006, Proceedings, Part IV 9*. Springer, 2006, pp. 133–146 (Cited on page 13).

[48] Chanho Kim et al. "Multiple hypothesis tracking revisited". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 4696–4704 (Cited on page 9).

[49] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. arXiv:1412.6980 [cs]. Jan. 2017. URL: http://arxiv.org/abs/1412.6980 (visited on 03/21/2023) (Cited on page 23).

[50] H. W. Kuhn. “The Hungarian method for the assignment problem”. en. In: *Naval Research Logistics Quarterly* 2.1-2 (1955), pp. 83–97 (Cited on pages 6, 56).

[51] Long Lan et al. “Semi-online multi-people tracking by re-identification”. In: *International Journal of Computer Vision* 128.7 (2020). Publisher: Springer, pp. 1937–1955 (Cited on page 13).

[52] Laura Leal-Taixé. *Dynamic Vision and Learning: Computer Vision III: Detection, Segmentation and Tracking (CV3DST) (IN2375)*. 2022. URL: https://dvl.in.tum.de/teaching/cv3dst-ss22/ (visited on 05/05/2023) (Cited on pages 5–7).

[53] Laura Leal-Taixé, Cristian Canton-Ferrer, and Konrad Schindler. “Learning by tracking: Siamese CNN for robust target association”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*. 2016, pp. 33–40 (Cited on pages 9, 10).

[54] Laura Leal-Taixé, Gerard Pons-Moll, and Bodo Rosenhahn. “Everybody needs somebody: Modeling social and grouping behavior on a linear programming multiple people tracker”. In: *2011 IEEE international conference on computer vision workshops (ICCV workshops)*. IEEE, 2011, pp. 120–127 (Cited on page 10).

[55] Laura Leal-Taixé et al. “Learning an image-based motion context for multiple people tracking”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 3542–3549 (Cited on page 10).

[56] Laura Leal-Taixé et al. *MOTChallenge 2015: Towards a Benchmark for Multi-Target Tracking*. arXiv:1504.01942 [cs]. Apr. 2015. URL: http://arxiv.org/abs/1504.01942 (visited on 05/06/2023) (Cited on pages 11, 12).

[57] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (Nov. 1998). Conference Name: Proceedings of the IEEE, pp. 2278–2324 (Cited on pages 30, 33).

[58] Fei-Fei Li, Jiajun Wu, and Ruohan Gao. *CS231n: Deep Learning for Computer Vision*. 2018. URL: http://cs231n.stanford.edu/index.html (visited on 02/02/2023) (Cited on page 17).

[59] Tsung-Yi Lin et al. “Feature pyramid networks for object detection”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 2117–2125 (Cited on page 88).

[60] Tsung-Yi Lin et al. “Microsoft coco: Common objects in context”. In: *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*. Springer, 2014, pp. 740–755 (Cited on pages 54, 57, 71).

[61] L. Liu et al. “Deep learning for generic object detection”. In: *A Survey* (2018) (Cited on page 54).

[62] Qiankun Liu et al. “GSM: Graph Similarity Model for Multi-Object Tracking.” In: *IJCAI*. 2020, pp. 530–536 (Cited on page 10).

[63] Wenhan Luo et al. “Multiple object tracking: A literature review”. In: *Artificial intelligence* 293 (2021). Publisher: Elsevier, p. 103448 (Cited on page 5).

[64] Tim Meinhardt et al. *TrackFormer: Multi-Object Tracking with Transformers*. arXiv:2101.02702 [cs]. Apr. 2022. URL: http://arxiv.org/abs/2101.02702 (visited on 03/21/2023) (Cited on pages 2, 4, 9, 10, 53, 58, 59, 64–67, 69, 70, 72, 74, 75).

[65] Anton Milan et al. *MOT16: A Benchmark for Multi-Object Tracking*. arXiv: 1603.00831 [cs]. May 2016. URL: http://arxiv.org/abs/1603.00831 (visited on 05/06/2023) (Cited on pages 11, 12).

[66] Anurag Mittal and Larry S. Davis. “M2tracker: A multi-view approach to segmenting and tracking people in a cluttered scene using region-based stereo”. In: *ECCV (1)*. 2002, pp. 18–36 (Cited on page 13).

[67] Duy M. H. Nguyen et al. “LMGP: Lifted Multicut Meets Geometry Projections for Multi-Camera Multi-Object Tracking”. en. In: 2022, pp. 8866–8875 (Cited on pages 13, 16, 70).

[68] Jonah Ong et al. “A Bayesian filter for multi-view 3D multi-object tracking with occlusion handling”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.5 (2020). Publisher: IEEE, pp. 2246–2263 (Cited on pages 13, 70).

[69] Aljoša Ošep et al. “Track, then decide: Category-agnostic vision-based multi-object tracking”. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 3494–3501 (Cited on page 10).

[70] Jiangmiao Pang et al. “Quasi-dense similarity learning for multiple object tracking”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2021, pp. 164–173 (Cited on page 10).

[71] Youngmin Park, Vincent Lepetit, and Woontack Woo. "Multiple 3D Object tracking for augmented reality". In: *2008 7th IEEE/ACM International Symposium on Mixed and Augmented Reality*. Sept. 2008, pp. 117–120 (Cited on page 6).

[72] Niki Parmar et al. "Image transformer". In: *International conference on machine learning*. PMLR, 2018, pp. 4055–4064 (Cited on page 10).

[73] Stefano Pellegrini et al. "You'll never walk alone: Modeling social behavior for multi-target tracking". In: *2009 IEEE 12th international conference on computer vision*. IEEE, 2009, pp. 261–268 (Cited on page 10).

[74] Hamed Pirsiavash, Deva Ramanan, and Charless C. Fowlkes. "Globally-optimal greedy algorithms for tracking a variable number of objects". In: *CVPR 2011*. IEEE, 2011, pp. 1201–1208 (Cited on page 9).

[75] Lorenzo Porzi et al. "Learning Multi-Object Tracking and Segmentation From Automatic Annotations". English. In: IEEE Computer Society, June 2020, pp. 6845–6854 (Cited on page 10).

[76] Kha Gia Quach et al. "Dyglip: A dynamic graph model with link prediction for accurate multi-camera multiple object tracking". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 13784–13793 (Cited on page 13).

[77] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *Advances in Neural Information Processing Systems*. Vol. 28. Curran Associates, Inc., 2015 (Cited on page 57).

[78] Seyed Hamid Rezatofighi et al. "Multi-Target Tracking With Time-Varying Clutter Rate and Detection Profile: Application to Time-Lapse Cell Microscopy Sequences". In: *IEEE Transactions on Medical Imaging* 34.6 (June 2015). Conference Name: IEEE Transactions on Medical Imaging, pp. 1336–1348 (Cited on page 5).

[79] Ergys Ristani and Carlo Tomasi. "Features for multi-target multi-camera tracking and re-identification". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 6036–6046 (Cited on page 10).

[80] Ergys Ristani et al. "Performance Measures and a Data Set for Multi-target, Multi-camera Tracking". en. In: *Computer Vision – ECCV 2016 Workshops*. Ed. by Gang Hua and Hervé Jégou. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 17–35 (Cited on pages 8, 14).

[81] Alexandre Robicquet et al. “Learning social etiquette: Human trajectory understanding in crowded scenes”. In: *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part VIII 14*. Springer, 2016, pp. 549–565 (Cited on page 10).

[82] Thomas Romeas, Antoine Guldner, and Jocelyn Faubert. “3D-Multiple Object Tracking training task improves passing decision-making accuracy in soccer players”. en. In: *Psychology of Sport and Exercise* 22 (Jan. 2016), pp. 1–9 (Cited on page 5).

[83] Ing G. Sagerer, Ilker Savas, and Dipl-Inform Joachim Schmidt. “Entwicklung eines Systems zur visuellen Positionsbestimmung von Interaktionspartnern”. In: () (Cited on page 47).

[84] Paul Scovanner and Marshall F. Tappen. “Learning pedestrian dynamics from the real world”. In: *2009 IEEE 12th International Conference on Computer Vision*. IEEE, 2009, pp. 381–388 (Cited on page 10).

[85] Shuai Shao et al. “Crowdhuman: A benchmark for detecting human in a crowd. arXiv 2018”. In: *arXiv preprint arXiv:1805.00123* (2018) (Cited on page 66).

[86] Hao Sheng et al. “Heterogeneous association graph fusion for target association in multiple object tracking”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 29.11 (2018). Publisher: IEEE, pp. 3269–3280 (Cited on page 9).

[87] Ilya Sutskever et al. “On the importance of initialization and momentum in deep learning”. en. In: *Proceedings of the 30th International Conference on Machine Learning*. ISSN: 1938-7228. PMLR, May 2013, pp. 1139–1147 (Cited on page 23).

[88] Siyu Tang et al. “Multiple people tracking by lifted multicut and person re-identification”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 3539–3548 (Cited on page 9).

[89] Pavel Tokmakov et al. “Learning to track with object permanence”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 10860–10869 (Cited on page 10).

[90] Ashish Vaswani and Anna Huang. *Transformers and Self-Attention*. Mar. 2021. URL: https://www.youtube.com/watch?v=5vcj8kSwBCY (visited on 03/03/2023) (Cited on page 17).

[91] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017 (Cited on pages 10, 40, 43–46, 54, 55).

[92] Paul Voigtlaender et al. “Mots: Multi-object tracking and segmentation”. In: *Proceedings of the ieee/cvf conference on computer vision and pattern recognition*. 2019, pp. 7942–7951 (Cited on page 10).

[93] Yongxin Wang, Kris Kitani, and Xinshuo Weng. “Joint object detection and multi-object tracking with graph neural networks”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 13708–13715 (Cited on page 9).

[94] Longyin Wen et al. “Multi-camera multi-target tracking with space-time-view hyper-graph”. In: *International Journal of Computer Vision* 122 (2017). Publisher: Springer, pp. 313–333 (Cited on page 13).

[95] Longyin Wen et al. “UA-DETRAC: A new benchmark and protocol for multi-object detection and tracking”. en. In: *Computer Vision and Image Understanding* 193 (Apr. 2020), p. 102907 (Cited on page 11).

[96] Lilian Weng. *Attention? Attention!* June 2018. URL: `https://lilianweng.github.io/posts/2018-06-24-attention/` (visited on 01/25/2023) (Cited on pages 17, 41).

[97] Bo Wu and R. Nevatia. “Tracking of Multiple, Partially Occluded Humans based on Static Body Part Detection”. In: *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’06)*. Vol. 1. ISSN: 1063-6919. June 2006, pp. 951–958 (Cited on page 9).

[98] Jialian Wu et al. “Track to detect and segment: An online multi-object tracker”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2021, pp. 12352–12361 (Cited on page 10).

[99] Yuanlu Xu et al. “Cross-view people tracking by scene-centered spatio-temporal parsing”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 31. Issue: 1. 2017 (Cited on page 13).

[100] Yuanlu Xu et al. “Multi-view people tracking via hierarchical trajectory composition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 4256–4265 (Cited on pages 13, 14).

[101] Zhenbo Xu et al. “Segment as points for efficient online multi-object tracking and segmentation”. In: *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part I 16*. Springer, 2020, pp. 264–281 (Cited on page 10).

[102] Kota Yamaguchi et al. “Who are you with and where are you going?” In: *CVPR 2011*. ISSN: 1063-6919. June 2011, pp. 1345–1352 (Cited on pages 8, 10).

[103] Quanzeng You and Hao Jiang. "Real-time 3d deep multi-camera tracking". In: *arXiv preprint arXiv:2003.11753* (2020) (Cited on pages 13, 70).

[104] Qian Yu, Gérard Medioni, and Isaac Cohen. "Multiple target tracking using spatio-temporal markov chain monte carlo data association". In: *2007 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2007, pp. 1–8 (Cited on page 9).

[105] Li Zhang, Yuan Li, and Ramakant Nevatia. "Global data association for multi-object tracking using network flows". In: *2008 IEEE conference on computer vision and pattern recognition*. IEEE, 2008, pp. 1–8 (Cited on pages 9, 10).

[106] Zhong-Qiu Zhao et al. "Object Detection With Deep Learning: A Review". In: *IEEE Transactions on Neural Networks and Learning Systems* 30.11 (Nov. 2019). Conference Name: IEEE Transactions on Neural Networks and Learning Systems, pp. 3212–3232 (Cited on page 7).

[107] Xingyi Zhou, Vladlen Koltun, and Philipp Krähenbühl. "Tracking objects as points". In: *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part IV*. Springer, 2020, pp. 474–490 (Cited on page 10).

[108] Xizhou Zhu et al. "Deformable convnets v2: More deformable, better results". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 9308–9316 (Cited on page 88).

[109] Xizhou Zhu et al. *Deformable DETR: Deformable Transformers for End-to-End Object Detection*. arXiv:2010.04159 [cs]. Mar. 2021. URL: http://arxiv.org/abs/2010.04159 (visited on 03/21/2023) (Cited on pages 10, 57, 58, 64, 72, 88–90).

Appendix A

Program Code / Resources

The source code, a documentation, some usage examples, and additional test results are available at https://github.com/tostenzel/mcmot-transformer.

Appendix B

Deformable DETR

In order to tackle these challenges, Zhu et al. [109] introduces a deformable attention module as replacement for the conventional attention module in DETR's transformer model. Drawing inspiration from deformable convolution [20, 108], the deformable attention module focuses its attention solely on a limited set of key sampling points surrounding a reference point. By allocating a fixed number of keys per query, Zhu et al. can alleviate the problems associated with convergence and feature spatial resolution by decreasing the transformer's complexity as a function of the image dimension to a sub-quadratic level. Furthermore, Zhu et al. use this module to aggregate multiple feature maps of different resolution taken from the CNN backbone in the encoder to "multi-scale" feature maps and for the object queries to aggregate the relevant information from these maps for the detection predictions. The design is inspired by the finding that multi-scale feature maps are crucial for teaching image transformers to effectively represent objects depicted at strongly distinct scales [59]. Figure B.1 depicts the complete Deformable DETR architecture.

Multi-head Attention revisited. When provided with a query element (e.g., a target word in the output sentence) and a set of key elements (e.g., source words in the input sentence), the multi-head attention module aggregates the key contents based on attention weights, which gauge the compatibility of query-key pairs. In order to enable the model to focus on diverse representation subspaces and positions, the outputs of various attention heads are combined linearly using adjustable weights.

Let $q \in \Omega_q$ denote an index for the query element represented by feature $z_q \in \mathbb{R}^C$, and let $k \in \Omega_k$ denote an index for the key element represented by feature $x_k \in \mathbb{R}^C$ with feature dimension C and set of query and key elements Ω_q and Ω_k,

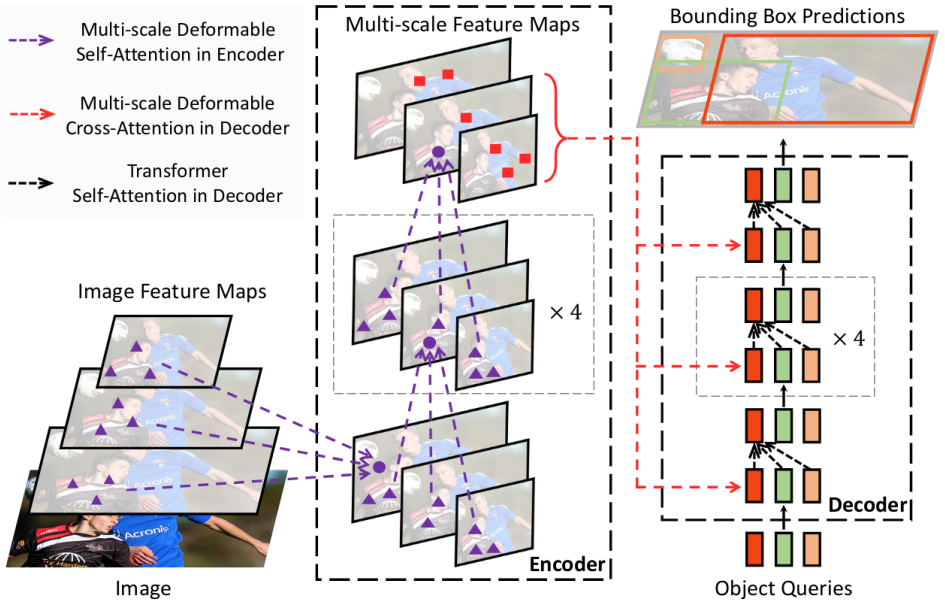

Figure B.1: High-level Deformable DETR architecture. We extract three feature maps at different resolution levels from the CNN backbone. In the encoder, we aggregate the information from all multi-scale feature maps with deformable self-attention, attending only to a learned sample of important locations around a learned reference point from every feature map for each feature map (deformable attention). This results in three encoder feature maps of the same dimensions. In the decoder, the learned object queries extract features from queries and values from themselves with conventional self-attention and from the keys from the encoder feature maps with deformable cross-attention. Again, for each object query, the learned reference point and a set of learned offsets is used to only query a set of keys from the encoder. With that, we replace the self-attention modules in the encoder and the and cross-attention modules in the decoder with 4 heads of the deformable attention module and learn from feature maps at different resolutions. This makes the transformer's complexity as a function of pixel number subquadratic and allows the model to better learn objects at strongly distinct scales. Image source: Zhu et al. [109] .

respectively. We compute the multi-head attention feature for query index q with

$$\text{MultiHeadAttn}(z_q, x) = \sum_{m=1}^{M} W_m[\sum_{k\in\Omega_k} A_{mqk} \cdot W_m^T x_k], \tag{B.1}$$

where m is index the attention head and $W_m \in \mathbb{R}^{C\times C_v}$ are learned weights with $C_v = C/M$. We normalize the attention weights $A_{mqk} \propto \exp\{\frac{z_q^T U_m^T V_m x_k}{\sqrt{C_v}}\}$ with learned weights $U_m, V_m \in \mathbb{R}^{C\times C_v}$ to $\sum_{k\in\Omega_k} A_{mqk} = 1$. In order to clarify distinct spatial positions, the representation features, denoted as z_q and x_k, are formed by concatenating or summing the element contents with positional embeddings.

Deformable Attention. The main challenge in applying Transformer attention to image feature maps is that it considers all potential spatial locations. To overcome this limitation, Zhu et al. [109] propose a deformable attention module. It selectively focuses on a small number of key sampling points around a reference point, irrespective of the spatial dimensions of the feature maps. By employing a small number of keys per query, it can address the issues of convergence and feature spatial resolution. In contrast to the previous notation, let $x \in \mathbb{R}^{C\times H\times W}$ denote an input feature map and let q denote the index for a query element with feature z_q and a 2-d reference point p_q. We calculate the deformable attention feature with with

$$\text{DeformAttn}(z_q, p_q, x) = \sum_{m=1}^{M} W_m[\sum_{k\in\Omega_k} A_{mqk} \cdot W_m^T x_k(p_q + \Delta p_{mqk})], \tag{B.2}$$

where k indexed the sampled keys and $K \ll HW$ is the total number of sampled keys. Futher, $\Delta p_{mqk} \in \mathbb{R}^2$ denotes the sampling offset and A_{mkq} the attention weights of the k^{th} sampling point in the m^{th} attention head, respectively. The attention weights are normalized to own over the sample keys. We feed query feature z_q to a $3MK$-channel linear projection operator. The inital $2MK$ channels encode the sampling offsets $Deltap_{mkq}$, and the latter MK channels are used as input to the `softmax` function to compute the attention weights A_mqk.

Multi-scale Deformable Attention. We extend the deformable attention module to multiple differently-scaled feature maps with a few small changes. Let $\{x^l\}_{l=1}^L$ denote the set of differently scaled feature maps, where $x^l \in \mathbb{R}^{C\times H_l\times W_l}$ and let $\hat{p}_q \in [0,1]^2$ denote the normalized reference points for each query element element q. Then, we calculate the multi-scale deformable attention with

$$\text{MSDeformAttn}(z_q, \hat{p}_q, \{x^l\}_{l=1}^{L}) = \sum_{m=1}^{M} W_m [\sum_{l=1}^{L} \sum_{k \in \Omega_k} A_{mlqk} \cdot W_m^T x^l(\phi_l(\hat{p}_q) + \Delta p_{mlqk})], \quad \text{(B.3)}$$

where l indexes the input feature and we expand the normalized attention weights and the reference point by this dimension. $\hat{p}_q \in [0, 1]^2$ are also normalized coordinates with $(0, 0)$ and $(1, 1)$ as top-left and bottom-right coordinates, respectively. Function $\phi_l(\hat{p}_q)$ is the inverse of the normalization function and maps $\hat{p}_q$ back to the respective feature map coordinates. In constrast to the singlescale deformable attention module, we sample LK feature map points instead of K points and interact the different feature maps with each other.

Results. On the COCO benchmark for object detection, Deformable DETR outperforms DETR, particularly in detecting small objects, while requiring only about one-tenth of the training epochs.

Ehrenwörtliche Erklärung

Ich versichere, dass ich die beiliegende Masterarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Mannheim, den 09. 08. 2023