HOCHSCHULE LANDSHUT
HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN

Computer Science Faculty

Master's Thesis

Computer Vision Based Vehicular Traffic Irregularity Analysis

Lucas Sas Brunschier

Supervisor(s):

Eduard Kromer, Prof.Dr.
Joerg Sichermann, Dipl.-Geogr.
Julian Strosahl, M.Sc.

HOCHSCHULE LANDSHUT
HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN

FREIGABEERKLÄRUNG DER/DES STUDIERENDEN

Name, Vorname der/des
Studierenden: ....

Hiermit erkläre ich, dass die vorliegende Bachelor-/Masterarbeit in den Bestand der Hochschulbibliothek aufgenommen werden kann und

❑ ohne Sperrfrist

oder nach einer Sperrfrist von

❑ 1 Jahr

❑ 2 Jahren

❑ 3 Jahren

❑ 5 Jahren oder länger

über die Hochschulbibliothek zugänglich gemacht werden darf.

....	....
(Datum)	(Unterschrift der/des Studierenden)

ERKLÄRUNG ZUR MASTERARBEIT

Sas Brunschier, Lucas

Hochschule Landshut
Fakultät Informatik

Hiermit erkläre ich, dass ich die Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt, sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

....	....
(Datum)	(Unterschrift des Studierenden)

Abstract

Identifying key moments in large-scale datasets is very important in making collected data usable by other systems. In this thesis, multiple state-of-the-art anomaly detection and irregularity analysis techniques are applied to the vehicular and pedestrian traffic video surveillance domain. Additionally these techniquse are also evaluated on a new vehicular surveillance video dataset. The resulting evaluation of these techniques presents unique benefits and drawbacks of using specific methods for the target use case of this thesis and shows how such a system can be used in conjunction with other components to generate an autonomous end-to-end scenario database. This database can contain actor trajectory-, video- and metainformation of relevant recorded scenes. Additionally, this thesis reveals a clear underrepresentation of video anomaly detection techniques in the open-source community. To fill this white spot, a general and adaptable open-source framework for training and using video anomaly detection machine learning models is provided. The framework covers a wide selection of different video anomaly detection neural network architectures, both complex and minimalistic. Incorporating state-of-the-art packaging and documentation tools makes the framework easily accessible to users.

Contents

Tables

Figures

CHAPTER 1

Introduction

1.1 Collaboration with e:fs TechHub GmbH

This master's thesis is the result of a collaboration with *e:fs*[1] and the University of Applied Sciences Landshut. *e:fs* is a joint venture company of *AKKA Technologies* and *CARIAD SE*, a member of the Volkswagen Group. The thesis is part of a combined effort called *SAVeNoW*[2]. *SAVeNoW* is state-subsidized by the *Federal Ministry of Transport and Digital Infrastructure* of Germany. The overall goal of *SAVeNoW* is to construct a digital twin of Ingolstadt. The twin is constructed by leveraging multi-sensor observations across Ingolstadt. Besides visualizing the measured reality of the traffic situation in Ingolstadt, the historical data can also be used to run simulations on real data (Montanari et al., 2021). Concept visualization of the system is shown in figure 1.1. Even though collecting large amounts of data is great for many use cases, not every data point or series of data points is equally important. That's why it's of critical importance, to have a method to filter out relevant scenarios. This thesis primarily focuses on processing video data from a single traffic-surveillance camera (sometimes referred to as *FKK*[3]) located and pointed towards a

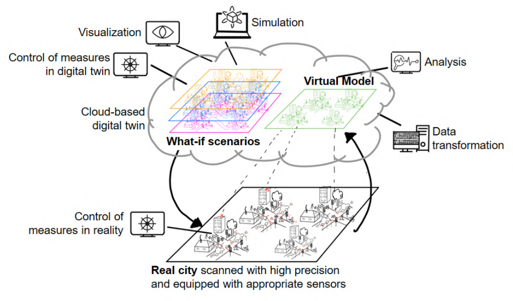

Visualization
Simulation
Control of measures in digital twin
Analysis
Virtual Model
Cloud-based digital twin
What-if scenarios
Data transformation
Control of measures in reality
Real city scanned with high precision and equipped with appropriate sensors

Figure 1.1: Concept visualization of the *digital twin* the *SAVeNoW* project tries to create.

[1]e:fs TechHub GmbH

[2]Members of the *SAVeNoW* project are 3-D-Mapping Solutions GmbH, ASAP Engineering GmbH, Conti Temic microelectronic GmbH, Deutsches Zentrum für Luft- und Raumfahrt, e:fs TechHub GmbH, Fraunhofer-Institut für Verkehrs- und Infrastruktursysteme, Katholische Universität Eichstätt-Ingolstadt, sepp.med GmbH, Technische Hochschule Ingolstadt, TWT GmbH Science & Innovation, Technische Universität München, Universität Stuttgart

[3]abbr. for "Forschungs-Kreuzungs-Kamera" (engl. Research-Intersection-Camera)

single intersection in Ingolstadt. The conceptual goal is to identify irregular events or situations captured by the camera stream. In the future, such a system could be used in conjunction with the previous master's thesis by "Perspective-Corrected Extraction of Trajectories from Urban Traffic Camera Using CNN" (Strosahl et al., 2022), which also originated from the *SAVeNoW* research project. The thesis focused exclusively on actor trajectory extraction of the *FKK* video-stream[4]. Figure 1.2 shows how the results of this thesis could be combined with "Perspective-Corrected Extraction of Trajectories from Urban Traffic Camera Using CNN" (Strosahl et al., 2022) to form an end-to-end video processing system, that encompasses anomaly detection and actor trajectory extraction. In a future effort, this system could be extended to multiple simultaneous camera streams.

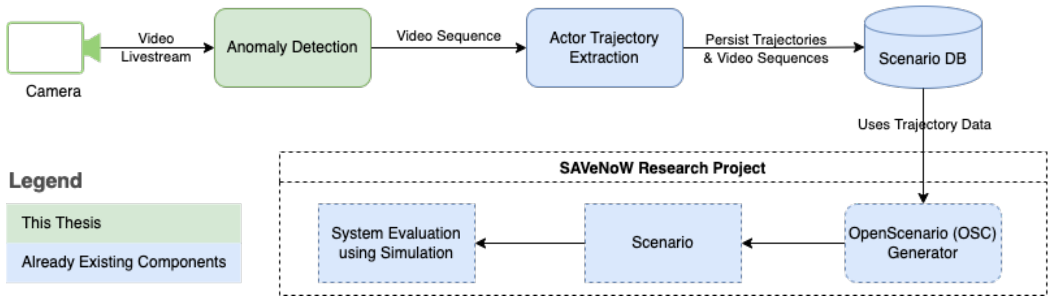

Camera
Video Livestream
Anomaly Detection
Video Sequence
Actor Trajectory Extraction
Persist Trajectories & Video Sequences
Scenario DB
Uses Trajectory Data
Legend
This Thesis
Already Existing Components
SAVeNoW Research Project
System Evaluation using Simulation
Scenario
OpenScenario (OSC) Generator

Figure 1.2: This concept diagram shows how the results of this thesis could be integrated with already existing company infrastructure. The anomaly detection system of this thesis (green) is used to trigger other systems that are already existing or planned. The resulting pipeline can automatically identify, process and persist interesting footage captured by a live stream camera.

1.2 Formal Problem Statement

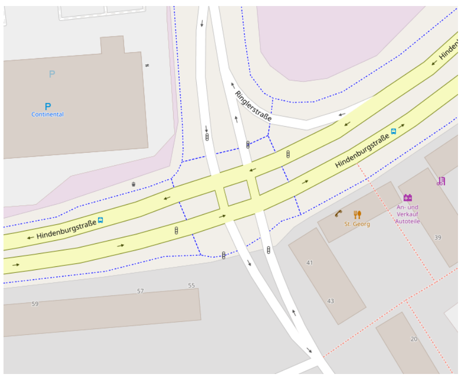


Figure 1.3: OpenStreet-Map view of the "Forschungs-Kreuzung" intersection. Source: openstreetmap.org

As previously stated in chapter 1.1, the primary goal of this thesis is to provide a way for *e:fs,* to identify and extract video-clips from a camera-stream (figure 1.4) in which actors exhibit anomalous (see 1.2.2) behavior. In the context of this thesis, actors are defined as autonomous objects or individuals that participate and interact with public traffic. Some basic examples would be cars, cyclists or pedestrians. The anomaly detection research field is already well-established and offers a variety of different approaches. Especially anomaly detection on vehicular traffic situations is a

[4]Open-source repository: `https://github.com/jul095/TrafficMonitoring`

research subject many companies are actively interested in. (Naphade et al., 2020; Zhao et al., 2021; Wu et al., 2021a) The traffic intersection this thesis is targeted towards (figure 1.3), features multiple lanes and complex actor behavior. The camera is mounted on top of a nearby building pointed toward the intersection, providing a good overview of all relevant areas for traffic surveillance. The system this thesis is trying to establish should work based on

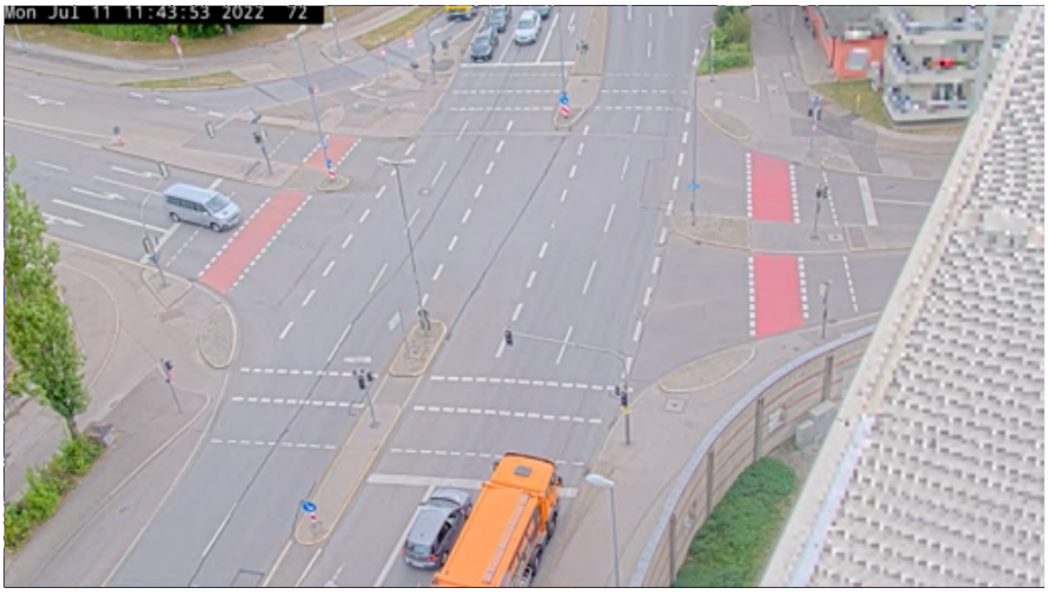


Figure 1.4: An excerpt frame of the surveillance camera's view over the *Hindenburg-Ringlerstraße* intersection in Ingolstadt.

the RGB video stream provided by the traffic surveillance camera, implying that the system has to work purely on pixel-level without any additional information like, for example, depth information or actor GPS annotations. Even though a single intersection is targeted, the system should be able to be applied to most similar environments with only minor adjustments. Adjustments could involve changing minor system parameters or for instance, performing retraining on an existing machine-learning model architecture.

1.2.1 Choice of Perspective

Video footage, typically encountered in image-based vehicular anomaly detection can be differentiated between three different camera perspectives.

1. Scenes that are captured from an egocentric perspective. A common example would be video footage filmed from cameras mounted behind car windshields. Modern cars with advanced driver assistance software often possess additional sensors that can be leveraged.

2. The perspective of a static camera with a fixed view of a traffic scene. Most of the time this footage comes from (traffic-) surveillance cameras. (Naphade et al., 2018) Often the camera is mounted at an angle, this implies that objects

are affected by the camera's perspective in the form of significant perspective distortion.

3. The third and most uncommon perspective is a top-down view of a traffic scene without any significant perspective in the scene. A good provider for this type of footage could be a drone equipped with a bottom-facing camera.

The second and third perspectives offer the best overview of the scene. Egocentric ground-level perspectives tend to be more susceptible to visual obstructions. The absolute top-down perspective also has the benefit, that all objects are scaled equally. Assuming that the distance to the ground is constant across all footage taken. Convolutional neural network architectures, often used for image processing are especially susceptible to differently scaled objects in a scene.

The *e:fs* use-case this thesis is targeted towards, is based on the second perspective previously listed. That's why the proposed systems of this thesis are also primarily designed to work with said perspective. Although, *e:fs* also has drone footage available for potential experiments.

1.2.2 Definition of Anomaly

The *Cambridge Dictionary* defines "Anomaly" as the following:

> A person or thing that is different from what is usual, or not in agreement with something else and therefore not satisfactory.

Because this thesis works primarily with unannotated data, separation into different types of anomalies is not feasible. The algorithm will find its own "definition" of an anomaly by learning a probability distribution P' from training data P. All sampled data points that differ significantly from distribution P' are classified as anomaly candidates by the algorithm. Subsequently, the input data should contain as few anomalies as possible, so that the algorithm will not learn a distribution in which anomalies are present. This approach is common in the unsupervised anomaly detection problem space.

1.3 Strategy

1.3.1 Why is Machine-Learning suitable for Video Vehicular Anomaly Detection

Especially with machine learning projects, it's often advisable to reflect on if it's even appropriate to solve a problem with the help of machine learning techniques. Often classic algorithms are more performant, predictable and maintainable, while also not requiring the amount of training data a machine-learning solution would require.

Image data is very informationally dense and therefore hard to abstract without any machine learning techniques. Especially the convolutional neural network architecture dominates classic object detection computer vision algorithms in both performance and usage in the industry (Szeliski, 2011; Zou et al., 2019). Its ability to abstract very high-level information from low-level input space is very valuable for computer-vision problems. But this thesis primarily tackles the anomaly detection problem and not an abstraction problem over the video data space. Therefore, a hybrid approach, in which the abstraction would be done with a state-of-the-art object detector and the anomaly detection would be purely done by a hand-crafted algorithm.

For example, a rule-based agent could identify if a car's spatial coordinate/trajectory is allowed for a timepoint T. Anomalies could then even be categorized into classes. For example: The complexity of each category would be dependent on the imple-

Anomaly Description	Spatial Context	Temporal Context
car runs red light	✓	✓
pedestrian present in an illegal position	✓	
actor stops without any reason	✓	✓
vehicle ignores the right of way	✓	✓

mentation complexity. A major drawback of such a system would be that someone has to manually segment a static scene into different regions and maybe even implement scene-specific rules. An end-to-end machine-learning-based approach doesn't need any manual intervention and may be even able to learn non-static perspectives while also generating "rules" for performing anomaly detection automatically. There are good points for using either one of the approaches mentioned. A system in which both approaches are combined would be possible as well. This thesis focuses solely on uncategorized anomaly detection techniques. If one would be interested in differentiating between different anomaly categories (like the ones in table 1.3.1) a supervised machine learning approach would be more fitting.

1.3.2 Methodology Summary

To find a working model architecture, strong inspiration is taken from various other scientific publications. The found architecture candidates are then implemented (see chapter 6). To ensure that the proposed artificial neural network architectures are implemented correctly, firstly training and evaluation on datasets that are used in literature are performed. As soon as it is ensured that the implementations are correct, progress can be made to examine whether the architecture is suited for the vehicular surveillance camera domain. To do so, first, an evaluation on the publicly available vehicular traffic anomaly detection dataset *Street Scene* (see 4.1.3) is performed. The primary motivation behind this is to have a baseline for other anomaly detection approaches that share the same domain. Lastly, each proposed architecture is evaluated on the, *FKK* dataset (see 4.2) that was introduced in the context of this thesis. Therefore, the resulting artifacts of this thesis are

1. correct implementations that are accessible as part of the video anomaly detection framework

2. objective evaluation between the implemented architectures based upon different use-cases/datasets

3. a *FKK* model candidate that is sufficient for the requirements set by *e:fs*

1.4 Contributions to the Scientific Community

As previously stated *anomaly detection* is a widely established field in the machine learning research community. Especially the automotive space is highly interested in identifying anomalous events. Potential use cases include:

- Anomalous sound/vibration signatures in engines (Malhotra et al., 2016)
- Vehicular anomaly detection based on ECU data (Narayanan, Mittal, and Joshi, 2016)
- Trajectory anomaly detection (Fu, Hu, and Tan, 2005)

Many of these publications work on information sparse data compared to video data. Therefore, an abstraction layer is often not necessary. Video data, on the other hand, is much more information dense and a well-performing abstraction layer is key so that the anomaly detection system is even able to work properly. This of course adds another layer to the already complex anomaly detection problem and therefore increases the complexity of the machine learning model as well.

1.4.1 Anomaly Detection on Vehicular Traffic Surveillance Camera Footage

There is already a couple of recent research works that tackle the surveillance camera anomaly detection problem. However, most are primarily targeted toward the highway surveillance video domain (Zhao et al., 2021; Wu et al., 2021b; Li et al., 2020a; Aboah et al., 2021). This is problematic because the highway setting is drastically different from the urban setting. Not only are there a greater variety of different types of actors in an urban setting, but most publications make the assumption, that if an anomaly occurs the car will eventually come to a full stop. (Zhao et al., 2021) An assumption like this simplifies the underlying problem because a background modeling algorithm (Adi et al., 2018) in combination with a vehicle/object detector is mostly sufficient for solving the anomaly detection problem. Making the assumption may be reasonable for the highway environment but in urban scenarios, actors come to a full stop regularly. Especially for traffic surveillance cameras pointed toward a traffic intersection. There are some, but few approaches that explore a more urban environment (Pourreza, Salehi, and Sabokrou, 2021). Background modeling algorithms are barely used in this setting because the underlying actor behavior is much more complex than the one typically encountered in the high-way scenario.

1.4.2 Video Anomaly Detection

Most anomaly detection systems that are applied to video footage, work by using a type of autoencoder that leverages convolutions as abstraction layers. Anomalies that these systems can identify are often of class:

- unknown object present in a scene
- fast movement present in a scene

Both of these anomaly types don't necessarily require a high-level understanding of a given scene. For this thesis identifying only these types of anomalies is not sufficient. Vehicular anomalies often require a deep contextual understanding of spatial and temporal information. For example, a car at a certain position could be perfectly fine for a point in time T but an anomaly for a future time $T + 1$. A more in-depth explanation of why certain model architectures were chosen as candidates can be found in chapter 6.

CHAPTER 2

Development Environment

2.1 Dependency and Virtual Environment Management

Python heavily relies on its package manager `pip`. `pip` allows users to easily install packages via a usual package manager `cli`. Unfortunately `pip` struggles with resolving dependencies in a conflict-avoiding way. This can result in installed libraries not behaving properly, often without emitting any kind of warning or error. There are a handful of wrapping libraries that abstract the `pip` API, to allow the user to automatically install dependencies for them into a virtual environment (like `pipenv`). A recent promising project called *Poetry*[1] tries to solve the issue of resolving project dependencies and making it easy to create reproducible builds. This is done via a `pyproject.toml` file in which all the information, like dependencies, development dependencies and various metadata values are stored. If an object in the dependency tree changes, the resolver finds a set of installation candidates that fulfill version restrictions and on success, creates a `.lock` file in which all dependency package versions are fixed. This way a second developer can exactly reproduce the versions of all dependencies with a single command (`poetry install`). If one wants to package the project into a distributable *Python* package, simply invoking `poetry build` suffices.

In hindsight, the choice to use *Poetry* as the package management solution was not optimal compared to frameworks like *anaconda*. This is mainly caused by not needing/using features like package building or publishing. Instead, an abstraction over *CUDA* would have been more useful.

2.2 Choice of Machine Learning Framework

The most popular and widely used machine learning frameworks occupying the Python domain, are *PyTorch*, *TensorFlow* and *SciKit-Learn*. Because this thesis mainly works

[1]`https://python-poetry.org`

with artificial neural networks and *SciKit-Learn* provides mainly classical machine learning algorithms, the choice lies primarily between *Tensorflow* and *PyTorch*. Both frameworks provide an ecosystem of additional tools provided either by the community or by the vendors themselves. For this thesis *PyTorch* is chosen, based on the more *pythonic*[2] API compared to the on *Theano* based library *TensorFlow*. *PyTorch* allows users to perform complex mathematical operations through a functional interface that is automatically differentiable, a key attribute for artificial neural networks. Additionally, wrapper libraries like *PyTorch-Lightning* (also mentioned in chapter 5.5 & 2.7) further abstract logic like data loading and distributed training. Handling and manipulating video data becomes also very easy with specialized libraries like `torchvision`. An alternative machine learning ecosystem that focuses on artificial neural networks is the previously mentioned *Tensorflow* framework. Because some prior researched architecture implementations were already implemented for the *PyTorch* framework by either the community or already included in the framework but missing from *Tensorflow*, the choice ultimately fell on *PyTorch*. Potential selling points of *Tensorflow* like the potential to rely on the Google Cloud and its TPU[3] hardware infrastructure didn't affect the choice in any way because there were no plans to use the Google Cloud for training.

2.3 Test-Suite

To ensure that all the implemented components found in this project are working as intended, a *PyTest* test suite is introduced. The test suite contains unit tests and integration tests and can be invoked with the `pytest` command. Statement test coverage reports are automatically generated and can either be viewed as an exported `.html` document. To guarantee a high code quality throughout the main line[4] of the project, automatic testing via a CI[5] environment is employed. Not only does the CI pipeline reject commits that fail tests but it also submits a coverage report to *coverage.io*. There the coverage report gets analyzed and users can view a nice visualization depicted in figure 2.1.

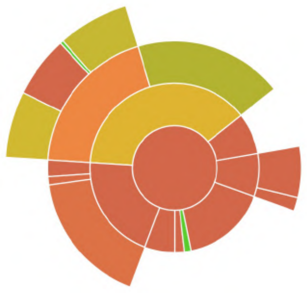


Figure 2.1: Visual coverage report.

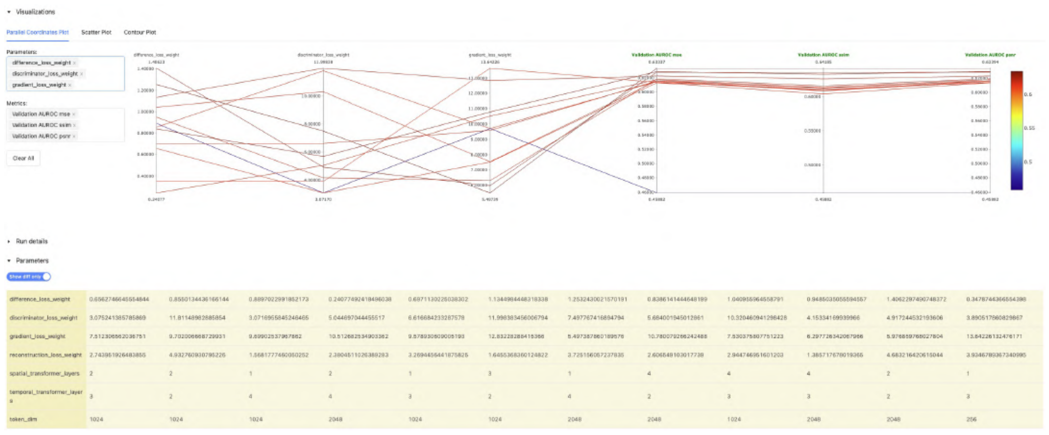
2.4 Visualizing Training Analytics through Tensorboard

Visualization of training progression and intermediate training artifacts is done with the from *TensorFlow* originating *Tensorboard* analysis tool. *PyTorch* supports storing training logs and artifacts in *Tensorboard* natively. *Tensorboard* provides a dynamic

[2]*Pythonic* describes an idiomatic style of programming found in Python.
[3]*abbr.* for Tensor Processing Unit
[4]In this context main line refers to the git branches `main` and `develop`.
[5]*abbr.* for *continuous integration*

view of various loss functions during training and is also able to display video information. This allows for fast debugging and experiment iteration cycles.

2.5 ML-Flow



difference_loss_weight	0.6562746645554844	0.8550134436166144	0.8897022991852173	0.2407749241849603 8	0.6971130226038302	1.1344984448318338	1.2532430021570191	0.8386141444648189	1.0409559645587 91	0.9485035055594557	1.4062297490748372	0.3478744366554398
discriminator_loss_weight	3.075241385785869	11.81148982885854	3.0716955845246465	5.044697044455517	6.616684233287578	11.998383456006794	7.497767416894794	5.684001945012861	10.320460941298428	4.1533416993996 6	4.917244532193606	3.890517860829867
gradient_loss_weight	7.512306562036751	9.702006668729931	9.69902537967862	10.512682534903362	9.578930609005193	12.832282884153 66	5.497387860189578	10.780079286242488	7.530375807751223	6.297728342067966	5.976859768027804	13.842261324761 71
reconstruction_loss_weight	2.743861926483865	4.932760930795226	1.5681777460050252	2.380451102638928 3	3.269445644187582 5	1.6455368360124822	3.725156057237835	2.606648103017738	2.944746951601203	1.385717678019365	4.683216420615044	3.9346789367340985
spatial_transformer_layers	2	2	1	2	1	3	1	4	4	4	2	1
temporal_transformer_layers	3	2	4	4	3	2	4	2	3	3	2	3
token_dim	1024	1024	1024	2048	1024	1024	2048	2048	1024	2048	2048	256

Figure 2.2: ML-Flow training job comparison of a hyper-parameter optimization job.

Even though Tensorboard is a great tool for visualization artifacts and training progression, managing a significant history of different types of training runs is not easily doable. Especially for hyperparameter tuning jobs, in which you typically have a large number of training jobs, comparing these efficiently to identify the most optimal model is crucial. This white spot is filled with the use of *Ml-Flow*. Among other things, *Ml-Flow* provides a web interface in which training runs can be grouped, compared and filtered. Logging to *Ml-Flow* can be done with a logger class that is provided by the `mlflow` *Python* package and is wrapped by *PyTorch-Lightning* to automatically log to *Tensorboard* and *Ml-Flow* simultaneously. Installation of *Tensorboard* and *Ml-Flow* is automatically handled by *Poetry* (chapter 2.1). Accessing the web interface can be done with either the `tensoboard --logdir logs` or `mlflow ui` command, based on the desired framework.

2.6 Python Documentation

To provide developers with documentation on how to use various components implemented in the context of this thesis, a HTML documentation is provided. An excerpt of this documentation can be found in appendix B. To keep the documentation up-to-date continuously, the documentation gets partly automatically generated and therefore can be updated automatically via various continuous integration tools and platforms. *Sphinx*[6], a tool specialized for generating documentation by parsing *Python DocStrings* is used as a generation tool. Compared to alternatives like *pydoc*[7], *Sphinx* supports more sophisticated parsing techniques like for example documentation inheritance. The documentation gets automatically regenerated by *GitHub*

[6]`https://www.sphinx-doc.org/en/master/index.html`
[7]`https://docs.python.org/3/library/pydoc.html`

Actions workflows as soon as a code base change is detected on the `main` branch of the project. An excerpt of the generated HTML documentation for a single module is provided in appendix B. Guides on how to use the project's components, for example how to train an implemented architecture on custom datasets are also provided in this documentation.

2.7 Training

Most of the later introduced neural network architectures require non-trivial training procedures. Because of the data throughput and processing power required by the machine learning models, training on a CPU is unfeasible. Even though training on a single GPU offers significant training time improvement, being able to distribute the training process between multiple GPUs becomes key to efficiently develop and evaluate model variants. There is a wide variety of different techniques to distribute training between different processing units. The implementations are non-trivial and that's why this thesis relies on external packages to handle different distribution techniques. More specifically the PyTorch Lightning[8] framework, a wrapper module over PyTorch that seamlessly integrates into the PyTorch eco-system. PyTorch Lightning offers *DP* (Data Parallel) and *DDP* (Distributed Data Parallel) training distribution strategies. *Data Parallel* splits a single batch into one for each GPU and then distributed model and input data to each one. After each GPU computes the forward and backward pass, the weights are sent back to the main process and averaged. The main process then computes the weights update and a single training cycle has finished. Because this method sends a lot of data between GPUs and the main process for each training step (model weights, training data and weight gradients) significant overhead is introduced. Figure 2.3 visualizes the *DP* training loop. *DDP* introduces

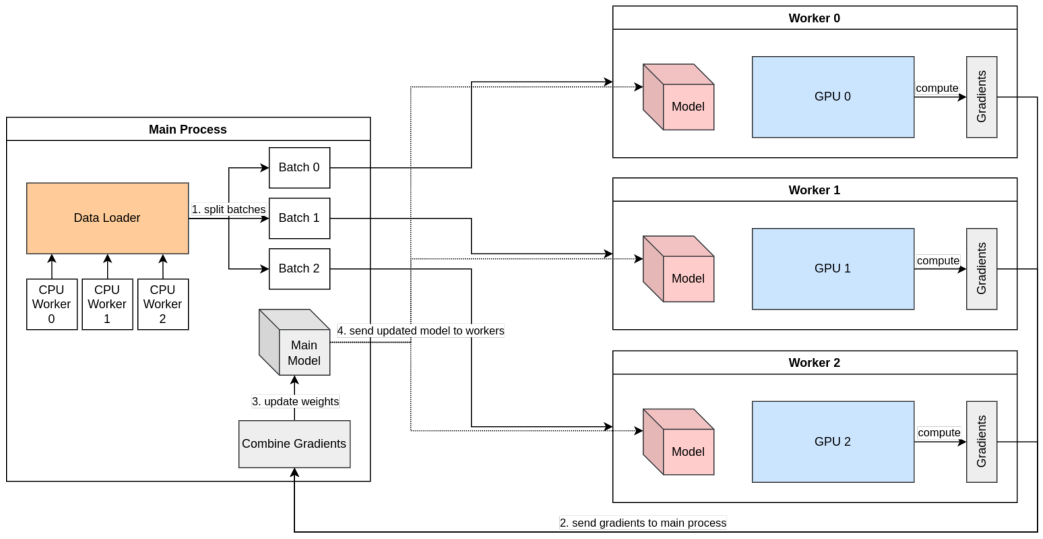


Figure 2.3: Visualization of the *Data Parallel* distribution strategy.

a method that reduces overhead significantly by reducing the amount of transmitted

[8]`https://www.pytorchlightning.ai`

data. Figure 2.4 shows how *DDP* synchronizes all the computed gradients across GPUs and then performs the weight update on each process individually.

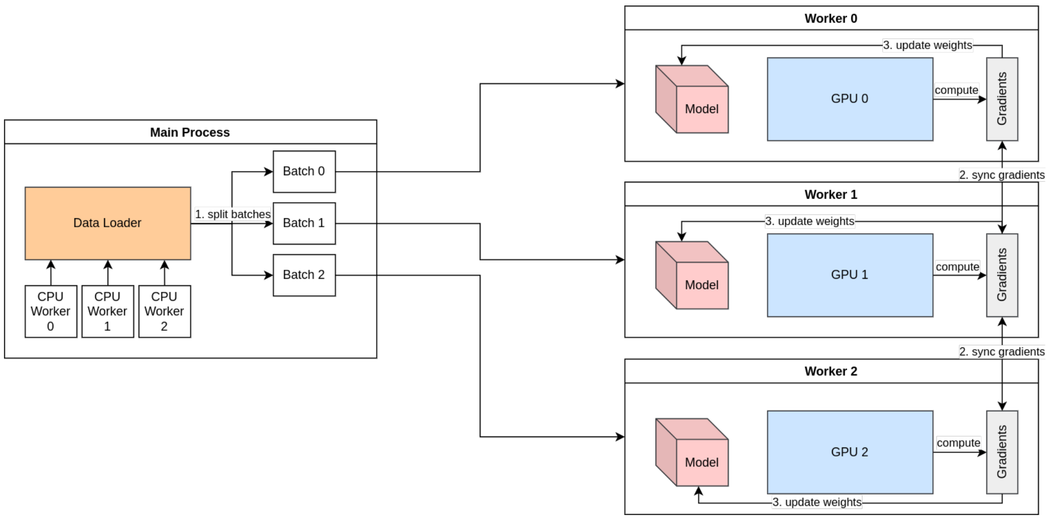


Figure 2.4: Visualization of the *Distributed Data-Parallel* distribution strategy.

2.8 Hyper-Parameter Tuning

Similar to model training (chapter 2.7), Most models introduced in this thesis have multiple non-trivial hyperparameters. Examples would be loss function weights, input timesteps, frame strides, *etc.*. Finding a good parameter configuration for a dataset and model architecture manually is unfeasible because of the enormous search space. Therefore, an automated hyperparameter tuning approach is required. The *ray*[9] machine learning framework integrates nicely into the *PyTorch/PyTorch-Lightning* ecosystem and implements state-of-the-art hyperparameter optimization techniques in the context of the *ray-tune*[10] submodule. Hyperparameter optimization runs can even be distributed across multiple GPUs, which leads to significantly faster iteration times. *Ray* explores the hyperparameter search space by primarily either performing a grid or random search. Publications have shown, empirically and theoretically that random search is more efficient for most hyperparameter optimization problems compared to manual or grid search (Bergstra and Bengio, 2012). Grid search performs an exhaustive search over a well-defined search space. Random search on the other hand randomly samples hyperparameter configurations out of a previously defined distribution. Hyperparameter optimization is performed after finding a non-optimized but empirically working model architecture. Ray will generate a new hyperparameter configuration for every training run and train every model variant for a fixed number of epochs. After a previously set, time or model variant limit, *ray* will compare every trained model by evaluating it based on the *AUROC* score (Bradley, 1997) achieved over the *test* dataset. Hyperparameter configurations can be found in chapter 8.

[9]`https://docs.ray.io/en/latest/index.html`
[10]`https://docs.ray.io/en/latest/tune/index.html`

CHAPTER 3

Theoretical Background

As stated earlier the goal of this thesis, is to create a system that can assess the regularity of a recorded scene. Therefore, the regularity function r will map a frame sequence to a probability variable.

$$r : \mathbb{R}^{T \times C \times H \times W} \to \mathbb{R}$$

Some approaches may even be able to not only asses the regularity of the entire scene but each timestep $t \in T$ in the scene. The regularity function then maps to T probabilities for each timestep in the input sequence.

$$r : \mathbb{R}^{T \times C \times H \times W} \to \mathbb{R}^T$$

To derive the anomaly score we can use a mapping $a : \mathbb{R} \to \mathbb{R}$ from regularity score to anomaly score defined as $a(x) = 1 - r(x)$. Because this transformation is trivial, it doesn't matter if an approach tries to detect regularity or anomalies.

> Anomaly detection, a.k.a. outlier detection or novelty detection, is referred to as the process of detecting data instances that significantly deviate from the majority of data instances. (Pang et al., 2021)

3.1 Anomaly Detection Introduction

Anomaly detection algorithms can be differentiated between deep learning based anomaly detection, sometimes also referred to as *deep anomaly detection* and classical anomaly detection techniques. In this thesis, we focus on *deep anomaly detection.* There are a variety of different neural network techniques potentially suited for the anomaly detection problem set. Examples include:

- Autoencoders
- Generative adversarial networks (GANs)

- Softmax likelihood models
- ...

Pang et al. (2021) and Chandola, Banerjee, and Kumar (2009a) differentiates between three different types of anomalies.

1. *Point anomalies*, are individual data points that individually differ from other data points. For the vehicular surveillance camera domain, a typical *point anomaly* would be an anomalous object in a captured image. The anomalous object is considered an anomaly independent of the overall context. Detection of *point anomalies* doesn't even require temporal context and therefore anomaly detection can be performed on individual frames.

2. *Conditional anomalies* are data points that are only considered anomalous if a set of conditions is fulfilled, that's why *conditional anomalies* are sometimes also referred to as *contextual anomalies.* This anomaly type is highly relevant for this thesis because many traffic anomalies are dependent on the context. For example, a car passing under a red traffic light is only anomalous because the traffic light is red.

3. *Group anomalies* are data points that are on their own not anomalous, but as soon as they form a distinct group they become anomalous. For example, spam mail on its own is not anomalous but most spam mails share features among them, which makes them identifiable. This anomaly category is not necessarily a type that is encountered with traffic surveillance footage. An example of *group anomalies* that share the traffic domain would be traffic congestion anomalies (Chen et al., 2019).

For this thesis especially the type *point* and *conditional anomaly* type is relevant.

3.2 Related Scientific Work

Anomaly detection isn't a novel problem that computer scientists and more specifically the machine learning community are trying to solve. However, it seems like it may not get as much attention as other similar research fields. This may originate from unsupervised anomaly detection being an overall hard problem to solve because most underlying data often requires unsupervised or self-supervised machine learning techniques. The lack of labeled training data present in many anomaly detection use cases often eliminates supervised approaches. Annotations are often behavior types and are therefore used with behavior classification systems. With the upcoming interest in anomaly detection by industry sectors like the automotive industry, anomaly detection gained increased popularity. The ability to detect anomalous and therefore often interesting events enables other machine learning systems to be trained on higher-quality datasets. Anomaly detection can partially be seen as a gateway system to improved datasets for other already existing systems.

Generally, anomaly detection systems try to learn a probability distribution P' by training on a dataset containing sampled data points from a regular distribution

P. Irregular data points are not learned by the system and therefore the system is unable to behave correctly. This incorrect behavior can be often trivially detected by observing the behavior of the system compared to the real world. For example, an anomaly detection system that is trained to predict car movement can predict regular movements accurately but as soon as a car moves irregularly the prediction will not be accurate anymore. In other words, the behavior of the real car is significantly different from the behavior modeled by the system and consequently, this difference can be used to infer whether the behavior that the car exhibits is anomalous or not.

Learning a probability distribution is significantly easier if the distribution's dimensional complexity is low (Chandola, Banerjee, and Kumar, 2009b). Because video data is very informationally dense and contains spatial and temporal relations across feature dimensions, the video anomaly detection problem is especially challenging. In the past, a few publications were made towards solving the video anomaly detection problem specifically, but because limited anomaly detection datasets are publicly available not a wide variety of different use cases has been explored thoroughly yet. There primarily use cases/datasets the science community is working on are:

- pedestrian behavior anomalies (including foreign objects as anomalies) (Yuan et al., 2021; Hasan et al., 2016)
- vehicular anomalies from an egocentric perspective (Haresh et al., 2020)
- (highway) traffic surveillance footage anomaly (Zhao et al., 2021; Wu et al., 2021b; Li et al., 2020a; Aboah et al., 2021)
- rural traffic surveillance footage anomaly (Pourreza, Salehi, and Sabokrou, 2021; Salas et al., 2007)

Of course, some publications work on other problems but most of the significant publications evaluate their approach on some of these use cases (and corresponding datasets). More information about the different datasets available to evaluate anomaly detection systems on, are described in the following chapter 4.

Even though multiple approaches were proposed by various publications over the last years (Arnab et al., 2021; Haresh et al., 2020; Park et al., 2021), there is a tangible lack of high-quality neural network architecture implementations targeted toward anomaly detection and more specifically towards the video medium. Some sparse implementations are available as open-source repositories, but are often published in conjunction with publications and are not necessarily designed for adaption to new use cases. Implemented architectures that are publicly available are often very basic and can't be described as state-of-the-art. Some notable implementations are:

- `https://github.com/hashemsellat/video-anomaly-detection`
- `https://github.com/aseuteurideu/STEAL`

The lack of general video anomaly detection frameworks is a clear identified white spot in the machine learning community and the resulting code from this thesis attempts to fill this blank with neural network architecture implementations that are:

- constructed out of modular and reusable components

- easy to use and should require minimal to no code to get acceptable results
- conform to the usual *PyTorch* APIs

3.3 Architectures

In this chapter some theoretical concepts on neural network layer architectures are elaborated on that will be later used in chapter 6.

3.3.1 Image Convolutions

The convolution operation (Fukushima, 1980) is one of the most important elements of today's computer vision techniques. Convolutional neural networks can learn spatial features very efficiently. The effectiveness of the convolution operations led to an almost complete displacement of traditional computer vision algorithms for tasks like object detection or segmentation and many others (Li et al., 2020b). A convolution operation extracts higher-level spatial features from the input image. It does so by multiplying an extract from the image $I_{i,j} \in \mathbb{R}^{k \times k}$ with a weight matrix of the same size $W \in \mathbb{R}^{k \times k}$ and calculating the average value of the resulting 3×3 matrix. This procedure is repeated as a sliding window procedure over the input image. The resulting averaged values will make up the generated feature map. To extract multiple features simultaneously, multiple filters can be used. Consequently, the convolution operation will result in multiple output feature maps. Figure 3.1 shows an example of how a single feature map pixel gets generated. Stacking convolution operations

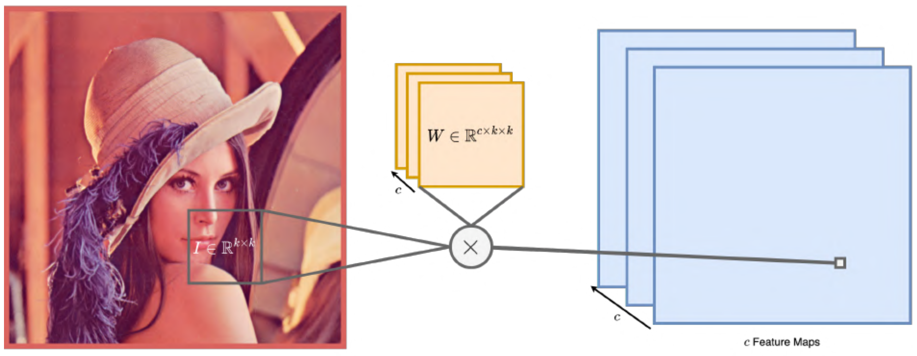

$W \in \mathbb{R}^{c \times k \times k}$
c
$I \in \mathbb{R}^{k \times k}$
c
c Feature Maps

Figure 3.1: A visualization of how a convolution on a single window results in a feature map pixel.

on top of each other enables the system to learn even higher-level features. This, for example, allows the network to differentiate between different object types. Because the convolution operation only needs to store weights for processing a single extract from the image it is quite efficient.

3.3.2 Autoencoder Architecture

Most approaches for anomaly detection try to learn a probability distribution derived from mostly regular data. With this learned knowledge about regularity, non-regular data points can be identified effectively. The general autoencoder architecture has historically proven to be a very effective and adaptable tool for learning essential features about low-level data distributions. They do so, by trying to compress data points taken from the input distribution into a much smaller representation also called *latent-space*. This transformation is typically performed by the *Encoder* E module. To generate gradients for the solver, a *Decoder* module D is used to transform the *latent-space* representation of a data point $\hat{x} = E(x)$ into the original input space. Therefore, the *Encoder* will try to store the most useful information into a latent vector by that the *Decoder* can construct the initial *Encoder* input. Assuming all components are trained perfectly, the output of D should be equal to the input to the E $x = D(\hat{x})$. In summary, autoencoders try to minimize the distance d between reconstruction and input.

$$\min_{d} \; D(E(x)) + d$$

Because the design is very unspecific, many other layer architectures can be used in conjunction with the autoencoder design. For image synthesis tasks, convolutional models like U-Net (Ronneberger, Fischer, and Brox, 2015), that follow the autoencoder design have achieved remarkable results at learning high-level information about visual data distributions.

3.3.3 Recurrent Neural Networks with Convolutional Long Short-Term Memory

The RNN[1] (Rumelhart, Hinton, and Williams, 1986) neural network architecture has proven to be very efficient in processing streams of data of variable length. An input

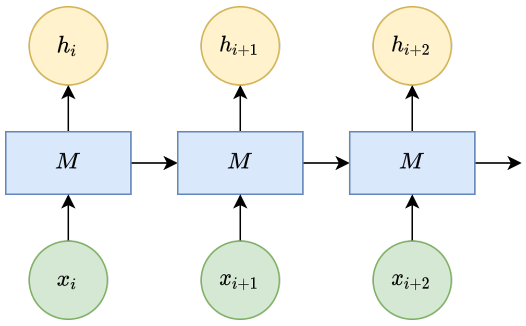


Figure 3.2: Unrolled visualization of an RNN module M processing a series of 3 data points.

to an RNN is a series of potentially variable length made out of individual fixed-sized data points x_i. Each data point x_i is then passed into the RNN module M which contains a hidden state h_i. The hidden change is manipulated with every new data point passed into the RNN module and therefore contains information about past data points processed by the module. Figure 3.2 shows an unrolled[2] visualization of a RNN processing a series of 3 data points x_i, x_{i+1}, x_{x+2}. RNNs only store weights

[1] *abbr.* for *Recurrent Neural Network*

[2] An unrolled representation of a cyclic neural network has all cycles removed by visualizing every module at each step separately.

to process a single data point at a time and simultaneously can compute a series of data points that are dependent on each other. This makes them remarkably efficient for certain domains like the NLP[3] field. Unfortunately, RNNs also can suffer from computational stability issues like the exploding/vanishing gradients problem. That's why an improved RNN sub-type, the *LSTM* architecture (Hochreiter and Schmidhuber, 1997) was proposed. *LSTMs* contain a forget gate, which allows gradients to flow unchanged through a layer. This doesn't necessarily affect the exploding gradients problem but can help to avoid vanishing gradients. Convolutional *LSTM* (Shi et al., 2015) networks are an adaptation of the original *LSTM* architecture for spatio-temporal data. Not necessarily does spatio-temporal data only refer to video data as shown by the original publication. Nevertheless, the convolutional *LSTM* architecture is highly suited for processing frame-series (i.e. video) data with the efficiency of a typical *LSTM* architecture. A convolutional *LSTM* module replaces ordinary matrix multiplication operations of a regular *LSTM* layer with convolution operations. The *LSTM* architecture can easily be adapted to follow the autoencoder design previously explained in chapter 3.3.2. Different from the usual *RNN* use-case, *RNN* autoencoders don't predict a future state, but instead try to reconstruct the input series by processing the latent space with various temporal recurrent modules (Malhotra et al., 2016; Luo, Liu, and Gao, 2017).

3.3.4 Transformer Architecture & Attention Mechanism

In recent years the attention mechanism (Vaswani et al., 2017) and the related *Transformer* architecture revolutionized some machine learning research fields. Especially with *NLP*[3] *Transformer* networks like *GPT-2* (Radford et al., 2019) and *GPT-3* (Brown et al., 2020) achieved remarkable results in text generation and text transformation tasks. Older architectures that were based on *RNN* architectures like *LSTM* or *GRU*[4] were surpassed significantly by *Transformer* networks. Since then the *Transformer* architecture was adapted to other machine learning domains and proved itself to be a viable new approach to many problem spaces (Arnab et al., 2021; Gong, Chung, and Glass, 2021). The *Transformer* architecture and *attention* mechanism try to solve a major drawback of *RNN* architecture types. As explained in chapter 6.3, RNNs operate on data sequences and information from past inputs is stored in a memory (hidden) cell h. Because all past inputs are compressed into a single shared cell, information from past inputs is diluted by newer inputs. Figure 3.3 visualizes how the information of an input contained in the hidden cell h gets diminished over time. This makes it more challenging for the architecture to learn temporal interactions over a long temporal distance. The *Transformer* architecture doesn't suffer from this problem and can learn long-distance temporal relations efficiently and effectively. The *attention* mechanism is one of the most important *Transformer* components that warrant the previously mentioned efficiency. An attention function is mapping a query Q and key-value pairs, K and V to an output (Vaswani et al., 2017). Q and K are used to compute weights for a weighted summation of V (details in 3.4) Some architecture details differ depending on which attention function is used. Popular choices

[3] *abbr.* for *Natural Language Processing*
[4] *abbr.* for *Gated Recurrent Unit*

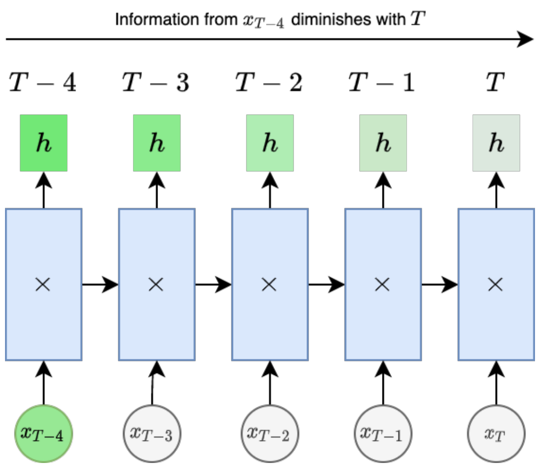


Figure 3.3: Information at $T-4$ gets passed through the network and every step involves a lossy operation that reduces the information of x_{T-4} contained in hidden state h.

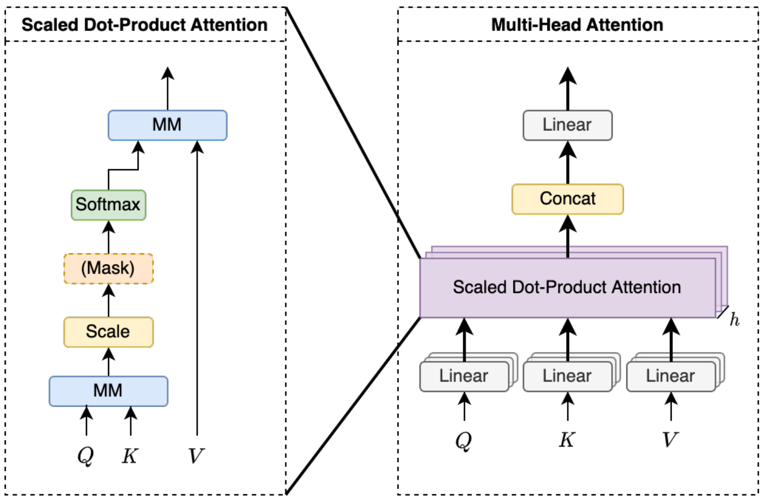
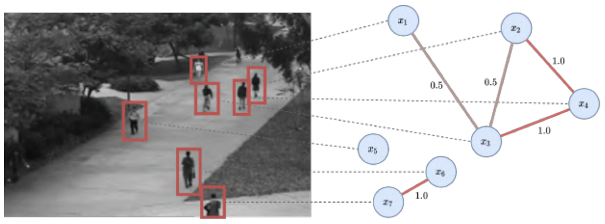


Figure 3.4: Visualization of all operations that are necessary for the implementation of either a scaled-dot-product or multi-head attention block. (Vaswani et al., 2017).

are *Scaled Dot-Product Attention* and *Multi-Head Attention* (Vaswani et al., 2017). The choice of which *attention* variant one chooses is based on a trade-off between runtime performance and theoretically learnable complexity. As depicted in figure 3.4, is the *Multi-Head Attention* module made out of multiple *Scaled Dot-Product Attention* modules connected in parallel. Every *Scaled Dot-Product's* input is additionally passed through a linear projection layer. This enables the module to learn different relations simultaneously.

3.3.5 Graph NN & Graph Convolution



$$\Rightarrow G_{a\times a} = \begin{bmatrix} 0 & 0 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 1.0 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0 & 1.0 & 0 & 0 & 0 \\ 0 & 1.0 & 1.0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1.0 \\ 0 & 0 & 0 & 0 & 0 & 1.0 & 0 \end{bmatrix}$$

Figure 3.5: Graph construction example and spatial connections between actors.

The interaction between arbitrary actors can be modeled as a graph data structure. Graph neural networks achieve great results on various use-cases (Wu et al., 2020; Chen et al., 2019). Publications that apply graph neural networks to the video domain include

- "Graph convolutional label noise cleaner: Train a plug-and-play action classifier for anomaly detection" (Zhong et al., 2019)
- "Graph neural network (GNN) in image and video understanding using deep learning for computer vision applications" (Pradhyumna, Shreya, et al., 2021)
- "Zero-shot video object segmentation via attentive graph neural networks" (Wang et al., 2019)
- "Uncertain graph neural networks for facial action unit detection" (Song et al., 2021)

A significant benefit of using a graph data structure as an input to the autoencoder is the reduced complexity compared to raw frames. Because anomaly detection on lower complexity data distributions is more effective, graphs could be well suited for such a problem. Typically each actor in a scene is transformed into a node V in a graph G. Directed or undirected, potentially weighted edges E in the graph, model interactions between various actors/nodes in a scene. Edges can not only model typical interactions but also temporal and spatial affiliation between actors. For example, spatial distance can be modeled as weighted undirected connections between nodes. To demonstrate how efficiently a graph can model a single image and part of its content let's construct a graph from an image with 7 actors depicted in figure 3.5. The graph is an undirected adjacency graph of shape $\mathbb{R}^{N\times N}$ with N being the number of actors in the scene. If a distance threshold is used, some connections can be set to 0 and therefore the resulting graph is rather sparse. The resulting adjacency matrix is also displayed in 3.5. To reduce the memory footprint of the graph even further, a sparse representation of shape $\mathbb{R}^{V\times 2}$ can be used to compress the graph to solely its connection between nodes.

The example, the graph G in figure 3.5 would result in a compressed matrix of shape $\mathbb{R}^{5\times 2}$. Operations on graphs differ greatly from regular matrix operations typically encountered with artificial neural networks. This is because the data structures the operations act upon are so different compared to regular neural networks. One popular operation is the so-called *graph convolution* operation, an adapted variation of regular convolution. The graph convolution operation is not inherently different from a conventional convolution typically encountered in computer vision applications. An input image can also be seen as a graph of connected pixels and a graph convolution will yield the same return as a 2d image convolution. Convolution can be described as an operation that shares information across points of a subset in the input space. For example, a kernel size of 3×3 will update the point value as a weighted sum of all 8 adjacent points and the point itself (see chapter 3.3.1). The selection of points in a regular convolution is spatially restricted by the distance towards the center point and the chosen kernel size. Graph convolutions remove this spatial restriction altogether. As a result points of arbitrary spatial distance can share information. Because of the implicated complexity, sharing information amongst all points in the input space is unfeasible and a new restriction needs to be introduced. Each point has a list of other points it is semantically connected to and objects update their information by a weighted sum of connected points and itself. Figure 3.6 depicts the graph convolution operation applied around the node x_0. Repeating the graph convolution by propagating the graph through multiple layers will spread the information of each node x_i further across a connected cluster of a graph.

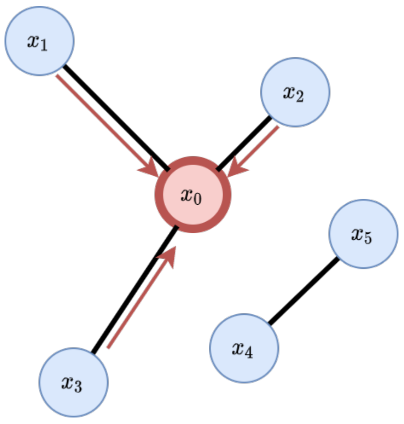


Figure 3.6: A single graph convolution operation around a node x_0. This convolution is repeated around every node in the graph.

CHAPTER 4

Datasets & Data Acquisition

4.1 Publicly Available Datasets

For the evaluation of the neural network architectures publicly available anomaly detection datasets are used. Because these datasets are widely used by other scientific literature, the results from other publications are used as a baseline for our anomaly detection systems. As soon as comparable performance to other state-of-the-art results is achieved, the advancement toward training and evaluating the model on the *e:fs FKK* dataset (chapter 4.2) can be made.

Name	Source
UCSD Pedestrian 1 & 2	`http://www.svcl.ucsd.edu/projects/anomaly/dataset.html`
Avenue Dataset	`http://www.cse.cuhk.edu.hk/leojia/projects/detectabnormal/dataset.html`
AIC21 Track 4	`https://www.aicitychallenge.org`
Street Scene	`https://www.merl.com/demos/video-anomaly-detection`

All datasets are available as *PyTorch Lighting Data Modules* (see chapter 5.5), therefore it's not necessary to manually download and prepare the video data[1].

4.1.1 UCSD Pedestrian Anomaly Dataset

The *UCSD* pedestrian dataset features two different subsets of data *UCSD 1* and *UCSD 2*. Both contain monochrome video clips that feature pedestrians traversing walkways. Anomalies are defined as cyclists, skaters or cars being in the scene. These anomalies are only present in the *test* dataset of each *UCSD* subset. The *test* datasets contain video footage with anomaly annotations.

[1]datasets that require some form of authentication are excluded

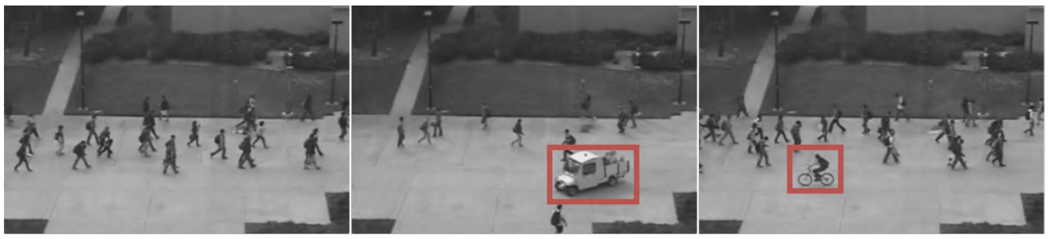


Figure 4.1: Excerpt from the *UCSD 2* dataset. Anomalies are marked in red.

Table 4.1: Length of the *UCSD Pedestrian 1 & 2* train and test dataset.

Subset	Train Length	Test Length
UCSD 1	272s (34 videos × 8s)	288s (36 videos × 8s)
UCSD 2	128s (16 videos × 8s)	96s (12 videos × 8s)

The most significant difference between both subsets, is that *UCSD 1* is significantly harder, because of the perspective distortion. Refer to figure 4.2 and figure 4.1 for the different camera perspectives. The *UCSD* dataset is used in many video

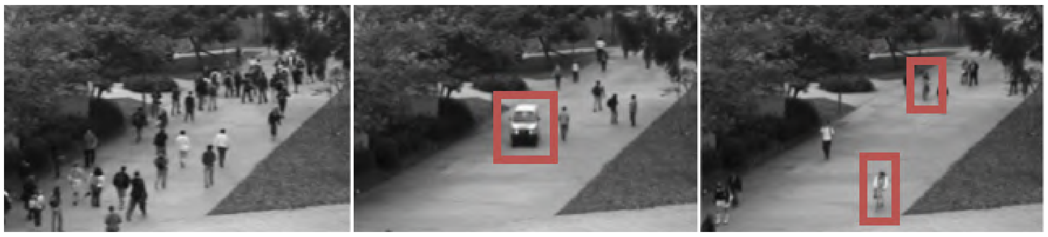


Figure 4.2: Excerpt from the *UCSD 1* dataset. Anomalies are marked in red.

anomaly detection publications as one of the most important baseline performance benchmarks. For this thesis, the dataset is especially relevant because the camera's perspective matches that of the *e:fs* surveillance camera. To make the dataset easy to access and download a `UCSD` *PyTorch Lightning* `DataModule` is provided. Refer to chapter 5.5 for implementation details and usage examples.

4.1.2 CUHK Avenue Dataset

Similar to the previously described *UCSD* dataset, *CUHK*[2] *Avenue* is also a very prominent baseline dataset in the video anomaly detection problem space. The dataset contains variable-length RGB footage taken from an eye-level perspective. Videoclips contain pedestrians walking and standing in the camera's field of view. Anomalies are people loitering and acting unusual (*i.e.* dancing or throwing). Although the camera's perspective is static, occasional micro-vibrations are present throughout the dataset. For some approaches, even small amounts of camera movement can have a significant impact on model performance. Compared to other datasets like *UCSD* the distribution between regular and irregular frames in the test dataset is heavily biased towards regular frames. Only $30,77\%$ of a total of ~ 15.000 frames in the test dataset contain

[2] *abbr.* for Chinese University of Honk Kong

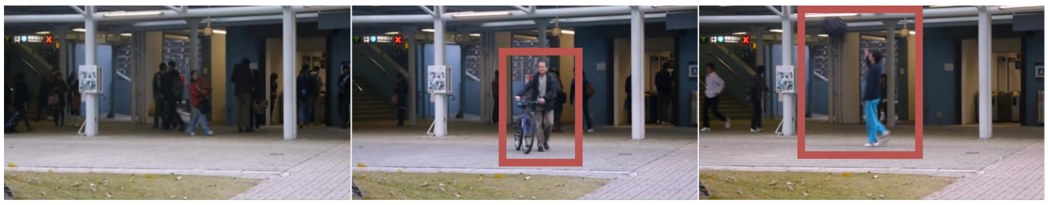


Figure 4.3: Excerpt from the *Avenue* dataset. Anomalies are marked in red.

irregular regions. This is why *Precision-Recall* is used as a performance metric with this dataset compared to the *Receiver-Operating-Characteristics* during evaluation in chapter 8.

Table 4.2: Length of the *Avenue* train and test dataset.

Dataset	Total Length
Train	~12min (16 videos)
Test	~11min (21 videos)

4.1.3 Street Scene (Ramachandra and Jones, 2020)

Although the *Street Scene* dataset is not widely used in video anomaly detection publications, it is a valuable dataset for this thesis because it shares the same domain as the *e:fs* use-case. The dataset shows camera surveillance footage from a static isometric perspective. Actors include various road vehicles, pedestrians and cyclists. Anomalies only present in the test dataset are considered vehicles parking in various ways or pedestrians/cyclists crossing the road. Anomaly annotations are present for the test dataset and are not localized. Videos are provided as individual frames and

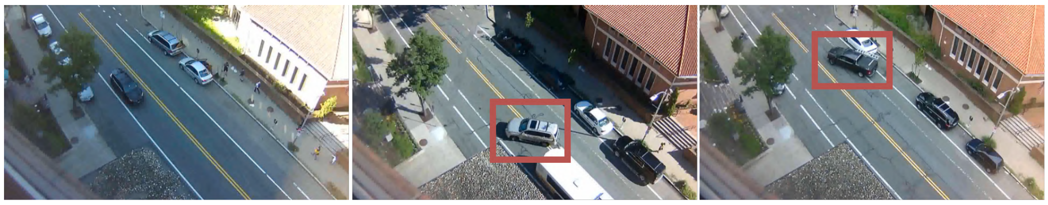


Figure 4.4: Excerpt from the *Street Scene* dataset. Anomalies are marked in red.

not as a single continuous video file. This in itself is not uncommon, but all frames are compressed as JPEG images. Concatenating all individual frames into a video reveals that the JPEG compression results in very jittery footage.

4.2 e:fs *FKK Anomaly Dataset*

In the context of this thesis, a completely new anomaly dataset called *FKK Anomaly Dataset* is introduced. The training dataset consists of variable-length 1080p video clips all taken by the *FK*[3] camera. Recorded footage features randomly selected scenes

[3]abbr. for "Forschungs-Kreuzung" (engl. Research-Intersection)

Table 4.3: Length of the *Street Scene* train and test dataset.

Dataset	Total Length
Train	∼98min (35 videos)
Test	∼38min (46 videos)

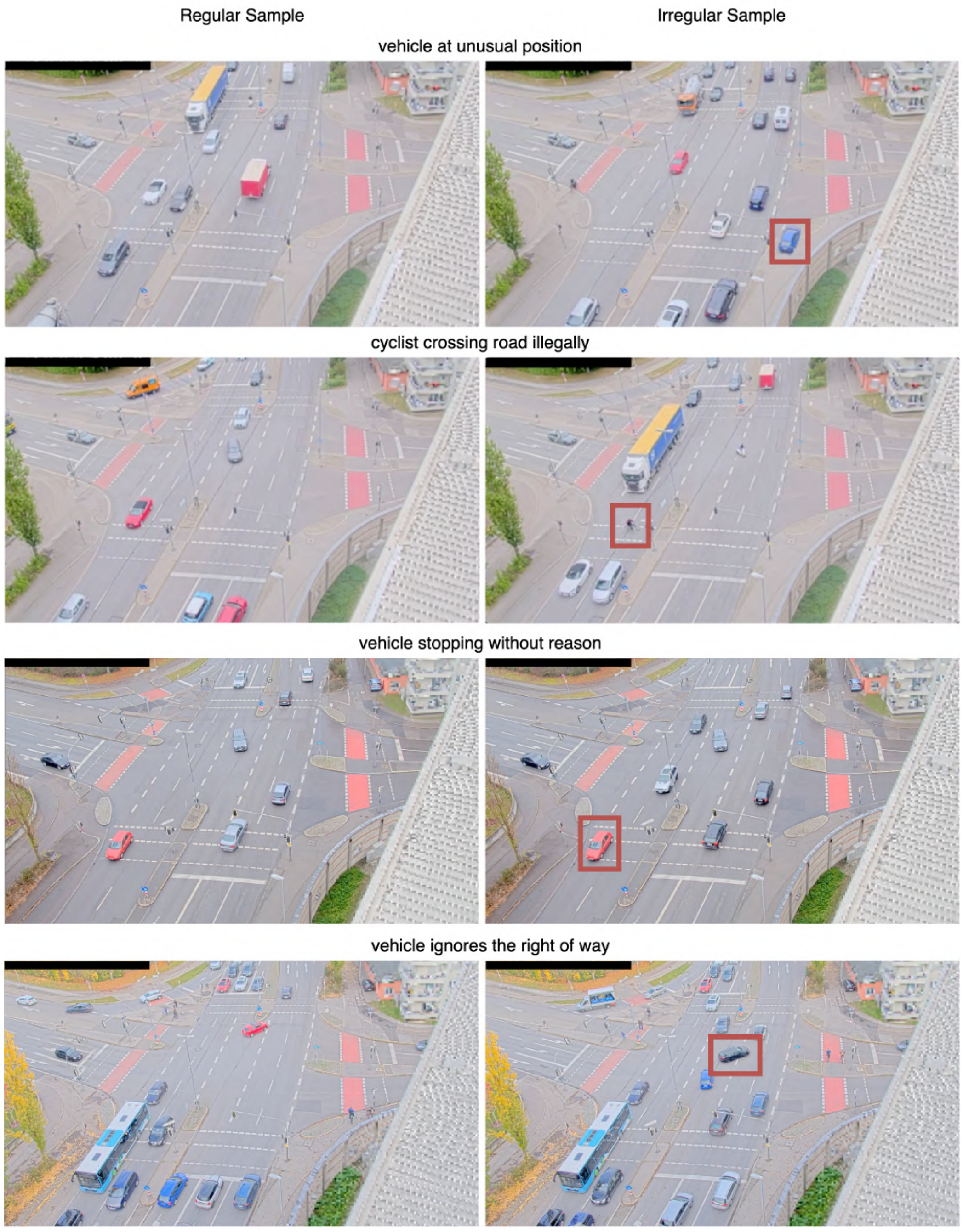

Regular Sample
Irregular Sample
vehicle at unusual position
cyclist crossing road illegally
vehicle stopping without reason
vehicle ignores the right of way

Figure 4.5: Excerpt from the *FKK Anomaly* dataset. Anomalies are marked in red.

exhibiting mainly typical actor behavior. The clips present in the training dataset are all unlabeled. There are two different variations of the dataset, one featuring only daylight scenes while the other also includes footage taken by night, dawn and sunset. Naturally, the second dataset is the more challenging for the network to learn, because it has to handle a wide variety of drastically different ambient light conditions. For real-world applications, the achieved test score on the day and night dataset is the more decisive metric for overall effectiveness. In addition to the training dataset a test dataset is provided, so models that were trained on the *FKK Anomaly Dataset* can be evaluated thoroughly. Each variable length clip of the test dataset features synthesized anomalous actors or behavior patterns. Because anomalies contained in the video segments are synthesized manually, a slight bias is introduced into the test dataset. To counteract this bias, a broad selection of different anomaly categories is included. Table 4.4 shows examples of different anomaly types and their contexts included in the test dataset. Binary and therefore not categorized frame annotations are provided with each test video segment. All actors present in the test dataset

Table 4.4: *FKK Anomaly Dataset* - Anomaly Types

Anomaly Description	Spatial Context	Temporal Context
vehicle at unusual position	✓	
vehicle stopping without reason	✓	✓
pedestrian/cyclist crossing road illegally	✓	
vehicle ignores the right of way	✓	✓

are completely distinct from the training dataset. This is to ensure that the trained model is exclusively confronted with previously unknown behavior or objects. Real[4] anomalies captured by the *e:fs FK* camera are not present in the test dataset, because they are hard to identify efficiently. Even though the number of test videos in the *e:fs FKK* dataset is not necessarily sufficient to evaluate the performance of a neural network architecture very accurately, it is sufficient for evaluating whether or not an approach is promising. It turned out that this was enough for this thesis and more test video clips wouldn't have helped much in finding a better architecture. In the future, an anomaly detection system could be used to continuously find potential anomaly candidates and build are more expressive test dataset.

Table 4.5: Length of the *e:fs FKK* train and test dataset.

Dataset	Total Length
Train	∼121min (49 videos)
Test	∼7min (5 videos)

[4]"Real" in this context means not synthesized

CHAPTER 5

Video Data Processing Pipeline

The systems proposed in this thesis are primarily designed to process video data. In combination with the high data throughput required by machine-learning algorithms, loading and processing video data becomes a non-trivial task. Using computer hardware efficiently becomes critically important, otherwise, system performance could be bottlenecked by data loading and processing. Fortunately the `torch.utils.data`[1] API provided by *PyTorch* can help tremendously with optimizing and implementing efficient data-loading pipelines. Features provided by the data API include:

- multithreaded data loading
- automatic batching of data
- integration with `torchvision.transforms`[2] modules

An additional primary factor for choosing the torch data loading framework for this thesis was that most of the code base was primarily written with *PyTorch* modules. Therefore, no additional dependencies would be introduced into the project. This also fits in nicely with the overall goal to stay as native to the *PyTorch* ecosystem as reasonably possible.

5.1 Data Loading

It's unfeasible to load all videos at once into system memory because this would exceed the available memory capacity on almost all systems. Therefore, an iterative data-loading solution is required. The implementation should also be as generic as possible to support a wide variety of different video datasets without any implementation changes. The lowest API for reading video data from a drive is the `torchvision.io.read_video` function. Internally this function either uses the Python

[1]https://pytorch.org/docs/stable/data.html
[2]https://pytorch.org/vision/stable/transforms.html

library `pyav` or `ffmpeg` for video decoding. The latter offers better performance although needing to be natively compiled for the target system. The `trafficanomalydetection` library and executables will choose `ffmpeg` if available to the system and fall back to `pyav` if not. There are different approaches for loading video files into data structures that are appropriate for training a machine learning model. This thesis makes use of the `Dataset`[3] base class provided by *PyTorch*. Consequently, this enables the use of the *PyTorch* `DataLoader` implementation, which provides multithreaded data-loading and flexible batching. The following subsections describe different implementations that are used in this thesis.

5.1.1 Windowed Video Dataset

Most machine learning algorithms that operate on time series data, require input sequences to be provided as discrete time windows. Therefore, the video image sequence needs to be separated into chunks of frames. Because the network can only observe a single frame window at a time, it must contain as much temporal information as possible. Assuming that the video clip, from which frame windows should be extracted was recorded at 25 frames per second. This results in a frame window of length 3 containing $\sim 0.12s$ of temporal information. Depending on the scene recorded, movement in these $\sim 0.12s$ are often minimal and therefore lack any significant temporal information. To counteract this effect *frame strides* are introduced. The *frame stride* parameter defines how many frames are skipped while selecting a window. A visualization is shown in figure 5.1. A *frame stride* of 1 results in no *frame strides* applied at all. With a *frame stride* of 2 a single frame between each selected frame is skipped, therefore, doubling the time a single window covers. The *frame stride* parameter can be considered a hyperparameter and should be optimized to be optimal for a specific dataset/architecture. Hyperparameter tuning techniques can be utilized to do so. Similarly, the *window strides* parameter defines how many frames the window moves from one window to another. Increasing this parameter can help with avoiding overfitting the model during training.

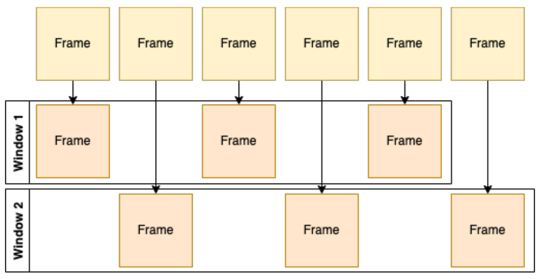


Figure 5.1: Visualization how *frame strides* and *window strides* affect the data loading process.

5.1.2 Graph Datasets

Contrary to the *Windowed Dataset* described in chapter 5.1.1 a `torch_geometric`[4] `LightningDataset` is provided for model architectures that operate on graphs as described in 6.5.1. Because graph construction is time intensive, online construction techniques are not feasible for training neural networks efficiently. The graph datasets provided in this thesis construct graph-structures upfront and store them as pickle

[3] `torch.utils.data.Dataset`
[4] `https://github.com/pyg-team/pytorch_geometric`

files in the respective dataset directory. If construction parameters change, for example, the feature extractor is replaced, all previously stored graphs get regenerated automatically. Depending on which parameters change, different preprocessing steps need to be recomputed. This, for example, allows the user to change graph generation parameters/implementations without having to run the object detector again. Dataset generation can vary greatly, depending on the number of frames present in the dataset and which feature extractor (object detector) is used. Generating the graph representation of the *UCSD 1* dataset (see 4.1.1) with a Mask *Faster RCNN-50* (Ren et al., 2015a; He et al., 2017) feature extractor backbone network takes approximately 30 minutes on a single consumer-grade *GPU*[5].

5.2 Data Processing Modules

The loaded video footage needs to be further processed before it can be passed into the input layer of a neural network. A modular architecture is used to provide as much flexibility as possible. Each preprocessing operation for example rescaling is implemented as a `torch.nn.Module`. This allows these modules to be used in conjunction with the `torchvision.transforms.Compose` class. A demonstration of how easy and maintainable a collection of preprocessing operations is shown below:

```
preprocessor = torchvision.transforms.Compose([
    Rescale(256, 256),
    Grayscale(),
    TransformChannels("channels_first")
])

# input should be of shape [T, W, H, C] or [T, C, W, H]
preprocessed_footage = preprocessor(input_tensor)
```

The `torchvision`[6] python package already implements some preprocessing modules[7] this thesis leverages. To integrate a preprocessor into the data loading process seamlessly, it can be easily passed into the constructor of the `VideoDataset` class.

```
preprocessor = torchvision.transforms.Compose([
    # ...
])
dataset = VideoDataset(..., transform=preprocessor)

# the preprocessor is applied internally
preprocessed_sample = next(iter(dataset))
```

The `VideoDataset` class will load video segments into memory, apply the given preprocessing operation and store the resulting frames as `uint8` tensors into a buffer. As soon as a new sample is requested (via `next()`) the `VideoDataset` only has to access the internal buffer and return a segment of it. This is multiple magnitudes faster as reading a time window from disk every time `next()` is called.

[5]Nvidia RTX 3070
[6]`https://pytorch.org/vision/stable/index.html`
[7]`https://pytorch.org/vision/stable/transforms.html`

5.3 Video Normalization

Input data normalization has proven to enhance numerical stability for various optimization techniques typically encountered with artificial neural networks. To normalize individual frames, first, all frames are scaled into the interval $[0, 1]$ by dividing loaded frames by 255. Then the channel-wise mean and standard deviation are calculated based on all frames in each training dataset. This ensures that the input training dataset distribution is normalized to a standard distribution. Most behavior

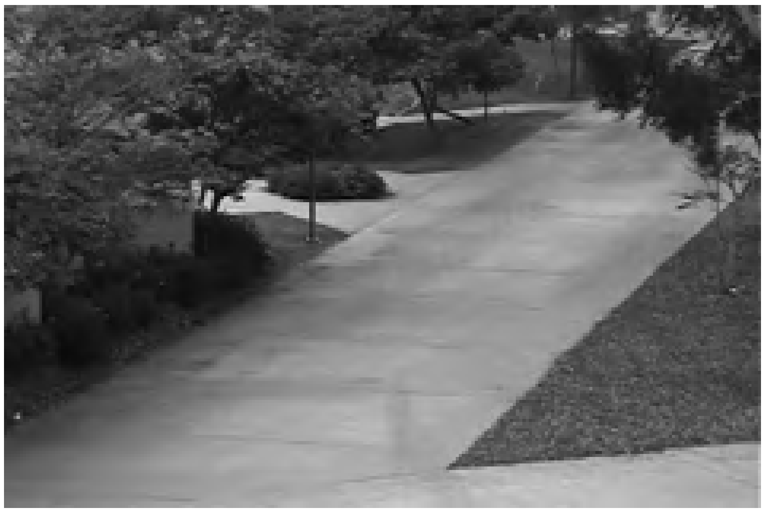


Figure 5.2: Extracting the global mean pixel values from the *UCSD 1* dataset will result in the background approximation of the scene.

analysis models are primarily interested in actors that occupy the scene. Therefore, background information about the scene can be discarded. For datasets that are captured from a static perspective, a naive posterior background subtraction algorithm can be used. The average pixel value that shares the same spatial position as all frames contained in the dataset is calculated. The foreground of a frame $F(im)$ can be isolated by subtracting the posterior global average frame bg with

$$F(im, bg) = im - bg$$

The combined normalization and background removal technique is implemented as a parameterized `torch.Module` and conforms to `torchvision` transformation standards and therefore can be used with the `Compose`[8] class. Because calculating normalization parameters (global average frame, average channel mean, average channel deviation) requires a complete traversal of the training dataset, all parameters are only calculated once and saved to a file. The process is completely automated via the implemented `pytorch_ligthning.DataModule` (chapter 5.5) and it's `prepare_data()` method. Generated normalization parameters for each dataset are also automatically loaded and passed to the implemented `Normalizer` class. For visualization purposes the `Normalizer` class implements a `.revert(normalized_input: torch.Tensor)` method so the generated frames can be restored into the original distribution and can be visualized without any color distortions.

[8]`https://pytorch.org/vision/stable/generated/torchvision.transforms.Compose.html`

5.4 Anomaly Annotations

Some video clips have anomaly annotations, primarily test and evaluation datasets. The structure of these annotations can vary drastically from dataset to dataset. For some datasets, annotations are provided as second intervals and for others frame intervals or direct frame annotations. To enable the `VideoDataset` to load available labels automatically, all provided labels are transformed into binary frame annotations. Each frame X_t is assigned a binary label $y_t \in [0, 1]$ with 1 representing an anomalous frame. After the transformation into our label representation, the label representation y is saved to a `numpy` text file alongside the video data. The `VideoDataset` class automatically detects whether labels are present and loads them automatically. It will return the labels alongside the image data as a python `tuple`.

```
dataset = VideoDataset(...)
(frames, labels) = next(iter(dataset))
```

Further detail on when and how these labels are generated is described in the following chapter 5.5.

5.5 PyTorch Lightning Data Modules as an Abstraction Layer

Most datasets have multiple types of datasets like *train*, *test*, *evaluation*. `PyTorch Lightning` provides an abstraction layer for datasets, data-retrieval and data loader configuration called `LightningDataModule`[9]. Each dataset gets its own `LightningDataModule` implementation, in which the following functionality is implemented:

API	Description
`train_dataloader()`	Returns the `DataLoader` object used for training.
`val_dataloader()`	Returns the `DataLoader` object used for validation.
`test_dataloader()`	Returns the `DataLoader` object used for testing.
`prepare_data()`	Download and transform labels if necessary.

The `LightiningDataModule` can be used in conjunction with the `Trainer`[10] class to easily implement the training, evaluation and testing process.

```
model = ...
trainer = pl.Trainer()

# will automatically download the dataset if necessary
data_module = UCSD()

# the correct loader will be used by the PyTorch Lightning trainer
trainer.fit(model=model, datamodule=data_module)
trainer.test(model=model, datamodule=data_module)
```

[9] `https://pytorch-lightning.readthedocs.io/en/stable/extensions/datamodules.html`
[10] `pytorch_lightning.Trainer`

CHAPTER 6

Irregularity Analysis Model Design

This chapter goes into detail and explains each neural network implementation this thesis covers thoroughly. Additionally, architecture-specific experiments and potential findings are described. Each architecture is then evaluated on each dataset from chapter 8. Almost all neural network approaches introduced in this chapter fall into one of two categories. One is a reconstruction-based autoencoder type, that will try to reconstruct its input tensor from latent space.

$$rec : \mathbb{R}^{T\times C\times H\times W} \to \mathbb{R}^{T\times C\times H\times W}$$

The second category is a future frame prediction (*abbr.* FFP) neural network. As the name would suggest, the network tries to guess a future frame $T+1$ of a given input sequence of size T. Formally the mapping of an FFP (ffp) model looks like this.

$$ffp : \mathbb{R}^{T\times C\times H\times W} \to \mathbb{R}^{C\times H\times W}$$

The output doesn't have any temporal dimension anymore, because the model will predict only one point in time. Even though both tasks seem very different most underlying architectures can be easily adapted toward either an *FFP* or a reconstruction-based model.

6.1 Self-Supervised Pixel-based Cuboidal Spatial Temporal Autoencoder

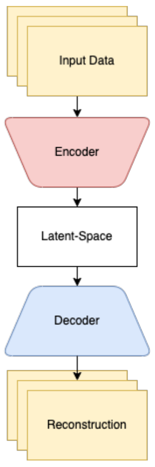


Figure 6.1: Flow of data through an autoencoder.

Because most of the available data provided by e:fs[1] is unlabeled, an unsupervised training approach is highly favorable. The Autoencoder (Schmidhuber, 2015) neural network architecture provides a great way of training model parameters self-supervised (Baldi, 2012). For more general information regarding the *Autoencoder* architecture refer to chapter 3.3.2. The neural network architecture used for the experiments in this thesis was inspired by *Learning Temporal Regularity in Video Sequences* (Hasan et al., 2016) which showed that you can detect anomalies in video sequences by using two-dimensional convolutional (Lecun et al., 1998) networks. The model is trained by batches of three-dimensional tensors with fixed dimensions of $[T, H, W]$, T being the desired length of an image sequence and H and W being frame width and height. T can be chosen freely during model creation/training and affects the model's training and runtime performance. Consequently, this architecture is only able to process monochrome images. Autoencoders primarily consist of three separate elements, visualized in figure 6.1. The *Encoder* $E : \mathbb{R}^{T\times H\times W} \rightarrow \mathbb{R}^{T\times H\times W}$ is responsible for compressing a given input $x \in \mathbb{R}^{T\times H\times W}$, into a significantly lower *feature-space*. This space is often referred to as the *latent-space* and serves as the *bottleneck* to the system. The *Decoder* $D : \mathbb{R}^{T\times H\times W} \rightarrow \mathbb{R}^{T\times H\times W}$ is used to invert the encoding operation by restoring a data-point in *latent-space* to *feature-space*. The mathematical formulation can be seen below.

$$\min \; D(E(x)) - x$$

To apply this approach to the camera surveillance use-case of this thesis, the assumption is taken, that the majority of the video footage used during training depicts non-anomalous behavior. If the recorded scene would be an intersection in which anomalous events happen regularly, it may not be feasible to use this type of technique. Instead, a supervised machine-learning approach would be more appropriate.

6.1.1 Architecture

The input to the network is a fixed-length series of monochromatic images each being a part of a continuous video sequence. Combined they build a single three-dimensional cuboid depicted in figure 6.2. The *Encoder* sub-network of the autoencoder consists of three convolutional layers with the first two followed by a two-dimensional max-pooling (Nagi et al., 2011) layer each. The first two-dimensional convolution is applied to the input cuboid with a kernel size of 14 and stride of 4. The layer will create

512 filters of the input data with each being of size $\sim 60 \times 60$. Max-pooling with a kernel size of 5 is applied to reduce the amount of computational complexity in the network and make the network generalize better. It is important to save the indices provided by the max-pooling layer, to be able to revert the operation during the decoding phase. The second convolutional layer is applied to the outputs of the first max-pooling layer and reduces the filter count to 256 with a kernel size of 5.

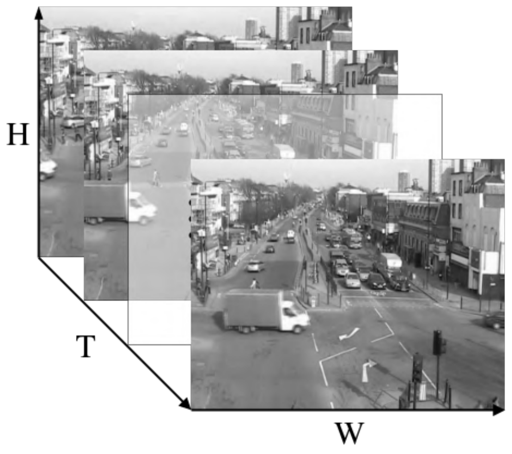


Figure 6.2: The constructed image sequence cuboid passed into the autoencoder network. The tensor has three dimensions of fixed length. One is the image sequence length T and the others are the width W and height H of all frames.

Again max-pooling is performed, this time with a lower kernel size of 2. Now the last convolutional layer transforms the data into the three-dimensional *latent space* of the autoencoder. It's of particular importance, that the latent space is multidimensional because otherwise spatial information that the *encoder* needs to reconstruct the original image sequence would be lost. The *decoder* now reverts all operations of the *encoder* in reverse order, with matching layer parameters. The following layer implementation is used to revert the standard two-dimensional convolution and max-pooling implementation.

- PyTorch `MaxUnpool2d` (implementation)
- PyTorch `ConvTranspose2d` (implementation)

A visualization of the proposed architecture is shown in figure 6.3. Each convolution and pooling block is followed by a non-linearity function. *Learning Temporal Regularity in Video Sequences* (Hasan et al., 2016) suggests either *tanh* or *sigmoid* as an activation function for this type of neural network architecture. For this thesis, the *sigmoid* activation is chosen based on empirical observations. The model is trained with the *Adam* (Kingma and Ba, 2014) optimizer and a learning-rate of 0.01. For regularization, L^2 norm is used over all model parameters with its intensity set to $1e^{-5}$. The choice of learning parameters is based upon the author's choice and reasoning stated in the original publication *Learning Temporal Regularity in Video Sequences* (Hasan et al., 2016).

6.1.2 Training

Getting the model to converge turned out to be rather easy to achieve. Because all training datasets have a fixed perspective of the scene, the model quickly learned to reconstruct the scene's (mostly) static background. Areas that typically show movement like streets or sidewalks are reconstructed with considerably less remaining detail. The network is also not able to reconstruct actors in the scene with sufficient

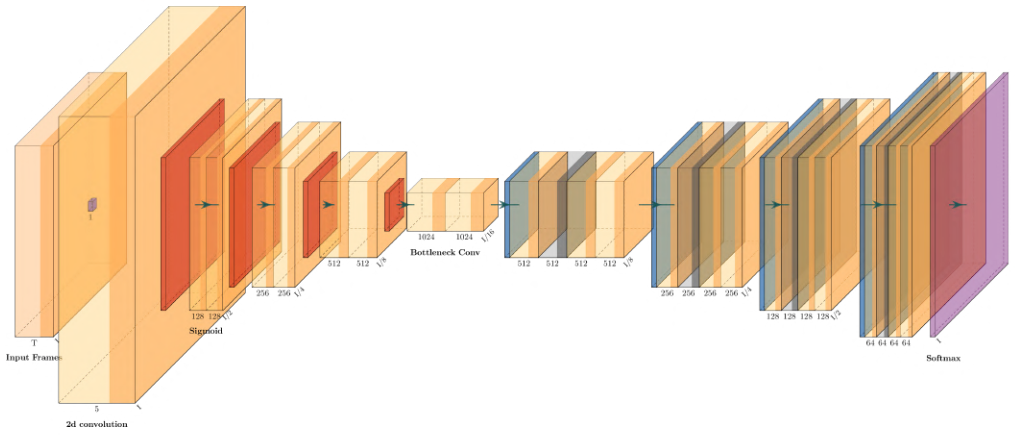


Figure 6.3: A visualization of the architecture used.

detail. Actor movement in general is not learned very well by the network architecture. Instead, the network appears to have a significant bias toward reconstructing areas of the videos that don't exhibit any movement. This behavior is unfortunately the opposite of what could be considered desirable. For detecting irregular behavior, especially moving regions of the frame are of interest.

6.1.3 Experiments & Findings

One-Dimensional Latent Space

The first architecture prototypes incorporated a n-dimensional latent space feature vector as an intended bottleneck to the system. This isn't atypical and many autoencoder architectures use a one-dimensional latent space. In combination with convolutional neural network layers, a one-dimensional bottleneck will, unfortunately, remove any spatial information from the *encoder* input entirely. Therefore, the *Decoder* isn't able to reconstruct the input from the latent space anymore.

Pixel-Based Contrast Bias

An unfortunate side-effect of working with the video frames on a pixel layer is that the optimizer is heavily biased towards higher contrast areas of the scene. Figure 6.4 shows an example of this behavior. This is a direct result of the criterion function the optimizer uses to tune model parameters. The *mean squared error* loss function or in other words the distance between two corresponding pixel values. Let $P(x, y)$ be the reconstructed pixel value at position (x, y) and $I(x, y)$ be the encoder input. The loss function is, therefore:

$$loss(x, y) = |P(x, y) - I(x, y)|_2^2$$

Unfortunately, this loss function is ill-posed for the problem the network tries to solve. The primary subject while performing anomaly detection is the behavior that dynamic objects exhibit. Objects that are very close to the scene's background pixel exhibit a very low response from the loss function in comparison to an object of a drastically different color. Desirable would be an equal response from both objects because both objects are semantically equally relevant. If the background can be extracted

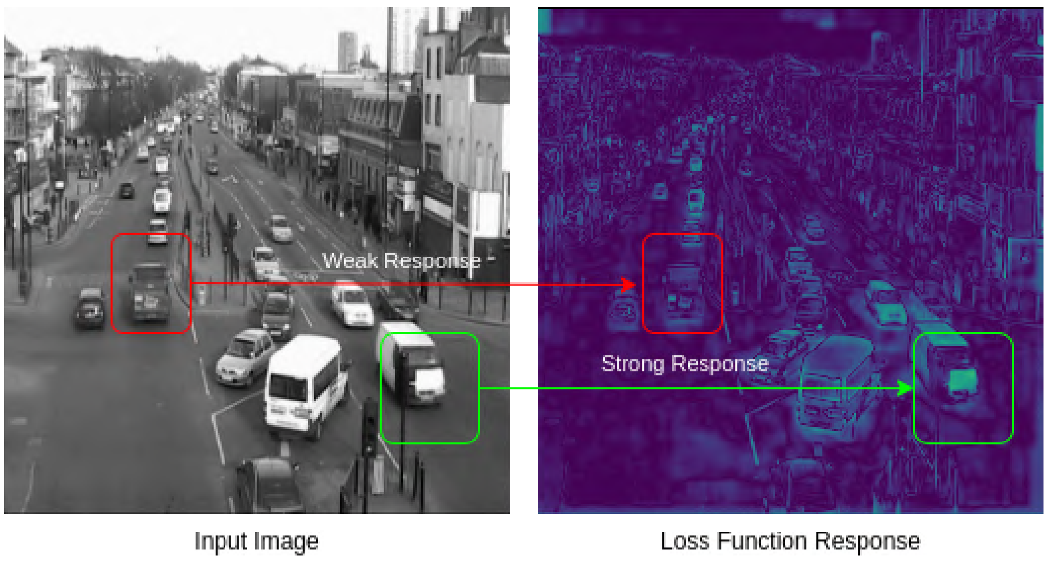


Figure 6.4: This visualization shows the bias towards high contrast objects, exhibited by the loss function (right).

from the frame the objective function can be improved by removing the background values. Let $P'(x, y) = |P(x, y) - B(x, y)|$ be relevant information, with $B(x, y)$ being the backgrounds value at position (x, y). Consequently, $I'(x, y) = |I'(x, y) - B(x, y)|$ is the relevant information from the input frames to the network. The now background bias corrected loss function is now:

$$loss'(x, y) = |P'(x, y) - I'(x, y)|$$

To isolate background and foreground from each other, an off-the-shelve background estimation algorithm can be used.

Importance of relative Object Size

The objective function needs to reduce each pixel distance to a single *loss* value. Typically, one would use the mean or average over all pixel deltas. Objects that cover fewer pixels of each frame have less influence on the overall loss compared to objects that take up more space in the frame. To counteract this bias, a separate segmentation network can be used, that generates binary masks for selected actor types. These masks allow us to generate separate anomaly scores for each type of actor. This option is not further explored in this thesis but is a potential future improvement.

Perspective Bias

A similar effect to the one described in chapter 6.1.3 is that the footage taken by the camera is perspectively distorted. This will result in a bias towards anomalies that appear close to the camera. A relatively crude but effective way of mitigating the perspective bias is, to apply a perspective correction to the video stream. The associated quality loss can be disregarded for most model architectures, because they work with a much lower input resolution. To apply a perspective correction, one must know scene-specific parameters to apply a homography to each frame.

6.2 Self-Supervised Pixel-based Cuboidal Spatial Temporal Autoencoder (***FastAno***)

An architecture similar to the one described in the previous chapter 6.1, is an autoencoder architecture proposed by *FastAno: Fast Anomaly Detection via Spatio-temporal Patch Transformation* (Park et al., 2021) that promises very good anomaly detection performance while having a simple architecture and fast execution speeds. Compared to the model described in chapter 6.1, the *FastAno* architecture leverages 3D convolutions to process spatial and temporal information. The architecture follows a variation of the U-Net (Ronneberger, Fischer, and Brox, 2015) convolutional autoencoder architecture with the encoder consisting of three stacks each containing a 3d convolutional and batch normalization layer. The decoder reverses the encoder architecture by replacing each convolutional layer with the respective deconvolutional layer. While decoding, the decoder will reduce the temporal dimension $0 \dots t$ to a single point $t+1$, the prediction for the next frame. This ranks this model into the future frame prediction category. During training, quadratic regions of the input cuboid get

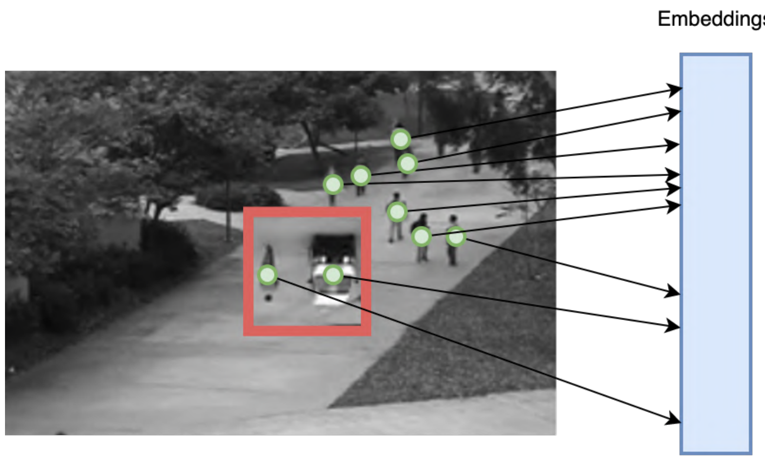


Figure 6.5: Training the model to identify rotated features, leads to more semantically relevant information in each frame embedding.

either rotated or temporally permuted. This should lead to better performance by the network (Park et al., 2021) because the network is forced to embed more critically important information into the latent space (Zaheer et al., 2020). The intuition behind this technique is that a system requires more semantical knowledge about objects if they should be recognized even if they are rotated (Gidaris, Singh, and Komodakis, 2018). After a few experiments and a qualitative and quantitative evaluation, significant improvements could be registered compared to the architecture described in 6.1. The architecture achieves remarkable performance, especially considering its low complexity and fast convergence. A thorough evaluation of this architecture on all

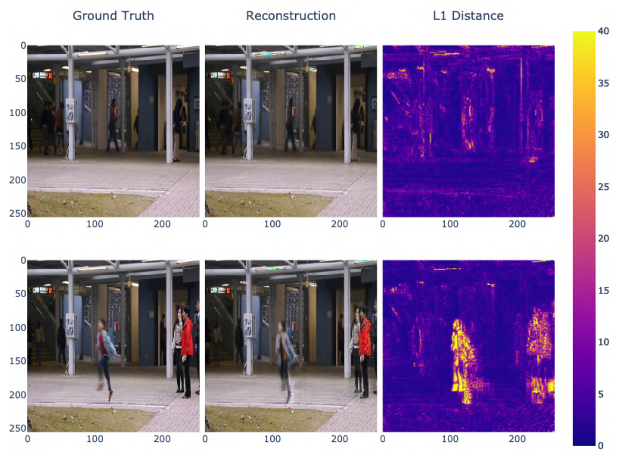


Figure 6.6: Comparison between a predicted frame with an anomaly present (bottom row) and one without any (top row).

datasets used in this thesis can be found in chapter A.2.

6.3 Spatial Convolutional LSTM Autoencoder

The neural network architecture proposed in chapter 6.2 is easily able to learn the spatial relations of the input dataset. But especially the network described in 6.1 struggles to efficiently learn the temporal relations between consecutive frames. Additionally, it's limited to monochrome input images because of the utilized 2d-convolution. Recent research has shown that using recurrent neural network architectures can improve the temporal learning capabilities of neural network architectures, while at the same time reducing their overall parameter count and computational complexity (Liu et al., 2018; Samuel and Cuzzolin, 2021; Luo, Liu, and Gao, 2017). Popular recurrent layer architectures like *LSTM*[1] or *GRU*[4] can learn temporal relations between different time steps. Because the internal state of the recurrent cell is a vector, spatial information about the input image(s) is discarded. The convolutional LSTM cell architecture proposed by *Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting* (Shi et al., 2015) replaces matrix multiplication operations of the LSTM cell with convolution operations. This allows the cell to store spatial information in addition to temporal information in its internal state. These properties can be used to construct an autoencoder similar to the one described in 6.1, except it uses multiple convolutional LSTM cells as the bottleneck of the network. Figure 6.7 shows a broad overview of the proposed architecture. The operations of the spatial feature extractor are applied to every frame of an image sequence individually. Not only does this reduce the parameter count of the network significantly, but also lets the LSTM layer focus exclusively on learning the temporal relations between a continuous series of time steps. While spatial encoder and decoder focus on solely constructing or deconstructing feature maps. This design follows the principle of separated concerns. The network receives an input of T consecutive (potentially)

[1] *abbr.* for *Long Short-Term Memory*

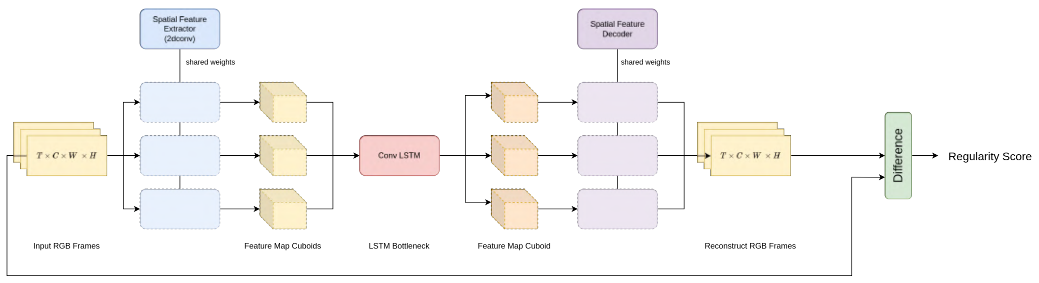


Figure 6.7: Each input frame gets transformed into a feature cuboid and passed into the LSTM unit. The recurrent ($LSTM$) unit acts as the bottleneck to the network. Finally a deconvolutional module transforms the $LSTM$ output back into input space. The difference between the input and the network's output is used as an indicator of regularity.

multi-channel frames. A spatial encoder SE compresses each frame individually into a feature map with 512 channels of size 12×12. Each feature map serves as an input to the convolutional LSTM module TE. The LSTM module creates a many-to-many reconstruction of size T. The last operation applied is a stacked deconvolution SD that reconstructs input space from feature space. The formula below describes how the components of the autoencoder interact with each other.

$$\hat{x} = SD(TE(SE(x)))$$

The deviation between the input frame sequence and the reconstructed output of the network is used as an indicator of whether anomalous behavior occurred in the given image sequence. To measure the distance between input and output images, the $L1$ norm can be used. The basis of this choice is research that suggests better training performance by using $L1$ over $L2$ norm (Zhao et al., 2017). Early results suggest that the reconstructed images appear to be very blurry and introduce significant noise. For detailed results refer to the evaluation chapter 8.4. A second architecture inspired by *Robust Unsupervised Video Anomaly Detection by Multi-Path Frame Prediction* (Wang et al., 2020) suggests leveraging lateral connections to allow the model to reconstruct objects in more detail. Unknown behavior or objects to the network are reconstructed in less detail and therefore cause a higher difference between input and reconstruction. The architecture follows a convolutional encoder-decoder architecture with lateral connections similar to *U-Net* (Ronneberger, Fischer, and Brox, 2015). Convolutions are applied to each frame of a series of fixed length T in parallel. Four stacked 2d-convolution layers make up the encoder of networks with $128, 64, 64$ and 32 filters respectively. The kernel size and stride are set to 3 and 2 for each layer in the encoder. Each convolutional layer passes its output in addition to the next encoding layer to a convolutional $LSTM$ layer (Shi et al., 2015) which makes up the lateral connections to the decoder. The output signatures of every lateral $LSTM$ layer are equal to their inputs and their kernel size is set to 3. The decoder is made up of deconvolutional layers mimicking the encoder's architecture but reversed. All layers are followed up with *ReLU* non-linearity functions. For training the model the *Adam* optimizer is used. This choice is based on the publication *Robust Unsupervised Video Anomaly Detection by Multi-Path Frame Prediction* and its overall popularity in the machine learning community. The optimizer is initialized with a learning rate of $1e^{-4}$ and a weight decay of $1e^{e-6}$. The results presented in the evaluation chapter 8.4 show that the lateral connections help to improve the reconstruction quality and therefore improve the inferred anomaly/regularity score.

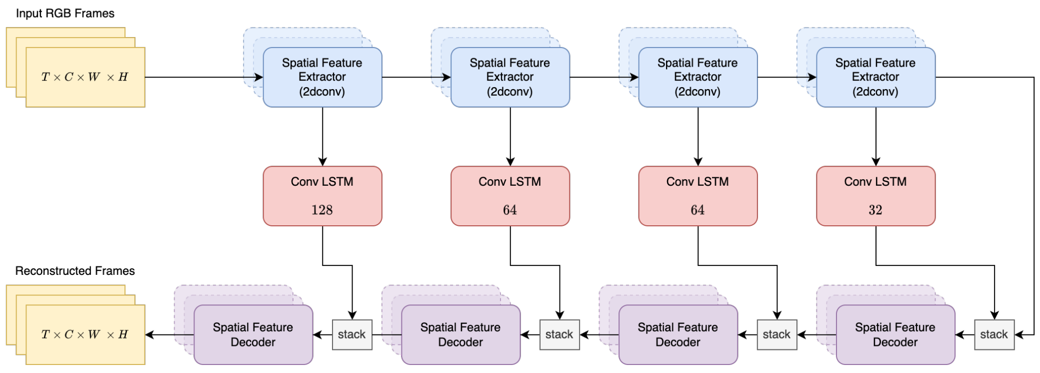


Figure 6.8: Spatio-temporal autoencoder architecture with lateral *LSTM* connections.

6.4 Many-to-One Future Frame Prediction Transformer

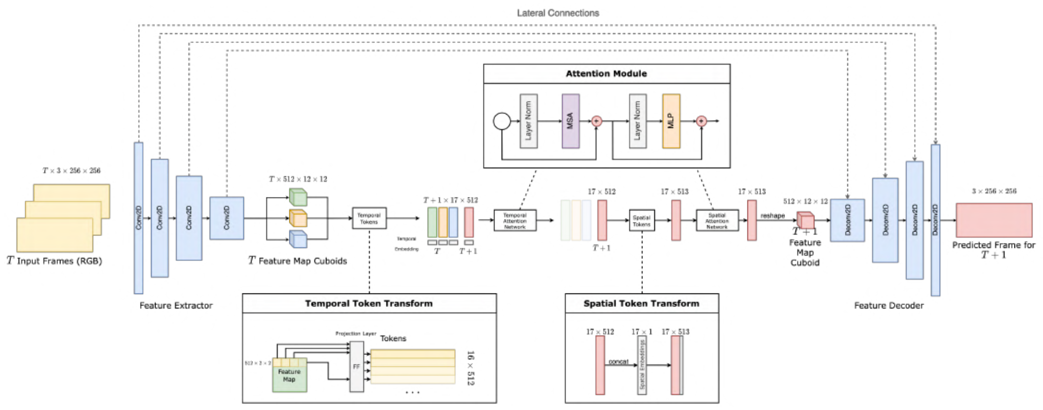


Figure 6.9: Transformer model architecture visualization.

The *LSTM convolutional* architecture described in chapter 6.3 is unable to capture spatial relations outside the highly limited context of the individual convolutional kernels. Contrary, the transformer neural network architecture (Vaswani et al., 2017) demonstrated highly capable of capturing the global context of an input sequence. Transformer networks are similar to an encoder, in that the network tries to compress the input space into a highly abstracted and compressed latent space. Especially transformer-based language models like *GPT-3* (Brown et al., 2020) are characterized by their ability to capture global context efficiently. The publication (Yuan et al., 2021) applies the multi-head self-attention transformer architecture to the anomaly detection problem class. Compared to other previously mentioned approaches the goal of this model is not to reconstruct an input sequence from latent space, but predict a future frame $T + 1$ from a T frame long input sequence. This type of model is often called a *FFP* (future frame prediction) model. Similar to previous architectures the model performance is used as an indicator for anomalies.

6.4.1 Model Architecture

The model architecture is highly inspired by the U-Net (Ronneberger, Fischer, and Brox, 2015) architecture, visualized in figure 6.10. The bottleneck section of the network is replaced with a spatio-temporal transformer, that is described in more detail below. Because the input to the model is a sequence of images and therefore very informationally dense, a layer of abstraction is required to reduce the overall computational complexity of the model. The U-Net feature extractor and decoder are applied individually to every frame of a T long input sequence. The resulting T feature map cuboid from the U-Net feature extractor, are each individually tokenized by separating all feature maps into $n \times n$ sized chunks. These chunks are then projected via a fully connected layer onto a token vector. The process is visualized in region *"Temporal Token Transform"* of figure 6.9. Feature map cuboids are therefore transformed into a $T \times S \times d$ with S being the number of total chunks and d being the internal token dimension. This transformation subsequently strips the temporal identity information of all token sets. To let the network differentiate between tokens of a specific time step, each time step is augmented with a single randomized token that allows the transformer module to differentiate between tokens.

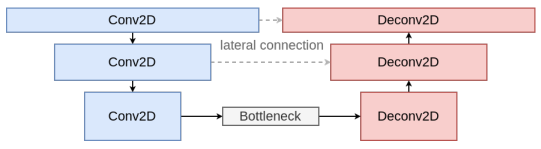


Figure 6.10: Architecture concept of the U-Net model. Encoder marked as *blue* and decoder as *red*.

6.4.2 Training

To train the model a series of different loss functions are utilized. The already specified *pixel-distance* loss is used to ensure the predicted pixels match as close as possible to the ground truth frame.

$$L_{dist}(x, y) = |x - y|_2$$

Additionally, the L_2 distance between the image gradient of the predicted frame and ground truth is added to the overall training criterion. This loss is referred to as *gradient-loss* in the context of this thesis. The objective of this loss is to drive the model to sharpen parts of the image that are also sharp in the ground truth frame. If the generator encounter objects that are not known to the model, it'll fail to successfully sharpen the edges of said object. (Yuan et al., 2021) To further improve the prediction accuracy, a discriminator D as part of an adversarial training module is introduced. The discriminator is trained to differentiate between generated and real frames. The generator G tries to trick the discriminator into classifying a generated/predicted frame as *real*.

$$L_{disc}(x) = 1 - D(G(x))$$

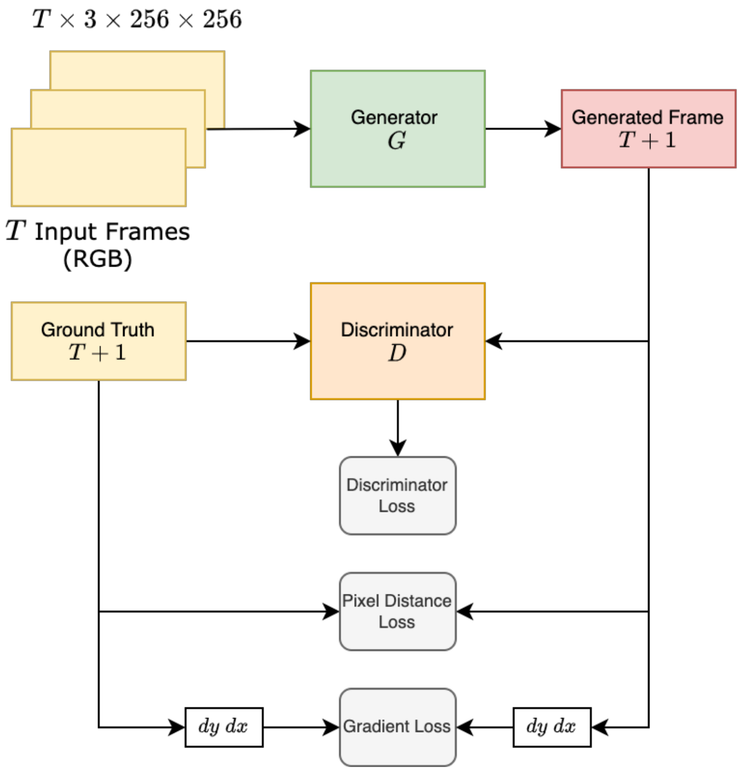


Figure 6.11: A visualization of all loss functions used during generator training.

Simultaneously a separate optimizer trains the discriminator to correctly classify images as either *generated* or *real*. During model training, both optimizers will eventually reach an equilibrium. The final training criterion is the weighted sum of all previously introduced loss functions.

$$L(x, y) = \lambda_{dist} L_{dist}(x, y) + \lambda_{grad} L_{grad}(x, y) + \lambda_{disc} L_{disc}(x)$$

The different weights (λ) chosen are further elaborated in the corresponding evaluation chapter. For both optimizers, *Adam* is chosen based on other literature (Yuan et al., 2021) and empirical evidence.

6.4.3 Experiments & Findings

Hyperparameter Tuning

To optimize hyper-parameters for the transformer model, a random search over the hyper-parameter search space is performed. The hyper-parameter search space is selected by manually selecting a plausible value range for every parameter. Table 6.1 shows the search configuration for the hyper-parameter tuning operation. Each model with a random parameter configuration is trained for 30 epochs on a single *Nvidia* A100 GPU. Because multiple *GPUs* are available to the system multiple models can be trained in parallel. After all hyper-parameter tuning training jobs are finished, the most promising parameter configuration is trained further to evaluate its effectiveness. The model parameter configurations and evaluation metrics can be seen in table 6.3.

Table 6.1: Hyperparameter Search Space

Attribute Name	Search Space
Frame Strides	[7, 8, ..., 14]
Gradient Loss Weight	0.5 - 5 (uniform sampling)
Reconstruction Loss Weight	0.5 - 2 (uniform sampling)
Discriminator Loss Weight	2 - 12 (uniform sampling)
Difference Loss Weight	4 - 8 (uniform sampling)
Token Dimensionality	[128, 256, 512, 1024]

Table 6.2: Hyper-parameter tuning results for the transformer network architecture performed on the UCSD pedestrian 2 dataset. Only the top 20 training runs of a total of 50 runs are displayed in the table below.

SSIM AU-ROC	PSNR AU-ROC	PMSE AU-ROC	F Strides	Token Dim	Recon. Loss Weight	Grad. Loss Weight	Diff. Loss Weight
0.758	0.753	0.762	7.0	256.0	1.296	0.551	5.354
0.742	0.726	0.723	8.0	512.0	0.74	4.716	6.517
0.74	0.723	0.734	9.0	1024.0	0.94	3.081	6.485
0.74	0.732	0.73	8.0	1024.0	1.472	0.881	6.829
0.739	0.725	0.722	7.0	256.0	0.861	1.126	7.355
0.734	0.719	0.724	8.0	128.0	1.589	2.142	5.151
0.733	0.721	0.734	7.0	256.0	0.645	2.206	7.331
0.726	0.705	0.715	9.0	128.0	1.38	4.543	4.52
0.722	0.706	0.714	9.0	128.0	0.616	4.635	7.52
0.721	0.708	0.709	8.0	1024.0	1.624	3.145	6.414
0.711	0.703	0.702	8.0	128.0	0.968	1.371	7.351
0.68	0.658	0.671	10.0	1024.0	1.509	4.158	7.013
0.677	0.657	0.672	10.0	256.0	0.59	2.012	6.81
0.669	0.652	0.664	10.0	128.0	1.676	1.862	4.225
0.668	0.651	0.668	12.0	128.0	1.066	2.453	4.649
0.663	0.653	0.66	10.0	1024.0	0.825	0.67	7.237
0.663	0.648	0.663	11.0	1024.0	1.928	1.068	5.429
0.662	0.648	0.646	10.0	128.0	1.35	0.627	5.593
0.661	0.654	0.659	13.0	1024.0	1.4	1.617	6.549
0.658	0.649	0.655	13.0	256.0	1.467	3.62	6.943

Through table 6.2 a strong correlation between the *AUROC* score(s) and the *Frame Strides* hyper-parameter can be observed. To further evaluate if the model architecture can leverage the increased temporal information present in frame windows with frame strides, an additional hyperparameter optimization run with disabled frame strides is performed.

Table 6.3: Hyper-parameter tuning results for the transformer network architecture performed on the *UCSD 2* dataset.

SSIM AU-ROC	PSNR AU-ROC	PMSE AU-ROC	F Strides	Token Dim	Recon. Loss Weight	Grad. Loss Weight	Diff. Loss Weight
0.831	0.83	0.827	1.0	512.0	1.603	0.512	4.714
0.831	0.827	0.827	1.0	512.0	1.64	3.325	5.948
0.82	0.821	0.812	1.0	512.0	1.11	1.972	7.409
0.825	0.828	0.823	1.0	512.0	0.721	1.726	4.012
0.815	0.82	0.812	1.0	512.0	0.904	0.744	4.156
0.811	0.812	0.805	1.0	512.0	0.929	1.129	6.948
0.806	0.811	0.802	1.0	512.0	1.056	3.241	4.205
0.807	0.806	0.798	1.0	512.0	1.042	2.228	6.405
0.804	0.803	0.796	1.0	512.0	1.232	1.902	7.415
0.805	0.803	0.802	1.0	512.0	0.82	1.914	4.015

It is observable that a model trained on data with no frame strides performs significantly better than a model that is. For a more detailed analysis of how the *Frame Stride* parameter affects the model performance, refer to the qualitative analysis in chapter 6.4.3. All *AUROC* scores are calculated based on model performance on the *Test* dataset. Each test clip of the *UCSD Pedestrian* dataset consists of 36 video clips, each 8 seconds long. The `WindowedVideoDataset` (see chapter 5.1.1) implementation is used to load test clips and apply the same prepossessing operations that are used during training. As soon as all sliding windows of a testing video clip were propagated through the network the resulting frame distance values are normalized to generate the anomaly values for every frame. This method for evaluating the anomaly detection performance of a model is typically used in other publications as well. Yet a good score doesn't necessarily mean as good of real-world performance, because real-world videos are often much longer than 8 seconds. This can significantly affect real-world model performance as soon as a model architecture is susceptible to regime shifts (chapter 8.1.2) in the input data. In most datasets covered in this work, regime shifts happen between scenes where there is much global movement and scenes where almost no movement happens. It's very unlikely that a video clip of 8 seconds contains such a drastic change in total movement. Because almost all test clips are separated by significant time gaps, combining all clips into a single, longer clip is not a viable option either.

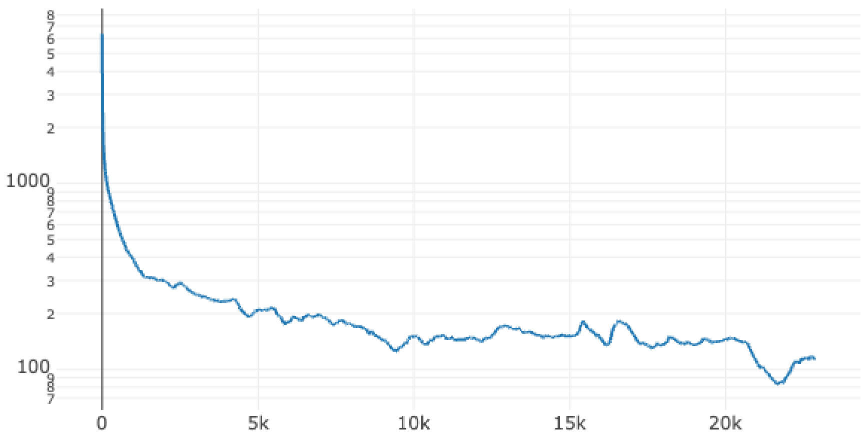


Figure 6.12: Training progress of the model's main loss function. The main loss function is defined as a weighted sum of all minor loss functions.

Implications of the *Frame Stride* Hyper-Parameter on the Model

In the previous section 6.4.3 *"Hyperparameter Tuning"* a significant correlation between model performance and the *Frame-Stride* hyper-parameter was discovered. Besides viewing the model purely through a numerical perspective, a qualitative analysis of the generation results allows us to learn more about the model's behavior. Figure 6.13 shows the different generation results from two models which only differ from each other by their *frame-stride* parameter. The model with *frame-stride* set to 1 is referred to as M_1 and the other as M_2. The vehicle highlighted by the red dashed box is being reconstructed very closely by model M_1. Compared to M_1, M_2 struggles to accurately predict the vehicle and even substitutes it with a person (marked in green in figure 6.13). The results generated by M_2 are for our use-case more desirable because the vehicle was not present in the training dataset and should subsequently be considered an anomaly. This behavior most likely originates from the model hav-

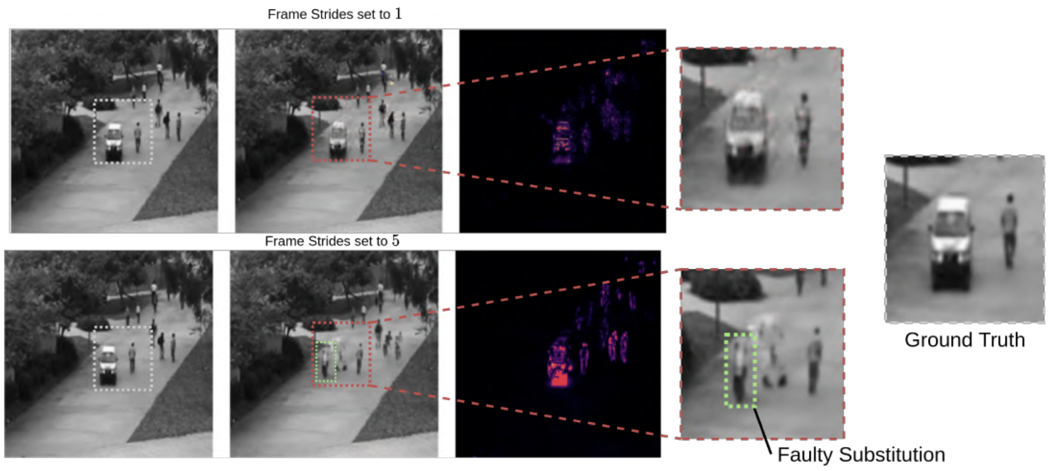


Figure 6.13: Two models trained with different frame strides and their predictions compared to each other.

ing an easier time reconstructing a time-step $t + 1$ which is closer in time to t and has naturally less difference. Unfortunately, a higher time difference also increases the overall reconstruction noise and therefore leads to a less distinctive anomaly signature of a frame. The anomaly heat map in figure 6.13, shows the average $L2$ distance of all corresponding pixels in the predicted and ground truth frame.

6.5 High-Level Representation Learning with Graph Neural Networks

Previous approaches can detect point anomalies with spatial context reliably. Contextual anomalies based on temporal and spatial context can be hard to identify by the networks. As previously mentioned, it's harder for neural networks to identify complex contextual anomalies based on a high-dimensional input space. Image pixels are highly dimensional and being able to learn actor behavior requires a high level of learned abstraction. If the primary objective of the anomaly detection system is to detect behavioral anomalies efficiently, a new approach is required. Anomalous actor behavior most often involves some form of interaction with one or many other actors. Graphs are a fitting data structure to model actors and actor relations in a scene. Each node in a graph represents an actor of a scene and edges represent spatial or temporal actor relations. If a system can learn normal actor relations in a graph, this knowledge can be used to identify abnormal nodes or clusters. Pourreza, Salehi, and Sabokrou (2021) proposes a system that uses an off-the-shelve object detection network, to extract a high-level representation of the input image(-sequence). A directed weighted graph structure is populated with the detected objects. Thereupon a global summary vector of the whole graph is constructed (Veličković et al., 2018) and a discriminator is trained to distinguish between nodes that are valid for a given global graph summary vector and irregular ones.

6.5.1 Graph Construction

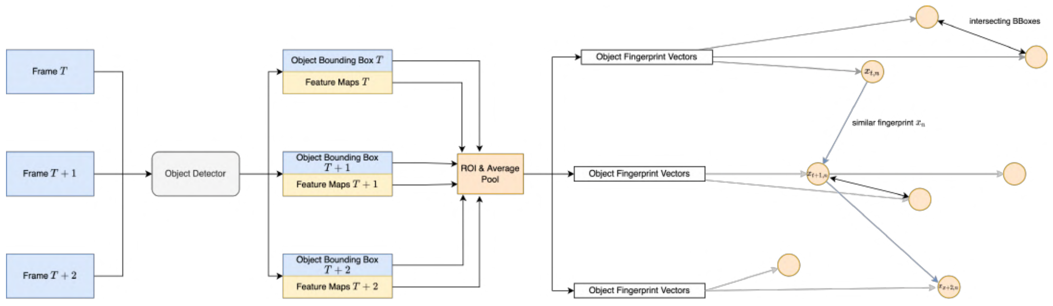


Figure 6.14: Constructing a spatio-temporal graph from object bounding boxes detected by an object detector.

Like Pourreza, Salehi, and Sabokrou (2021) suggests, a *Faster R-CNN* model (Ren et al., 2015b) trained on the *Microsoft COCO* (Lin et al., 2014) dataset is used as the object detector. The *Detectron 2* (Wu et al., 2019) *Faster R-CNN* implementation together with the provided weights is used. *Faster R-CNN* is used as the object detector architecture because it can be potentially used in soft real-time applications because of its fast runtime. Other speed-optimized architectures like *Yolo V5* are also already implemented but are currently not utilized. Each arbitrary-sized image of a series $I_{0...t}$ gets individually processed by the object detector. *Detectron 2* provides us with the bounding boxes of all objects that are in the scene O_{N_t} with N_t being the number of objects N in frame t. Additionally, a *PyTorch* forward hook gets used to extract the last activation F_t from the object detectors backbone *ResNet*. F_t is a feature map encoding high-level semantic information about objects in a given image. Feature map F_t and object bounding boxes O_{N_t} are used in an

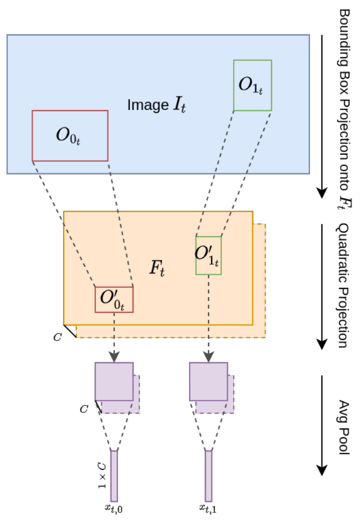


Figure 6.15: The ROI pooling procedure to extract feature vectors from feature maps.

ROI pooling (Ren et al., 2015b) procedure[2] followed by
averaging pooling the extracted patch of the feature map. Region of interest (*abbr.* ROI) pooling extracts the bounding box region of an image. This process is graphically depicted in figure 6.15. The resulting object vectors $x_{t,i} \in \mathbb{R}^C$ are a summary of all high level features extracted by the object detector for a given object i of N_t.

The final constructed graph should be a projection of spatial and temporal relations between actors. Spatial relations of a time t are modeled as an undirected $N_t \times N_t$ adjacency matrix Gs with each element $Gs_{i,j}$ representing the intersection of bounding box O_i and O_j. For calculating the intersection between two bounding boxes the IoU[3] is used. To model temporal connections, the similarity between two feature vectors of the same objects $x_{t,i}$ and $x_{t+1,i}$ at different times in space is exploited. Because the feature vectors originate from a high-level feature map, subtle pixel changes triggered through movement should not affect an object's feature vector much. As suggested by Pourreza, Salehi, and Sabokrou (2021) the *cosine similarity* function is used to measure the distance between $x_{t,i}$ and $x_{t+1,i}$. Therefore, the directed temporal graph x is constructed by comparing each element at time t with all the elements $t+1$. Finally, a combined spatio-temporal graph G_{st} can be constructed by combining G_{temp} and Gs in the following manner:

$$G_{st} = \begin{pmatrix} G_t^{spatial} & G_{t,t+1}^{temporal} & 0 & \dots & \dots \\ 0 & G_{t+1}^{spatial} & G_{t+1,t+2}^{temporal} & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & 0 & G_T^{spatial} & G_{T-1,T}^{temporal} \end{pmatrix} \in \mathbb{R}^{N \times N}$$

G_{st} is a quadratic directed $N \times N$ graph with N being the total amount of objects present in each frame of a time sequence T. Consequently, N can be of arbitrary size and the graph can be generated efficiently by discarding early time steps and appending new ones.

6.5.2 Architecture

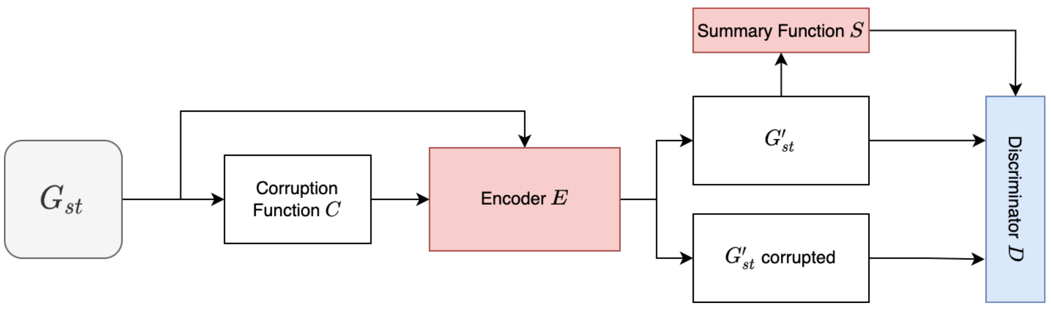


Figure 6.16: Deep Graph Infomax Architecture.

For learning regularity the approach suggested by Veličković et al. (2018) is utilized. First, the previously constructed spatio-temporal graph G_{st} is encoded using

[2]In this implementation feature map F_t gets scaled up by a factor $2\times$, so the case of an object being smaller than a 1×1 region after the projection is avoided.

[3]abbrev. for *Intersection over Union*

a graph convolutional layer. Nodes in the graph that are connected to each-other pass their information to themselves. After performing the graph convolution(s) a summary vector S of the graph is constructed.

$$S = \frac{1}{N} \sum_{i=0}^{N} x_i$$

Then a discriminator D is trained to differentiate between regular and non-regular nodes by evaluating a node x_i in relation to S.

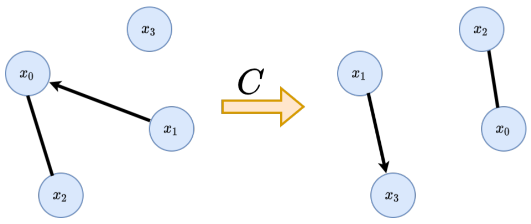


Figure 6.17: Affect of the corruption function C on a graph G.

For training the discriminator negative samples are required, otherwise, the discriminator isn't able to learn the difference between regular and irregular nodes. To generate negative samples the graph G gets corrupted by a corruption function C. A random row-wise permutation is applied to the graph G as C. The transformation is visualized in figure 6.17. The corrupted graph $G' = C(G)$ is then also propagated through the encoder E. Positive and negative samples are then passed to the discriminator D to distinguish between them. Figure 6.16 illustrates the whole architecture.

CHAPTER 7

Deployment and Production Usage

The resulting network of this thesis can be deployed in conjunction with a web visualization platform, a high-performance data storage solution and a *RTSP* interface. All services are containerized and can be deployed to a *Kubernetes* cluster. The different components utilized are further described in detail in this chapter.

7.1 RTSP Proxy

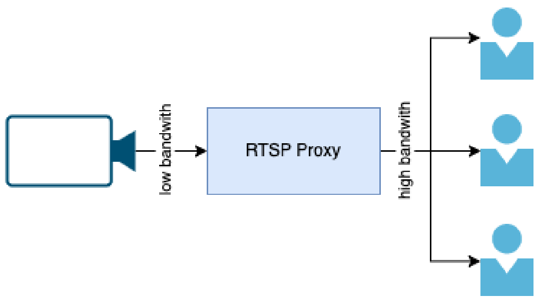


Figure 7.1: A demonstration of how the proxy-server is being used, to relieve the RTSP camera uplink.

The *FKK* camera stream is available as an RTSP stream. Unfortunately, the camera uplink is unable to support a multitude of connections simultaneously. Because the camera is used by multiple research groups, a more robust and better-performing solution is required. The solution should be able to scale to many concurrent streams efficiently without any stream interruptions. Figure 7.1 visualizes the new architecture. A proxy server is used that can redistribute the *RTSP* stream to an arbitrary amount of clients. The open-source software *rtsp-simple-server*[1] lets us redistribute, record and transcode the input stream. Ultimately the server provides different streams under the following paths:

[1]`https://github.com/aler9/rtsp-simple-server`

`/proxy`	An identical mirror of the source stream.
`/compressed`	Transcoded and compressed mirror of the source stream. Optimal if one wants to save bandwidth while still receiving a Full-HD stream.
`/scaled`	A highly scaled-down version of the source stream. The Full-HD streams gets compressed to 360p. This variant is best if one doesn't need the high resolution other streams provide and prioritize overall stability.

Additionally, all stream variants are on-demand transcoded with `x264` and served as *HLS* streams. In contrast to *RTSP*[2], *HLS*[3] streams can also be played in a web browser. This is especially useful for visualizing the stream in a web interface.

7.2 Inference Service

To infer regularity/anomaly scores from a continuous stream of data a containerized inference service is used. The inference service has access to the preprocessing classes associated with each architecture. For inference, the native software platform *PyTorch* is avoided. Instead the lightweight machine learning runtime library *ONNXRuntime* (onnxruntime.ai) is used. *ONNXRuntime* enables the use of many different hardware accelerators[4] on a multitude of platforms. The library is also very lightweight and therefore is perfectly suited for containerization. For invoking the runtime, the *ONNXRuntime Python* bindings are used. The inference service connects to the database (see 7.3) and logs inferred information to said database. Input data is provided by the RTSP stream provided by the *RTSP Proxy* service explained in chapter 7.1. If a second camera is added to the system, it is sufficient to start a second inference service container.

7.3 Data Persistence

To analyze historic data, a way to persistently store video and the corresponding model inference data is required. The data to store is by nature of historical form and manipulating historic data is not regularly required. Typically only data points at the current time step are written to a database. There are a variety of different database distributions that specialize in time-series data. Popular examples would be *TimescaleDB* (www.timescale.com) and *InfluxDB* (www.influxdata.com). For this thesis, the latter is chosen based on its ability to process extremely high-frequency input data streams. Additionally, *InfluxDB* allows this project to scale up to support multiple camera streams and inference services simultaneously.

Persisting the surveillance camera's video stream allows machine learning models to be trained on continuously growing datasets. The *RTSP Proxy* (explained in chapter 7.1) is used to automatically store the stream from the camera as a 1080p video

[2]*abbr.* for Real-Time Streaming Protocol

[3]*abbr.* for HTTP-Streaming

[4]CoreML, CUDA, DirectML, oneDNN, OpenVINO, TensorRT, NNAPI, ACL, ArmNN, MIGraphX, Rockchip NPU, SNPE, TVM, Vitis AI (source: onnxruntime.ai)

file to cloud storage. To improve the storage efficiency the *RTSP Proxy* additionally compresses the stream in real-time.

7.4 Grafana Dashboard

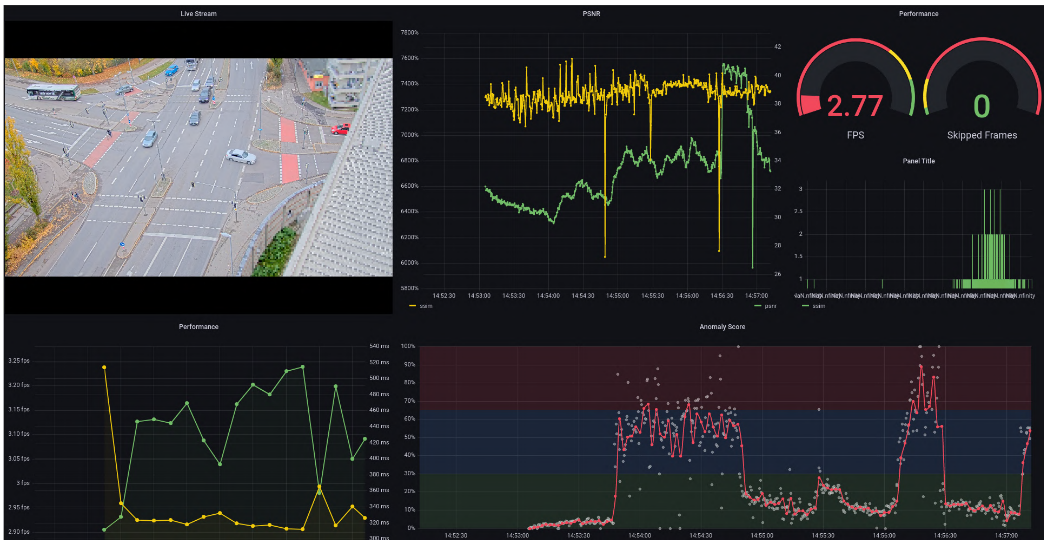


Figure 7.2: Inference visualization using Grafana.

To provide the user with an easy-to-use and understand interface, *Grafana* is leveraged as a dashboard software. The data is provided by an `InfluxDB` (chapter 7.3) instance and the *RTSP*-proxy servers *HLS* interface (chapter 7.1). To play the *HLS* stream the *innius-video-panel* plugin[5] for *Grafana* is used.

A user can easily access the dashboard by navigating to a web address and entering valid access credentials. By default, the dashboard shows a periodically refreshing view of the last five minutes recorded. But the user is also able to specify the desired time range manually. An especially useful feature for analyzing past events.

Figure 7.2 shows a screenshot of the *Grafana* web interface. Raw metrics like PSNR[6] or SSIM[7] are visualized in the topmost panel, while the interpreted anomaly score is visualized in the *Anomaly Score* panel. Additional meta information like performance metrics is also displayed. A user is therefore easily able to check if the model is ill performing or if any issues with the stream occurred.

It needs to be noted, that the displayed *HLS* stream is not completely in sync. This is caused by a limitation with the *HLS* protocol. *HLS* aggregates individual frames from the stream into batches and then transmits them to all stream subscribers. For some applications, this restriction isn't significant but for some, latency may be important. Automatic emergency response dispatch comes to mind. Latency-focused applications can leverage the low-latency *LL-HLS* protocol (Durak et al., 2020), that can serve incomplete video segments to clients. Because this protocol requires a TLS certificate the deployment is slightly more complicated.

[5]`https://github.com/innius/grafana-video-panel`

[6]Peak Signal to Noise Ratio

[7]Structural similarity score

CHAPTER 8

Evaluation

Because of time constraints, a thorough evaluation of the graph neural network architecture mentioned in chapter 6.5 was not feasible. The graph neural network architecture varies significantly from other prediction or reconstruction based approaches and therefore requires additional effort to evaluate.

8.1 Regularity Score Derivation

All previously introduced models generate image data by either reconstructing an input or predicting a future frame. This on its own doesn't help us identify anomalous scenarios. A reliable method for deriving the anomaly score by for example comparing ground truth data with the prediction is required. The easiest method would be to just calculate the distance between ground truth and prediction. As the distance function, either $L1$ norm, $L2$ or even the *Structural Similarity (SSIM)* (see chapter 8.1.1) score can be used. To reduce the image distance values to a probability estimation of

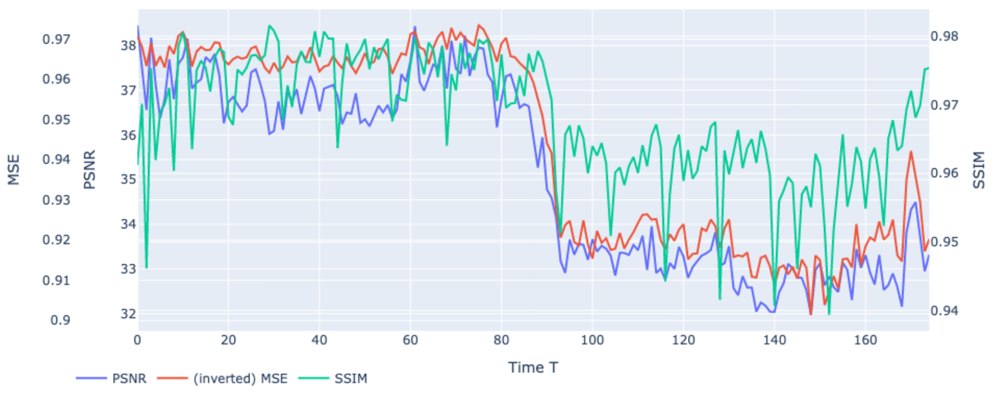


Figure 8.1: A side-by-side visualization of all the distance functions between the ground truth and the predicted frame by the Transformer architecture.

whether a frame is regular or not the distance values get normalized for a window of size T. The normalized probabilities of the image distance values in figure 8.1 are depicted in figure 8.2. Of course, the values produced by each distance function

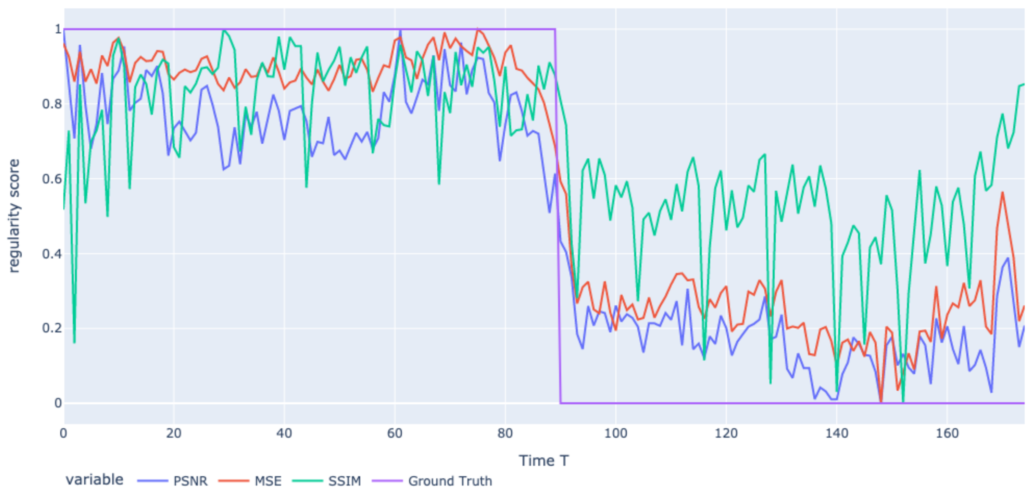


Figure 8.2: Regularity probability calculated based on the previously calculated image distance values.

get also normalized to a fixed interval. Because the system will usually work on a theoretically indefinite stream of numbers a windowed normalization approach is applied.

8.1.1 Choice of Distance Function

As shown in figure 8.1, different distance functions can be leveraged for deriving a regularity score. This inherently creates the problem, of which distance function to choose. For different use cases, there may be different optimal distance functions. This requires a separate evaluation of each distance function for each use case. The evaluation will not only be a single AUROC score but one for each implemented distance function. The models' final performance is the maximum value of all AUROC scores. Implemented distance functions are:

- $L1$ distance between ground truth and prediction
- $L2$ distance between ground truth and prediction
- SSIM score (Wang et al., 2004)

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + c)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}$$

x, y each is a window of $H \times W$
μ_x and μ_y are the pixel sampled mean of x and y
σ_x^2 and σ_y^2 are the variances of x and y
σ_{xy} is the covariance of both x and y x, y in

- Peak Signal to Noise Ratio (PSNR)

$$PNSR(x, y) = 10 * log_{10}(\frac{max(x)^2}{MSE(x, y)})$$

  x, y each is a window of $H \times W$

- Patched Peak Signal to Noise Ratio
- Patched SSIM score
- Patched $L1$ distance

8.1.2 Regime Shifts

Especially noticeable for the FKK use case is a regime shift between halting and flowing traffic. A regime shift is a large and abrupt change in a function or ecosystem, often encountered in the financial domain (Aloui, Hammoudeh, and Hamida, 2015). In our case, the regime shift originates from the model making naturally more mistakes predicting moving objects compared to static ones, as seen in figure 8.4. The effects on the distance function(s) can be observed in figure 8.3. To counteract this

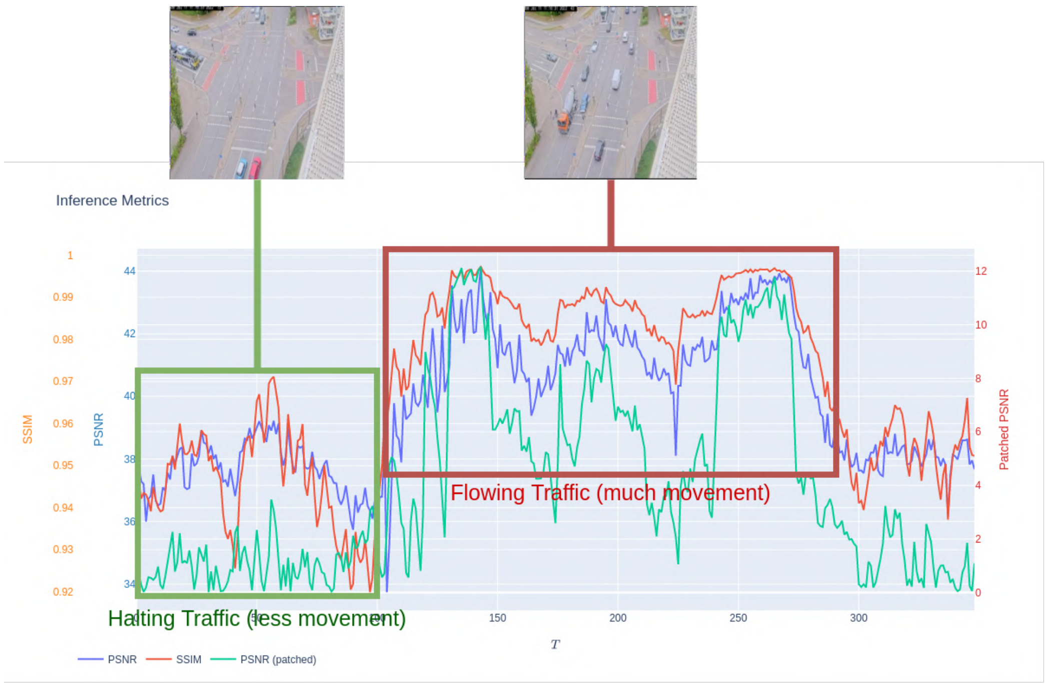


Figure 8.3: Flowing and halting traffic resulting in a regime shift.

behavior the model can be improved by either getting the model to predict moving objects better or identify regime shifts and compensate for them during the regularity score derivation. The first solution is quite challenging because the models' anomaly detection capabilities suffer if the model is too good at predicting $T + 1$ or reconstructing T. Therefore, a statistical approach for detecting regime shifts is required. Such a system is out of scope for this thesis and therefore only mentioned further in chapter 9.2.

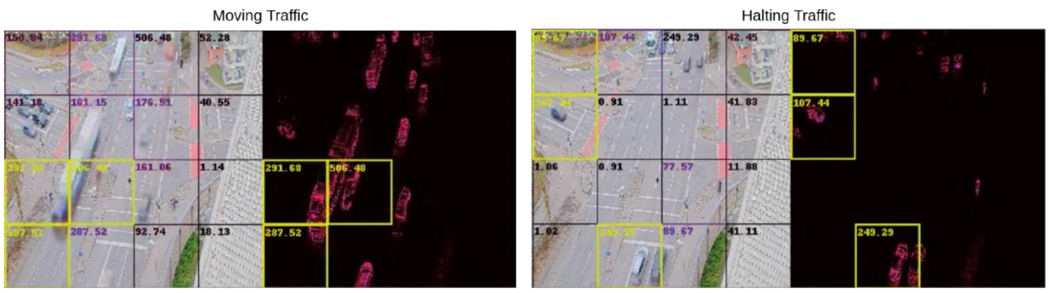


Figure 8.4: The shown example demonstrates that even if there is no anomaly present in the scene, the distance values are significantly higher if there are moving objects in a scene. The visualized distance function is the patched MSE.

8.2 Spatial Temporal Cuboidal Autoencoder

For the evaluation of further anomaly detection architectures covered in this thesis, the architecture discussed in chapter 6.1 inspired by Hasan et al. (2016) is used as a baseline. Not only is this architecture chosen because of its simplicity, but also because it's one of the most cited publications (with 803 combined citations in *Google Scholar*) in the field. Compared to other publications in the video anomaly detection field, the total citation amount is comparable high. This makes this architecture ideal as a baseline for other architectures introduced in chapter 6 of this thesis.

8.2.1 Hyperparameter Tuning

Compared to other architectures, the spatio-temporal cuboidal autoencoder architecture has comparably few hyperparameters to tune. Nevertheless, hyperparameter tuning is still performed to get as close to reaching the architecture's potential as possible. The process of how the hyperparameter search space is performed is described in detail in chapter 2.8. Table 8.1 shows the chosen search space for hyperparameter tuning. Possible values for hyperparameters are purely selected on subjective plausibility. Inspiration is taken from other publications using similar neural network architecture variations. The *Activation Function* hyper-parameter refers to every non-linearity function present in the model except the output activation layer. The last activation heavily affects which data ranges are output by the model and an incorrect choice makes the model incompatible with potential postprocessing operations.

Table 8.1: Spatio-temporal cuboidal autoencoder hyperparameter search space.

Attribute Name	Search Space
Optimizer Type	[Adam, Adagrad, SGD]
Learning Rate	Uniform sampling in range $1e^{-5}$ - $1e^{-2}$
Activation Function	[*ReLU*, *sigmoid*, *tanh*, *LeakyReLU*]

8.2.2 Evaluation on Baseline Datasets

Unfortunately the results achieved in the original publication *Learning Temporal Regularity in Video Sequences* (Fukushima, 1980) could not be reproduced. Evaluation metrics shown in figure A.1 and 8.5 show a clear inability to predict anomalies reliably in both baseline datasets.

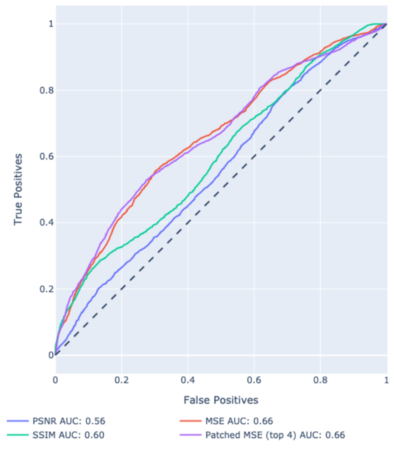


Figure 8.5: Receiver operating characteristic curve achieved on the *UCSD 1* test dataset.

Moving objects are only reconstructed as very blurry shadow-like figures. This is insufficient for deriving an expressive anomaly/regularity score from differences between reconstruction and ground truth. Examples can be observed in figure 8.6 and 8.7. A definitive reason why the results differ so drastically between the implementation of the publication and the one from this thesis is unknown. Especially because the proposed architecture is comparably simple to other architectures that were also evaluated in the context of this thesis. After unsuccessfully reproducing the evaluation results on the *Baseline* datasets, the decision to not further evaluate this architecture on domain-specific datasets was taken. Another factor that solidified this decision was the fact that other approaches like the convolutional autoencoder augmented with *RNN* components like the one described in chapter 6.3 claimed to have more potential than an architecture that solely relies on convolutional layers.

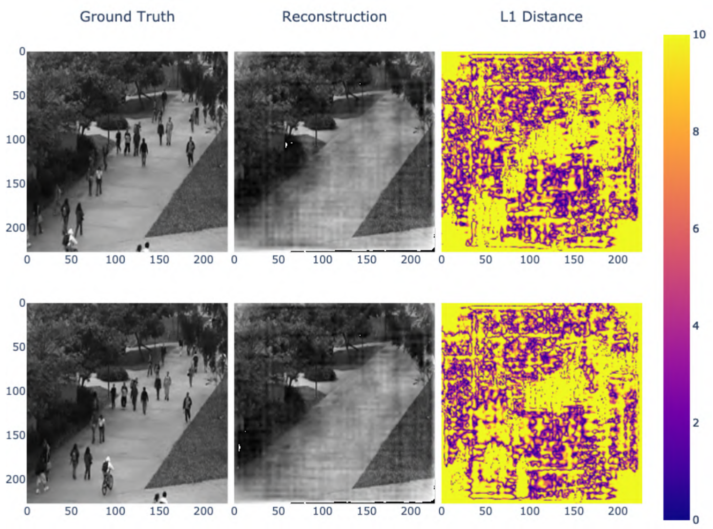


Figure 8.6: Convolutional autoencoder ground truth and reconstruction comparison on the UCSD 1 dataset.

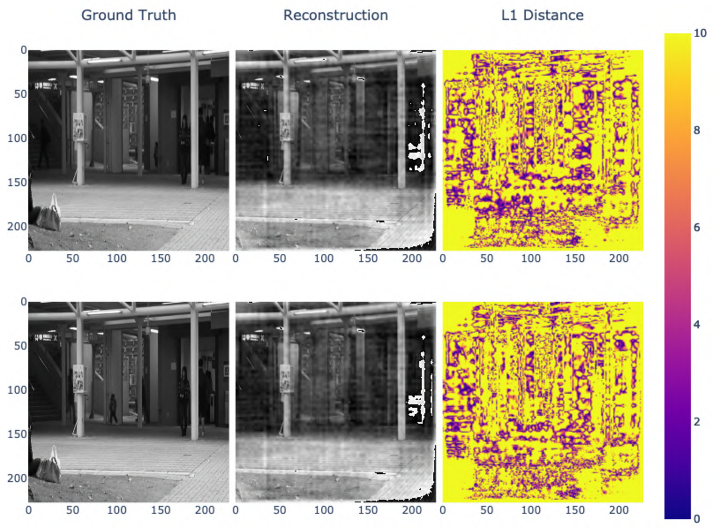


Figure 8.7: Comparison between the model's prediction and corresponding ground truth frame. The first row shows a scene with no anomalous objects or events. The bottom row contains visible anomalous objects.

8.3 FastAno

Compared to the related *Spatio-Temporal* autoencoder architecture described in chapter 6.1, the *FastAno* architecture (6.2) proposed by *FastAno: Fast Anomaly Detection via Spatio-temporal Patch Transformation* (Park et al., 2021) improves upon it by a significant margin. Especially considering its architectural simplicity and very fast learning speeds compared to more complex architecture like the transformer network described in chapter 6.4. The reimplementation of the *FastAno* neural network architecture of this thesis matches the anomaly detection accuracy achieved by the original publication closely[1].

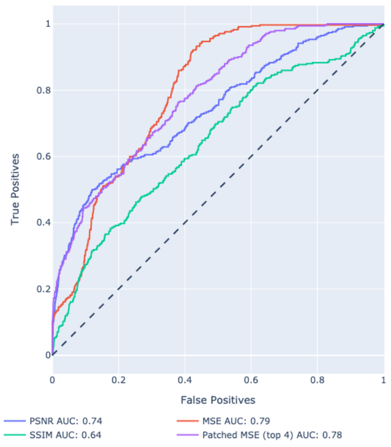


Figure 8.8: Evaluation results of the *FastAno* architecture on the *UCSD Pedestrian 1* dataset.

Qualitative analysis reveals that the model can achieve such good results because anomalous objects and behavior get predicted with significant visual artifacts. These artifacts result in a high response by the image distance function and therefore impact the anomaly/regularity score significantly. For an example refer to figure 6.6. Especially interesting is the value range the difference-heat-map covers. Other approaches typically don't generate any pixels that differ more than $\sim$ 10 pixel values from the ground truth. This approach on the contrary generates pixels that have anomalous pixel differences with a distance of $\sim$ 40. Compared to other approaches, this makes this architecture especially confident in its regularity derivation and therefore the final regularity-score graph considerably less volatile. Examples can be found in the appendix chapter A.2.

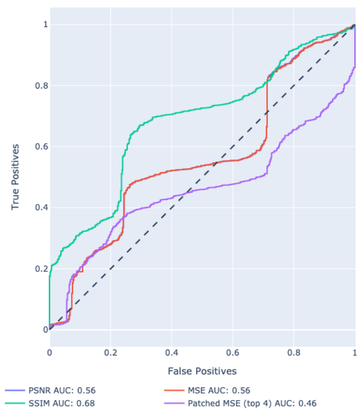


Figure 8.9: Evaluation of the *FastAno* architecture on the *e:fs FKK* dataset.

Experiments on the *Street Scenes* dataset were unfortunately not as successful. The predicted frame is covered with severe artifacts regardless of whether there is an artifact depicted or not. Although it is hard to find a tangible reason why the prediction performance is drastically worse than comparable datasets like *Avenue* or *UCSD Pedestrian 1 & 2*, a plausible reason would be the heavy JPEG compression artifacts present in the *Street Scene* dataset. These artifacts make individual pixels very jittery and therefore hard for the network to predict. Figure 8.10 shows the intense prediction artifacts generated. Applying the architecture on the *e:fs FKK* datasets doesn't yield the same artifacts as described earlier. This furthermore consolidates the conjecture, that JPEG artifacts are responsible for the network's misbehavior. Although the net-

[1]Accuracy tested on the *UCSD Pedestrian 1 & 2* and *Avenue* dataset

work behaves properly (as visible 8.9), anomaly/regularity score inference doesn't work as well as with the *UCSD Pedestrian 1 & 2* and *Avenue* datasets. Especially moving actors are all equally mispredicted by the network and therefore no meaningful regularity assertion can be made (see A.2.3). Typically the results in the regularity score correlate with the overall movement in the frame - a usual issue with reconstruction/prediction approaches. The anomaly graphs for each test clip can be seen in figure A.6.

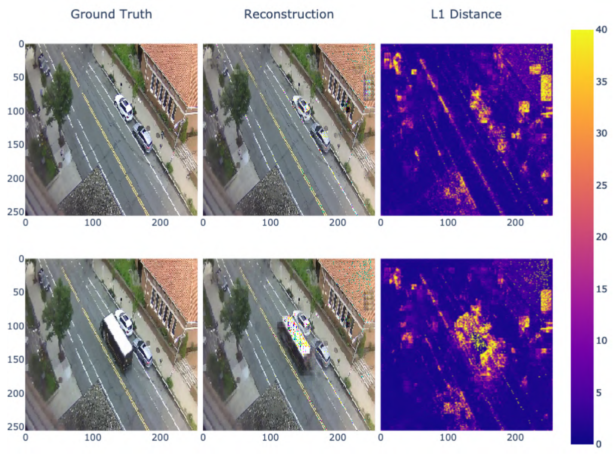


Figure 8.10: Applying the *FastAno* architecture on the *Street Scene* dataset, results in heavy prediction artifacts regardless of whether there is an anomaly present or not.

8.4 Convolutional LSTM Autoencoder

8.4.1 Evaluation on Baseline Datasets

First, a validation versus established state-of-the-art datasets is performed. This confirms whether or not the architecture implementation of this thesis is correct or not. The primary goal is to reproduce the results of the original publications as closely as possible. Therefore, the dataset selection from the publication is matched, so a direct evaluation-accuracy comparison becomes possible.

***UCSD* Pedestrian 1 & 2**

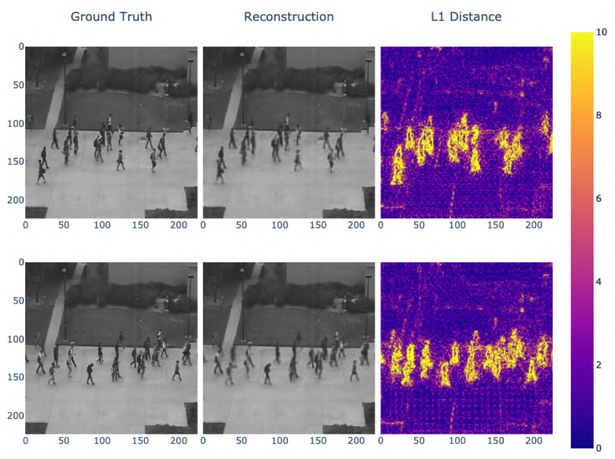


Figure 8.11: Comparison between the model's reconstruction and corresponding ground truth frame.

Compared to the architecture entirely based on two-dimensional convolutional layers, described in chapter 6.1 the proposed LSTM architecture can learn temporal relations between objects more efficiently. This becomes clearly visible by comparing figure 8.11 and 8.6. Actors and their movements are significantly more accurately reconstructed. The improvement can be seen in figure 8.12 which shows an overall improved anomaly detection performance on the *UCSD* dataset. For all individual anomaly score graphs please refer to the appendix section A.3.1. The reconstructed frames are very blurry and therefore the inferred regularity scores very volatile. This makes the model overall not usable for reliable anomaly detection. The improved convolutional autoencoder architecture enhanced with lateral convolutional LSTM connections described in chapter 6.3 improves reconstruction

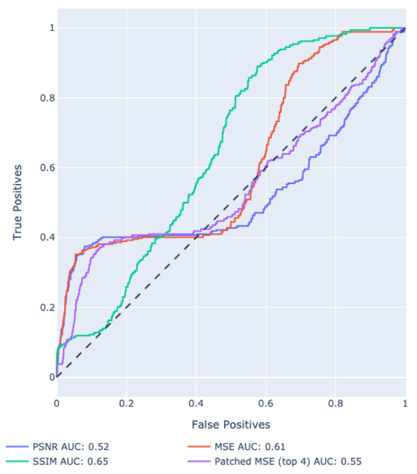


Figure 8.12: Convolutional LSTM Autoencoder Evaluation Results on the *UCSD 1* dataset.

sharpness significantly. Figure 8.13 shows the input and corresponding reconstruction of the proposed architecture. Compared to the same visualization of the previous architecture in figure 8.11, a significant increase in sharpness is noticeable. The reconstruction quality of anomalous objects or behavior is also noticeable but the differences are still present as shown in the *L*1 distance visualization. The more defined differences between input and reconstruction express themselves in a less volatile and more confident regularity score.

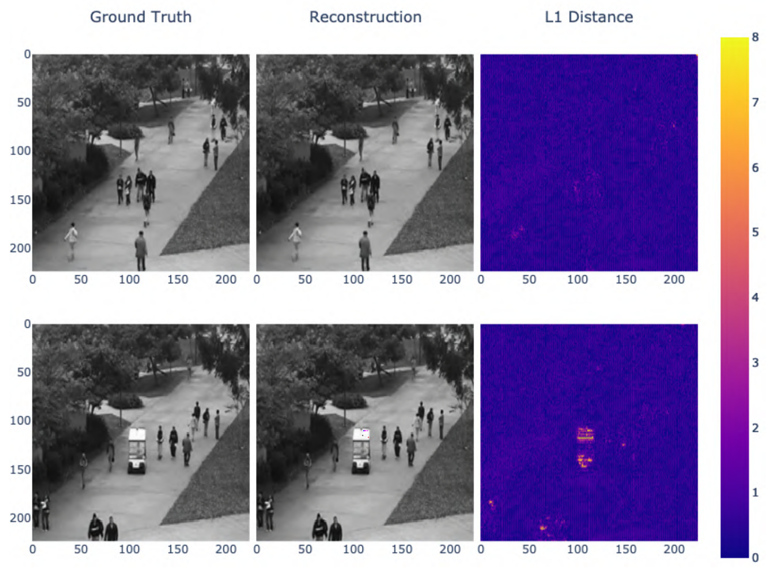


Figure 8.13: Comparison between the lateral convolutional *LSTM* model's reconstruction and corresponding ground truth frame on the *UCSD Pedestrian 1* dataset.

Avenue Dataset

Contrary to the isometric surveillance camera perspective of the *UCSD* dataset, the *Avenue* datasets allows an evaluation on a different perspective. Because the ratio of regular and irregular frames present in the footage *precision-recall curve* is used as a metric. More specifically the area under the *precision-recall curve* (in short *AUPRC*) is better suited for heavily biased data compared to the *AUROC*. Detailed regularity scores for each video clip present in the test dataset can be found in appendix A.12.

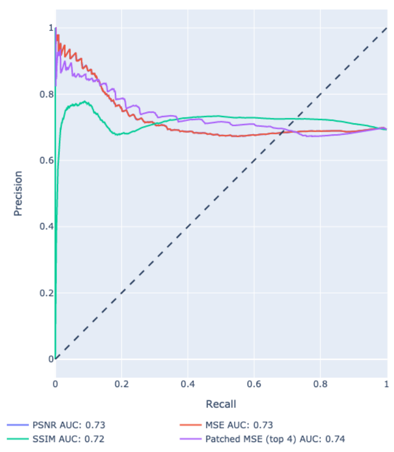


Figure 8.16: Performance convolutional LSTM autoencoder with lateral connection measured by the precision-recall curve.

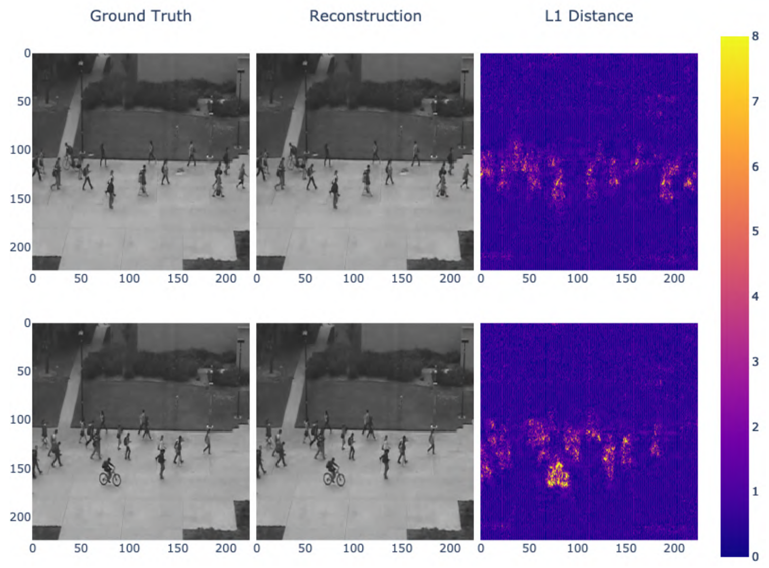


Figure 8.14: Comparison between the lateral convolutional *LSTM* model's reconstruction and corresponding ground truth frame on the *UCSD Pedestrian 2* dataset.

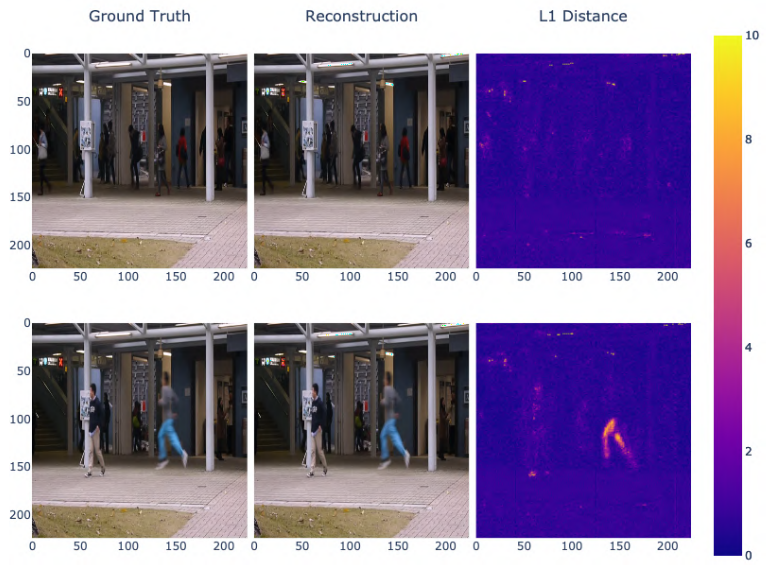


Figure 8.15: Qualitative evaluation of reconstructed frames by the autoencoder with lateral LSTM connections. The upper comparison contains no anomaly and the lower does.

8.4.2 Application to the Road Traffic Domain

Street Scene

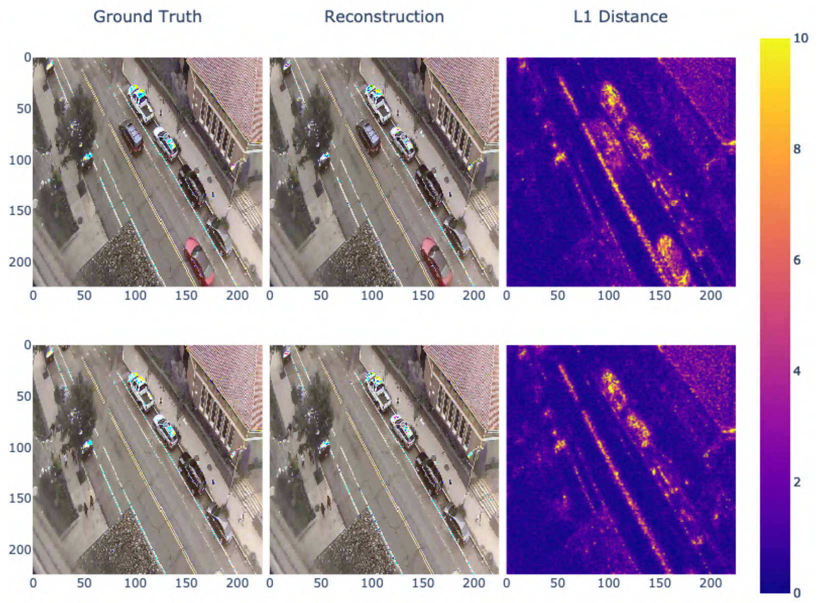


Figure 8.17: Comparison visualization of the convolutional LSTM model's reconstruction and ground truth on the *Street Scene* dataset.

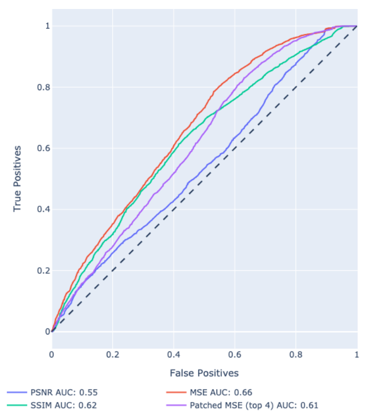


Figure 8.18: Reciever operating characteristic curve achieved on the *Street Scene* dataset by the lateral LSTM convolutional autoencoder architecture.

Contrary to the evaluation results of chapter 8.3, a qualitative analysis suggests that the convolutional LSTM architecture (chapter 6.3) isn't affected much by the JPEG compression artifacts present in the *Street Scene* dataset. Even though the reconstructed frames are reconstructed without any faulty artifacts, anomalies can't be identified by their reconstruction and corresponding ground truth. Naturally, this significantly affects the model's anomaly detection capability negatively. Of course, a purely qualitative analysis isn't sufficient to assess whether the input data quality affects the anomaly detection capabilities. Only because there are no apparent artifacts present in the reconstruction as with the *FastAno* architecture the model can still be held back by the underlying data. Because all machine learning models covered in this thesis are inherently black-box models giving explanations on certain effects or model behavior is highly non-trivial. Contrary to other architectures that use U-Net-inspired feature extractors, the convolutional LSTM architecture produced remarkable sharp reconstructions visible in figure 8.17. Other architectures like the *Transformer* architecture described in chapter 6.4 typically struggle with reconstructing/predicting (fast) moving actors accurately.

FKK Dataset

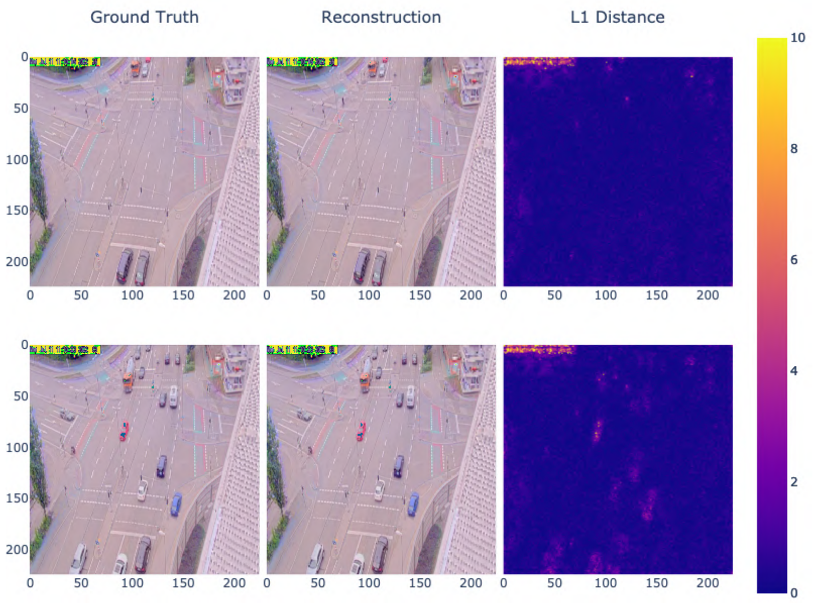


Figure 8.19: Comparison visualization of the convolutional LSTM model's reconstruction and ground truth on the *FKK* dataset.

The lateral *LSTM* architecture can accurately predict the future frame of a given input sequence. Figure 8.19 clearly demonstrates how the model can draw moving and static objects with significant detail. Unfortunately, it also appears that the architecture is unable to capture a deep contextual understanding of the underlying scene. This manifests itself in anomalous behavior being reconstructed just as regular behavior.

8.5 Transformer Architecture

The first experiments performed on baseline datasets suggest very promising anomaly detection performance. Figure 8.20 shows the achieved *AU-ROC* scores on the *UCSD 2* dataset. It's visible which image distance functions (described in chapter 8.1) work the best for the model and dataset. *SSIM* shows significantly worse performance than other used distance functions. Figure 8.20 shows how false positive classifications are a clear weakness of the model compared to false negatives. For the evaluation results on the more challenging *UCSD 1* dataset refer to appendix chapter A.5.1. *UCSD 1* provides more of a challenge for pixel-based anomaly detection systems, because of the perspective distortion of objects. Because objects are at different scales, anomalies in the background have a significantly lower impact on the overall anomaly score.

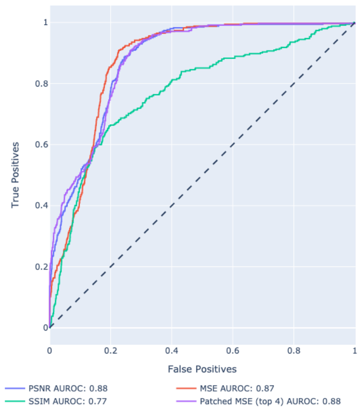


Figure 8.20: Evaluation results on the *UCSD 2* dataset.

8.5.1 Qualitative Evaluation

Because the artifacts generated by all reconstruction-based models can be visualized natively, a qualitative analysis of the generated frames is easily possible. Such an analysis also reveals significantly more information about the model's behavior than a purely quantitative evaluation. The reconstruction comparison in figure 8.21 shows

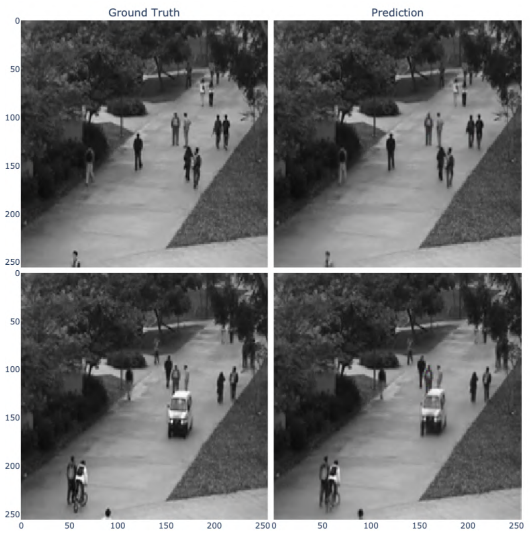


Figure 8.21: Comparison between the model's prediction and corresponding ground truth frame. The first row shows a scene with no anomalous objects or events. The bottom row contains visible anomalous objects (car and cyclist). The model was trained without any frame strides.

that the network's prediction and the corresponding ground truth is almost identical independent of whether an anomaly is present or not. Still, the model can detect anomalous content in the frame(s). It can do so by subtle differences in the predicted frames in contrast to the ground truth. Edges of anomalous objects are not rendered as sharp as others. At closer inspection, even the geometry of some anomalous objects is not reconstructed/predicted correctly. For example, the bicyclist's rear wheel features noticeable reconstruction artifacts. Some generation artifacts are also clearly visible in the car's shadow.

8.5.2 Avenue Dataset

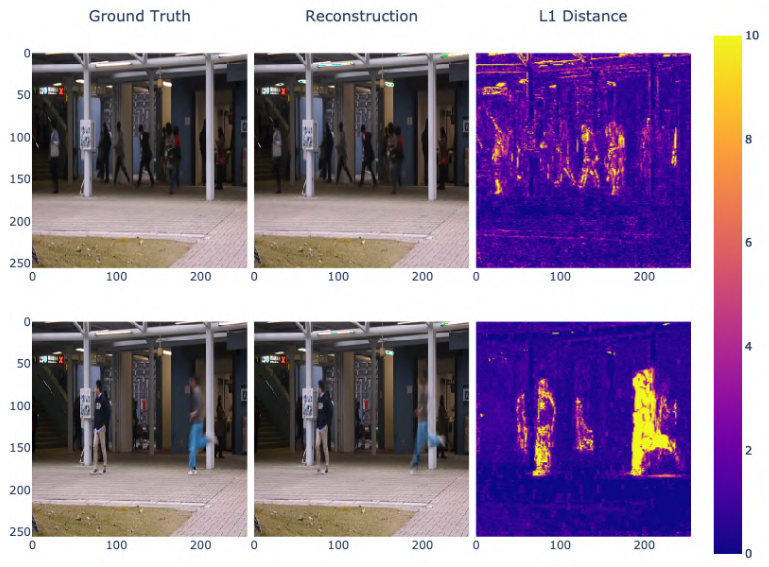


Figure 8.22: Reconstruction and ground truth comparison. The scene depicted in the bottom row shows an anomaly in the form of a person running.

As with the *UCSD Pedestrian* dataset, the architecture can detect irregularities in the video footage reliably. Of course, this approach also suffers from the drawbacks of other anomaly detection techniques that use pixel distance as an indicator of whether an object is anomalous or not. Primarily, small objects that behave/are anomalous unfortunately don't have as much impact as objects that are bigger or closer to the camera. Because footage of the *Avenue* dataset is taken out of a very low perspective, this effect is enhanced greatly.

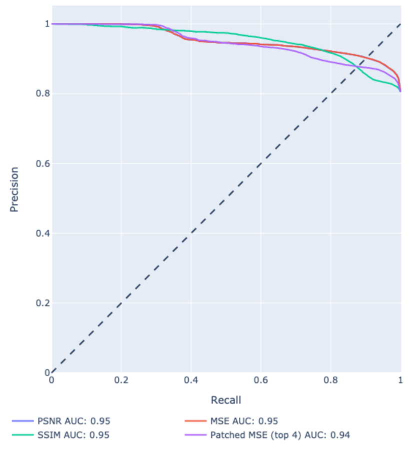


Figure 8.23: Evaluation of the transformer architecture on the *Avenue* test dataset.

8.5.3 Application to the Road Traffic Domain

To evaluate how good the architecture can be applied to the target domain of this thesis, training and evaluation are performed on two different vehicular anomaly detection datasets. First, the model adaptability is tested on the *Street Scene* dataset (chapter 4.1.3) then an evaluation on the target *FKK Dataset* (chapter 4.2) is performed.

Street Scene

The first attempts at training a model on the *Street Scene* dataset resulted in very blurry moving actors. Consequently, the regularity score correlated with the global movement in the scene. The effect is very similar to the one shown in figure 8.11 encountered during the LSTM evaluation (chapter 8.4). Increasing various model parameters for example the internal token dimension size, unfortunately, didn't yield any improvements or changes in behavior. Instead of investing too much time and effort into the *Street Scene* dataset, the decision was made to continue to the *e:fs FKK* dataset. The reasoning behind this decision were the following arguments:

- Because of the heavy and suboptimal compression on the *Street Scene* datasets, results may be not representative of the overall architecture performance.
- The architecture was not evaluated on the *Street Scene* dataset by the publication's authors and therefore no baseline evaluation results are available.

FKK Dataset

Similarly to the previous experiments on the *Street Scene* dataset (chapter 8.5.3) first training runs resulted in models that predict moving actors only as very blurry objects. Because the transformer architecture especially has much more hyper-parameters that can be tuned compared to other architectures covered in this thesis, hyper-parameter tuning is performed. The tuning process is especially time intensive for the *Transformer* architecture because of the complex search space and the implications of the different parameters on the model. To make matters worse, the model converges much slower than other architectures resulting in each training run taking around seven hours (two epochs) on a single A100 *Nvidia* GPU. Table 8.2 shows the hyper-parameter tuning runs sorted by evaluation score. The performance of different model architectures is highly dependent on their parameter configuration as seen by the large spectrum of AUROC evaluation scores.

Table 8.2: Hyper-parameter tuning results for the transformer network architecture performed on the *FKK* dataset.

SSIM AU-ROC	PSNR AU-ROC	PMSE AU-ROC	Token Dim	Recon. Loss Weight	Grad. Loss Weight	Diff. Loss Weight
0.715	0.71	0.312	1024.0	4.68	8.422	1.04
0.699	0.696	0.327	2048.0	4.913	12.561	0.402
0.695	0.698	0.322	1024.0	1.724	5.127	0.421
0.683	0.689	0.331	1024.0	1.646	7.034	0.041
0.679	0.685	0.333	256.0	3.61	7.799	0.466
0.678	0.687	0.335	2048.0	3.525	5.245	0.475
0.668	0.676	0.363	2048.0	0.884	9.514	0.845
0.666	0.671	0.349	256.0	0.803	13.674	0.097
0.656	0.673	0.339	1024.0	1.437	7.341	0.526
0.649	0.668	0.367	256.0	0.887	11.797	0.105
0.647	0.669	0.346	256.0	0.623	7.395	0.781

Continued on next page

Continued from previous page

SSIM AU-ROC	PSNR AU-ROC	PMSE AU-ROC	Token Dim	Recon. Loss Weight	Grad. Loss Weight	Diff. Loss Weight
0.644	0.665	0.363	1024.0	4.123	10.049	0.555
0.643	0.662	0.34	256.0	2.612	14.043	0.118
0.636	0.664	0.363	1024.0	1.804	13.099	0.731
0.633	0.657	0.368	2048.0	4.654	5.975	1.462
0.631	0.652	0.37	1024.0	3.385	5.013	0.975
0.63	0.663	0.374	1024.0	2.323	9.761	1.482
0.62	0.651	0.374	256.0	2.243	13.965	0.071
0.502	0.502	0.502	256.0	1.88	13.236	0.576

After all hyper-parameter tuning runs have finished, a qualitative analysis of the most promising model is performed. A comparison between the ground-truth frame and prediction can be seen in figure 8.25. Similar to other approaches presented in this thesis, any movement captured in the video results in considerable unsharpness. This consequently diminishes the ability of the model to predict anomalies reliably. To rule out any preprocessing operations having negative effects on the model performance, a single model is trained without any preprocessing applied to the input data. The model's hyperparameter configuration is the same as the most promising configuration found during hyperparameter tuning. As expected removing the preprocessing operations only reduced the model's evaluation score by ~ 0.03 *AUROC*.

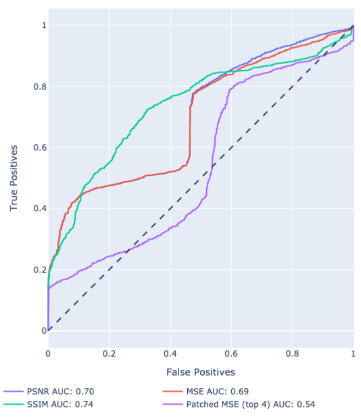


Figure 8.24: Reciever operating characteristic curve over *e:fs FKK* dataset.

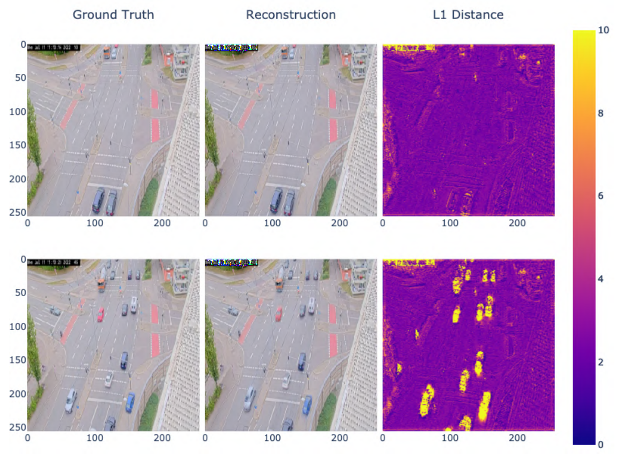


Figure 8.25: Comparison between ground truth and prediction with an anomaly present (bottom row) and without (top row).

8.6 Evaluation Summary

In this chapter, a direct comparison between the model performance of every implemented architecture is done.

Table 8.3: Evaluation scores achieved by each model architecture over all datasets.

Dataset	Spatio-Temporal Cuboidal AE	FastAno	Lateral Conv-LSTM AE	Transformer
UCSD 1	0.72 AUROC	0.77 AUROC	0.72 AUROC	0.88 AUROC
UCSD 2	0.62 AUROC	0.79 AUROC	0.68 AUROC	0.86 AUROC
Avenue	0.80 AUPRC	0.95 **AUPRC**	0.74 AUPRC	0.95 **AUPRC**
Street Scene	N/A	0.48 AUROC	0.66 AUROC	0.47 AUROC
FKK	N/A	0.68 AUROC	0.63 AUROC	0.71 AUROC

Table 8.4: The distance function used to derive the anomaly score for each dataset and model architecture.

Dataset	Spatio-Temporal Cuboidal AE	FastAno	Conv-LSTM AE	Transformer
UCSD 1	MSE	MSE	Patched MSE (top 4)	PSNR / Patched MSE
UCSD 2	MSE	MSE	MSE	PSNR
Avenue	MSE	MSE / PSNR	Patched MSE (top 4)	SSIM
Street Scene	N/A	N/A	MSE	PSNR
FKK	N/A	SSIM	SSIM	SSIM

Table 8.5: Comparison between architectures implemented in this thesis and other state-of-the-art approaches. Values taken directly by the corresponding publication.

Dataset	Spatio-Temporal AE (Hasan et al., 2016)	FastAno (Park et al., 2021)	Conv-LSTM AE (Luo, Liu, and Gao, 2017)	TransAnomaly (Yuan et al., 2021)
UCSD 1	0.81 AUROC	N/A	0.681 AUROC	0.840 AUROC
UCSD 2	0.90 AUROC	96.3 AUROC	0.811 AUROC	0.964 AUROC
Avenue	0.702 AUROC	85.3 AUROC	0.745 AUROC	0.870 AUROC
Street Scene	N/A	N/A	N/A	N/A

In summary, none of the reconstruction-based approaches resulted in a neural network model that can detect anomalies on the *e:fs FKK* dataset reliably. The anomaly detection performance on baseline datasets like *UCSD* or *Avenue* is close to other state-of-the-art publications. It stands to reason, that the proposed architectures are not able to learn a deep semantic understanding of a scene and its actors.

Most approaches appear to only work on a relatively crude pixel level and are unable to capture actor interactions efficiently. The two most promising image-based architectures are *FastAno* for its great performance (both in terms of speed and accuracy) on baseline datasets and the *Transformer* architecture because of its potential to learn deep temporal and spatial relations. Further research on applying approaches that work on a higher abstraction level, like the graph neural network architecture seems promising for use cases that require a high level of semantical understanding of actor interactions.

8.7 Runtime Performance

Anomaly detection is often used as a technique to identify interesting excerpts from a large amount of data. Consequently, a fast inference time is very important for most use cases. For video anomaly detection especially, possible real-time detection on low-powered hardware can be very important. If a model/architecture is real-time capable, a continuous stream can be assessed by the system without any intermediate storage required. For testing the performance of different architectures, each trained[2] model gets executed with the ONNX-Runtime framework for 60 seconds. The average inference time for a single frame is used as an indicator of execution speed. Of course, execution times are highly relative to the hardware the model gets executed on. That's why two different GPUs are tested separately. One is a high-end Nvidia A100 card especially targeted toward deep learning and the second is a consumer-grade CUDA-capable graphics card. The latter could be realistically used for an edge computing[3] setup. Lastly, an additional benchmark on a consumer-grade CPU[4] is performed which shows how each architecture performs on unspecialized hardware. Because of layer incompatibilities between ONNX-Runtime and the *PyTorch* MaxUnpool2D layer a representative comparison was not possible. Because of the architectural similarity between the *Spatio-Temporal* autoencoder and *FastAno* architecture the performance should be comparable.

Table 8.6: Comparison between architectures implemented in this thesis and other state-of-the-art approaches. Data loading and preprocessing operations are not measured for all architectures.

Model	CPU	Nvidia A100
Spatio-Temporal	N/A	N/A
FastAno	180 Hz	204 Hz
Conv-LSTM	100 Hz	1600 Hz
Transformer	1 Hz	153 Hz

[2]Theoretically, whether a model is trained shouldn't make any difference for performance analysis.

[3]Edge computing in contrast to cloud computing, roughly describes the execution of computationally intensive tasks on the local device instead of sending the input data to the cloud and leveraging the computation power of remote servers.

[4]Intel Core i7 6 Cores @ 2,2 GHz

Table 8.6 shows that the *FastAno* and *ConvLSTM* networks are easily executable on a medium-powered CPU. Especially noticeable is the *Transformer* architecture as a significant outlier that requires a GPU to be executed productively.

CHAPTER 9

Conclusion

9.1 Anomaly Detection System

Unfortunately, the results of this thesis suggest, that current pixel-based anomaly detection neural network architectures are not particularly suited for analyzing complex behavior. Other approaches (like chapter 6.5) that work on an abstraction layer should more easily learn behavior patterns. Abstraction also introduces additional complexity to the neural network design process and more research has to be conducted in this field. For example, the choice of which type of abstraction one would use for a certain use case and how this abstraction is generated is highly non-trivial. Nevertheless are pixel-based anomaly detection systems very well suited for use cases that don't require much high-level understanding and are focused on very distinct anomalies and fast inference times.

9.2 Future Work

9.2.1 Open Source Release

As already mentioned in chapter 9.3, *e:fs* as the intellectual property owner plans to allow the project and the associated anomaly detection framework to be published as open source software. To confine with company policies some parts of the project still require additions like copyright headers and approval by the legal department. With an open-source release also comes further development and maintenance of the framework.

Optimizing Already Present Neural Network Architectures

Currently implemented neural network architectures still leave much room for implementation fine-tuning. Because of the time constraints of this thesis and the comparably small performance gains that are theoretically possible, fine-tuning the implementations was not feasible in the context of this thesis. Publications like *Yolo* (Redmon et al., 2015) show that small implementations should not be neglected while searching for small performance gains.

Supporting new State-of-the-Art Architectures

The anomaly detection research field is still very active and there are plenty of new publications that claim to improve upon the approaches already implemented in conjunction with the thesis. Some are only slight deviations from currently implemented approaches like Lee, Nam, and Lee (2022) improving upon the transformer architecture described in 6.4 utilizing optical flow-based components. The required effort to implement such a model architecture is fairly low because of already existing elements present in the framework.

9.2.2 Combining Anomaly Detection with Actor Trajectory Extraction

To create the end-to-end system described in chapter 1.1, *e:fs* will organize a student project that'll try to fuse a real-time anomaly detection system and the actor trajectory algorithm (Strosahl et al., 2022). The planned start date for the student project is the 1st of October 2022 and will last six months.

9.3 Open Source Anomaly Framework

Unfortunately, there are not many publicly available machine-learning frameworks that allow users to use state-of-the-art anomaly detection neural network architectures in an accessible way. Creating such a framework would be a valuable contribution to the machine learning research space and fill a clear white-spot in the industry. That's why this thesis not only produced multiple methods of detecting anomalies in the traffic domain but also provides a framework for anomaly detection that applies to other domains as well. Through its modular design, one can easily adapt or extend implemented model architectures to fit a specific use case. For uses, where no major architecture design changes are required and access to the hyper-parameter is enough, a simple model/training configuration file is sufficient.

```
{
    "input_size": [256, 256],
    "model_type": "transformer",
    "dataset": "ucsd",
    "skip_frames": 1,
    "lr": 0.01,
    "decay": 0.0005,
    "optimizer": "adagrad",
    "motion_threshold": 20,
    "batch_size": 4,
    "timesteps": 5,
    "model_parameters": {
        "latent_dim": 2048,
        "token_dim": 256,
        "gradient_loss_weight": 2,
        "reconstruction_loss_weight": 0.65,
        "discriminator_loss_weight": 9,
        "difference_loss_weight": 7.2,
        "temporal_transformer_layers": 1,
        "spatial_transformer_layers": 3
    }
}
```

With the configuration file created, one can simply invoke

```
train --config ./custom-config.json
```

to start a training run. To register single or multiple custom datasets to train a model on a `pytorch_lightning` dataset class has to be implemented. A thorough publicly hosted web documentation (see 2.6) is additionally provided. The documentation contains information on how to train a custom model and generated documentation on Python classes used by the framework. The whole code base of this thesis will be publicly available after the publication of this thesis as soon as the project passes all open source requirements established by *e:fs TechHub* GmbH. The license under which the framework will be published will be Apache-2.0[1]. This allows users to use the framework even for commercial purposes.

All the code necessary to reproduce the presented results in this thesis is published as a public repository hosted on github.com. The project is both hosted under the author's name-space (SirBubbls/traffic-anomaly-detection) and simultaneously as a fork under the *e:fs TechHub's* open source name-space (github.com/EFS-OpenSource) and SAVeNoW (github.com/savenow). Third-party resources are not hosted in the repository but need to be installed manually or semi-manually. All freely available datasets used in this thesis can be easily downloaded with a provided `Makefile`. Some datasets require permission to download and therefore can't be downloaded without any authentication.

9.3.1 Ease of Use

During this thesis, the project was migrated and set up on different machines multiple times. Because of the utilized technologies for managing the project and dependencies as described in chapter 2.1, deployment was very easy. No manual installation of dependencies was necessary besides the correct *Nvidia CUDA* drivers if there are any GPUs available to the system.

9.4 Personal Retrospective

At last, I want to give my personal opinions on things that went well and things I would do differently in retrospect. The project's setup and the utilized tooling

[1]`https://www.apache.org/licenses/LICENSE-2.0`

proved to be very solid and reliable. Poetry (chapter 2.1) proved to be very useful in providing reliable behavior across different systems. This is accomplished by having hard compatibility constraints on dependency and sub-dependency module versions. *Anaconda* or *miniconda* still have their right of existence, because they manage *CUDA* environments way better than *Poetry* does. This means, that if developers don't all have access to *CUDA* capable GPUs, *anaconda* would probably be a better choice for dependency management. The additional *PyTest* CI environment caught various errors before deployment or a merge to the main line. Especially integration issues that originated from changes made to low-level modules were caught reliably. Because detecting such issues is very hard to do manually, the automated setup proved itself to be very valuable.

PyTorch and its additional abstraction layers like *PyTorch Lightning* were also a good choice, but in retrospect using the *PyTorch* abstraction layers from the beginning would have saved some time during development. Most frameworks like *PyTorch Lightning* use the same APIs as vanilla *PyTorch* anyways and therefore no development overhead is introduced. Although migrating from vanilla *PyTorch* to a framework like *PyTorch Lightning* can be time costly. For this project *PyTorch Lightning* was indispensable because of their provided distributed training implementations described thoroughly in chapter 2.7.

Regarding the machine learning models and their architectures, it was quite surprising how few fleshed-out implementations of already existing video anomaly detection architecture proposals are available by either the corresponding authors or third-party "community" implementations. This of course made the evaluation of architectures in the vehicle traffic video surveillance domain significantly more difficult and time intensive. Architectures had to be implemented first, checked for "correctness" and only then an evaluation and experiments on the target domain could be performed. As a consequence only around 20% of the overall work effort went into getting the architecture to work with the target use case and the rest into implementing and evaluating proposed architectures for "correctness". This is also precisely the reason why the decision was made to make all architecture implementations of this thesis not only available to the public, but also make them comfortable to use for other researchers.

Reflecting on the various model architectures that were implemented and evaluated in the context of this thesis, a faster pivot towards architectures that work on higher-level data as with the graph neural network described in chapter 6.5 probably would have been better. Ultimately working on the pixel level has too many drawbacks that are hard limits of the medium itself. Chapters 6.1.3 and 6.1.3 come to mind. Of course, creating an abstraction layer from the video-frame level isn't trivial either and introduces significant overhead but in retrospect would have probably been more expedient.

Writing the thesis and the associated scientific work was done with a non-trivial setup which proved itself to be very efficient. Instead of writing pure *LaTeX*, the markup language *Org* was used and transpiled to *LaTeX* using Emacs and then further processed using a typical *LaTeX* pipeline. *Org* in combination with *org-babel* allows for literate programming similar to a *Jupyter* notebook environment. This means that the sources of all generated graphs and especially evaluation visualizations are embedded into the source file of this thesis and they are automatically regenerated as soon as the underlying data changes. Therefore, the results presented in this thesis are easily reproducible and the underlying data can be accessed by others on demand.

Bibliography

Aboah, Armstrong et al. (2021). *A Vision-Based System for Traffic Anomaly Detection Using Deep Learning and Decision Trees.* arXiv: `2104.06856 [cs.CV]`.

Adi, Kusworo et al. (May 2018). "Automatic Vehicle Counting Using Background Subtraction Method on Gray Scale Images and Morphology Operation". In: *Journal of Physics: Conference Series* 1025, p. 012025. DOI: `10.1088/1742-6596/1025/1/012025`.

Aloui, Chaker, Shawkat Hammoudeh, and Hela Ben Hamida (2015). "Price discovery and regime shift behavior in the relationship between sharia stocks and sukuk: A two-state Markov switching analysis". In: *Pacific-Basin Finance Journal* 34, pp. 121–135.

Arnab, Anurag et al. (2021). "Vivit: A video vision transformer". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 6836–6846.

Baldi, Pierre (2012). "Autoencoders, unsupervised learning, and deep architectures". In: *Proceedings of ICML workshop on unsupervised and transfer learning.* JMLR Workshop and Conference Proceedings, pp. 37–49.

Bergstra, James and Yoshua Bengio (2012). "Random Search for Hyper-Parameter Optimization". In: *J. Mach. Learn. Res.* 13, pp. 281–305.

Bradley, Andrew P. (1997). "The use of the area under the ROC curve in the evaluation of machine learning algorithms". In: *Pattern Recognition* 30.7, pp. 1145–1159. ISSN: 0031-3203. DOI: `https://doi.org/10.1016/S0031-3203(96)00142-2`. URL: `https://www.sciencedirect.com/science/article/pii/S0031320396001422`.

Brown, Tom B. et al. (2020). *Language Models are Few-Shot Learners.* DOI: `10.48550/ARXIV.2005.14165`. URL: `https://arxiv.org/abs/2005.14165`.

Chandola, Varun, Arindam Banerjee, and Vipin Kumar (July 2009a). "Anomaly Detection: A Survey". In: *ACM Comput. Surv.* 41. DOI: `10.1145/1541880.1541882`.

— (2009b). "Anomaly detection: A survey". In: *ACM computing surveys (CSUR)* 41.3, pp. 1–58.

Chen, Cen et al. (2019). "Gated residual recurrent graph neural networks for traffic prediction". In: *Proceedings of the AAAI conference on artificial intelligence.* Vol. 33. 01, pp. 485–492.

Durak, Kerem et al. (2020). “Evaluating the performance of apple’s low-latency HLS”. In: *2020 IEEE 22nd International Workshop on Multimedia Signal Processing (MMSP)*. IEEE, pp. 1–6.

Fu, Zhouyu, Weiming Hu, and Tieniu Tan (2005). “Similarity based vehicle trajectory clustering and anomaly detection”. In: *IEEE International Conference on Image Processing 2005*. Vol. 2. Ieee, pp. II–602.

Fukushima, Kunihiko (1980). “Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position”. In: *Biological Cybernetics* 36, pp. 193–202.

Gidaris, Spyros, Praveer Singh, and Nikos Komodakis (2018). *Unsupervised Representation Learning by Predicting Image Rotations*. DOI: 10.48550/ARXIV.1803.07728. URL: https://arxiv.org/abs/1803.07728.

Gong, Yuan, Yu-An Chung, and James Glass (2021). “Ast: Audio spectrogram transformer”. In: *arXiv preprint arXiv:2104.01778*.

Haresh, Sanjay et al. (2020). *Towards Anomaly Detection in Dashcam Videos*. DOI: 10.48550/ARXIV.2004.05261. URL: https://arxiv.org/abs/2004.05261.

Hasan, Mahmudul et al. (2016). *Learning Temporal Regularity in Video Sequences*. DOI: 10.48550/ARXIV.1604.04574. URL: https://arxiv.org/abs/1604.04574.

He, Kaiming et al. (2017). *Mask R-CNN*. DOI: 10.48550/ARXIV.1703.06870. URL: https://arxiv.org/abs/1703.06870.

Hochreiter, Sepp and Jürgen Schmidhuber (Dec. 1997). “Long Short-term Memory”. In: *Neural computation* 9, pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

Kingma, Diederik P. and Jimmy Ba (2014). *Adam: A Method for Stochastic Optimization*. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

Lecun, Y. et al. (1998). “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11, pp. 2278–2324. DOI: 10.1109/5.726791.

Lee, Joo-Yeon, Woo-Jeoung Nam, and Seong-Whan Lee (2022). *Multi-Contextual Predictions with Vision Transformer for Video Anomaly Detection*. DOI: 10.48550/ARXIV.2206.08568. URL: https://arxiv.org/abs/2206.08568.

Li, Yingying et al. (June 2020a). “Multi-Granularity Tracking with Modularlized Components for Unsupervised Vehicles Anomaly Detection”. In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. Seattle, WA, USA: IEEE, pp. 2501–2510. ISBN: 978-1-72819-360-1. DOI: 10.1109/CVPRW50498.2020.00301. URL: https://ieeexplore.ieee.org/document/9150675/ (visited on 03/05/2022).

Li, Zewen et al. (2020b). *A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects*. DOI: 10.48550/ARXIV.2004.02806. URL: https://arxiv.org/abs/2004.02806.

Lin, Tsung-Yi et al. (2014). *Microsoft COCO: Common Objects in Context*. DOI: 10.48550/ARXIV.1405.0312. URL: https://arxiv.org/abs/1405.0312.

Liu, Wen et al. (2018). “Future frame prediction for anomaly detection–a new baseline”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 6536–6545.

Luo, Weixin, Wen Liu, and Shenghua Gao (2017). “Remembering history with convolutional LSTM for anomaly detection”. In: *2017 IEEE International Conference on Multimedia and Expo (ICME)*, pp. 439–444. DOI: 10.1109/ICME.2017.8019325.

Malhotra, Pankaj et al. (2016). “LSTM-based encoder-decoder for multi-sensor anomaly detection”. In: *arXiv preprint arXiv:1607.00148*.

Montanari, Francesco et al. (2021). “Maneuver-Based Resimulation of Driving Scenarios Based on Real Driving Data”. In: *2021 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1124–1131. DOI: 10.1109/IV48863.2021.9575441.

Nagi, Jawad et al. (Nov. 2011). “Max-pooling convolutional neural networks for vision-based hand gesture recognition”. In: pp. 342–347. DOI: 10.1109/ICSIPA.2011.6144164.

Naphade, Milind et al. (2018). “The 2018 NVIDIA AI City Challenge”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 53–537. DOI: 10.1109/CVPRW.2018.00015.

Naphade, Milind et al. (June 2020). “The 4th AI City Challenge”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pp. 2665–2674.

Narayanan, Sandeep Nair, Sudip Mittal, and Anupam Joshi (2016). “OBD_SecureAlert: An anomaly detection system for vehicles”. In: *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, pp. 1–6.

Pang, Guansong et al. (2021). “Deep learning for anomaly detection: A review”. In: *ACM Computing Surveys (CSUR)* 54.2, pp. 1–38.

Park, Chaewon et al. (2021). *FastAno: Fast Anomaly Detection via Spatio-temporal Patch Transformation.* DOI: 10.48550/ARXIV.2106.08613. URL: https://arxiv.org/abs/2106.08613.

Pourreza, Masoud, Mohammadreza Salehi, and Mohammad Sabokrou (2021). *Ano-Graph: Learning Normal Scene Contextual Graphs to Detect Video Anomalies.* DOI: 10.48550/ARXIV.2103.10502. URL: https://arxiv.org/abs/2103.10502.

Pradhyumna, P, GP Shreya, et al. (2021). “Graph neural network (GNN) in image and video understanding using deep learning for computer vision applications”. In: *2021 Second International Conference on Electronics and Sustainable Communication Systems (ICESC)*. IEEE, pp. 1183–1189.

Radford, Alec et al. (2019). “Language Models are Unsupervised Multitask Learners”. In.

Ramachandra, Bharathkumar and Michael Jones (2020). *Street Scene: A new dataset and evaluation protocol for video anomaly detection.* arXiv: 1902.05872 [cs.CV].

Redmon, Joseph et al. (2015). *You Only Look Once: Unified, Real-Time Object Detection.* DOI: 10.48550/ARXIV.1506.02640. URL: https://arxiv.org/abs/1506.02640.

Ren, Shaoqing et al. (2015a). “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.” In: *NIPS*. Ed. by Corinna Cortes et al., pp. 91–99. URL: http://dblp.uni-trier.de/db/conf/nips/nips2015.html#RenHGS15.

Ren, Shaoqing et al. (2015b). *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.* DOI: 10.48550/ARXIV.1506.01497. URL: https://arxiv.org/abs/1506.01497.

Ronneberger, Olaf, Philipp Fischer, and Thomas Brox (2015). *U-Net: Convolutional Networks for Biomedical Image Segmentation.* DOI: `10.48550/ARXIV.1505.04597`. URL: `https://arxiv.org/abs/1505.04597`.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (1986). "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations.* Ed. by David E. Rumelhart and James L. Mcclelland. Cambridge, MA: MIT Press, pp. 318–362.

Salas, Joaquin et al. (2007). "Detecting Unusual Activities at Vehicular Intersections". In: *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pp. 864–869. DOI: `10.1109/ROBOT.2007.363094`.

Samuel, Dinesh Jackson and Fabio Cuzzolin (2021). "SVD-GAN for Real-Time Unsupervised Video Anomaly Detection". In.

Schmidhuber, Jürgen (2015). "Deep learning in neural networks: An overview". In: *Neural Networks* 61, pp. 85–117. DOI: `10.1016/j.neunet.2014.09.003`. URL: `https://doi.org/10.1016%2Fj.neunet.2014.09.003`.

Shi, Xingjian et al. (2015). *Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting.* DOI: `10.48550/ARXIV.1506.04214`. URL: `https://arxiv.org/abs/1506.04214`.

Song, Tengfei et al. (2021). "Uncertain graph neural networks for facial action unit detection". In: *Proceedings of the AAAI Conference on Artificial Intelligence.* Vol. 35. 7, pp. 5993–6001.

Strosahl, Julian et al. (2022). "Perspective-Corrected Extraction of Trajectories from Urban Traffic Camera Using CNN". In: *2022 International Conference on Connected Vehicle and Expo (ICCVE)*, pp. 1–7. DOI: `10.1109/ICCVE52871.2022.9742966`.

Szeliski, Richard (2011). *Computer vision algorithms and applications.* URL: `http://dx.doi.org/10.1007/978-1-84882-935-0`.

Vaswani, Ashish et al. (2017). *Attention Is All You Need.* DOI: `10.48550/ARXIV.1706.03762`. URL: `https://arxiv.org/abs/1706.03762`.

Veličković, Petar et al. (2018). *Deep Graph Infomax.* DOI: `10.48550/ARXIV.1809.10341`. URL: `https://arxiv.org/abs/1809.10341`.

Wang, Wenguan et al. (2019). "Zero-shot video object segmentation via attentive graph neural networks". In: *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 9236–9245.

Wang, Xuanzhao et al. (2020). *Robust Unsupervised Video Anomaly Detection by Multi-Path Frame Prediction.* DOI: `10.48550/ARXIV.2011.02763`. URL: `https://arxiv.org/abs/2011.02763`.

Wang, Zhou et al. (2004). "Image quality assessment: from error visibility to structural similarity". In: *IEEE Transactions on Image Processing* 13.4, pp. 600–612. DOI: `10.1109/TIP.2003.819861`.

Wu, Jie et al. (June 2021a). "Box-Level Tube Tracking and Refinement for Vehicles Anomaly Detection". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pp. 4112–4118.

— (2021b). "Box-Level Tube Tracking and Refinement for Vehicles Anomaly Detection". In: *2021 IEEE/CVF Conference on Computer Vision and Pattern Recog-*

nition Workshops (CVPRW), pp. 4107–4113. DOI: `10.1109/CVPRW53098.2021.00464`.

Wu, Yuxin et al. (2019). *Detectron2*. `https://github.com/facebookresearch/detectron2`.

Wu, Zonghan et al. (2020). "A comprehensive survey on graph neural networks". In: *IEEE transactions on neural networks and learning systems* 32.1, pp. 4–24.

Yuan, Hongchun et al. (2021). "TransAnomaly: Video Anomaly Detection Using Video Vision Transformer". In: *IEEE Access* 9, pp. 123977–123986. DOI: `10.1109/ACCESS.2021.3109102`.

Zaheer, Muhammad Zaigham et al. (2020). *Old is Gold: Redefining the Adversarially Learned One-Class Classifier Training Paradigm.* DOI: `10.48550/ARXIV.2004.07657`. URL: `https://arxiv.org/abs/2004.07657`.

Zhao, Hang et al. (2017). "Loss Functions for Image Restoration With Neural Networks". In: *IEEE Transactions on Computational Imaging* 3.1, pp. 47–57. DOI: `10.1109/TCI.2016.2644865`.

Zhao, Yuxiang et al. (2021). *Good Practices and a Strong Baseline for Traffic Anomaly Detection.* arXiv: `2105.03827 [cs.CV]`.

Zhong, Jia-Xing et al. (2019). "Graph convolutional label noise cleaner: Train a plug-and-play action classifier for anomaly detection". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 1237–1246.

Zou, Zhengxia et al. (2019). "Object detection in 20 years: A survey". In: *arXiv preprint arXiv:1905.05055*.

APPENDIX A

Appendix - Supplementary Evaluation Material

A.1 Spatio Temporal Convolutional Autoencoder

A.1.1 Avenue

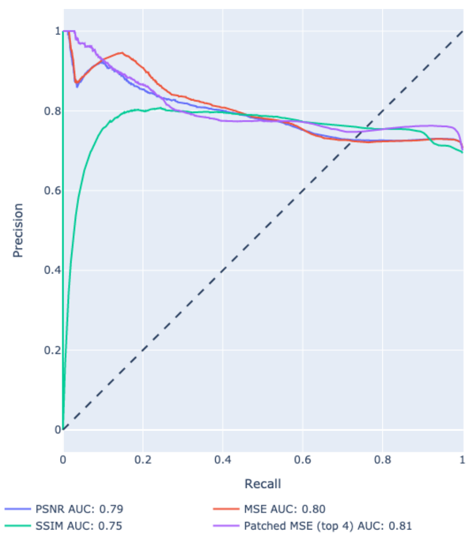


Figure A.1: Precision recall curve achieved on the *Avenue* test dataset.

A.1.2 UCSD

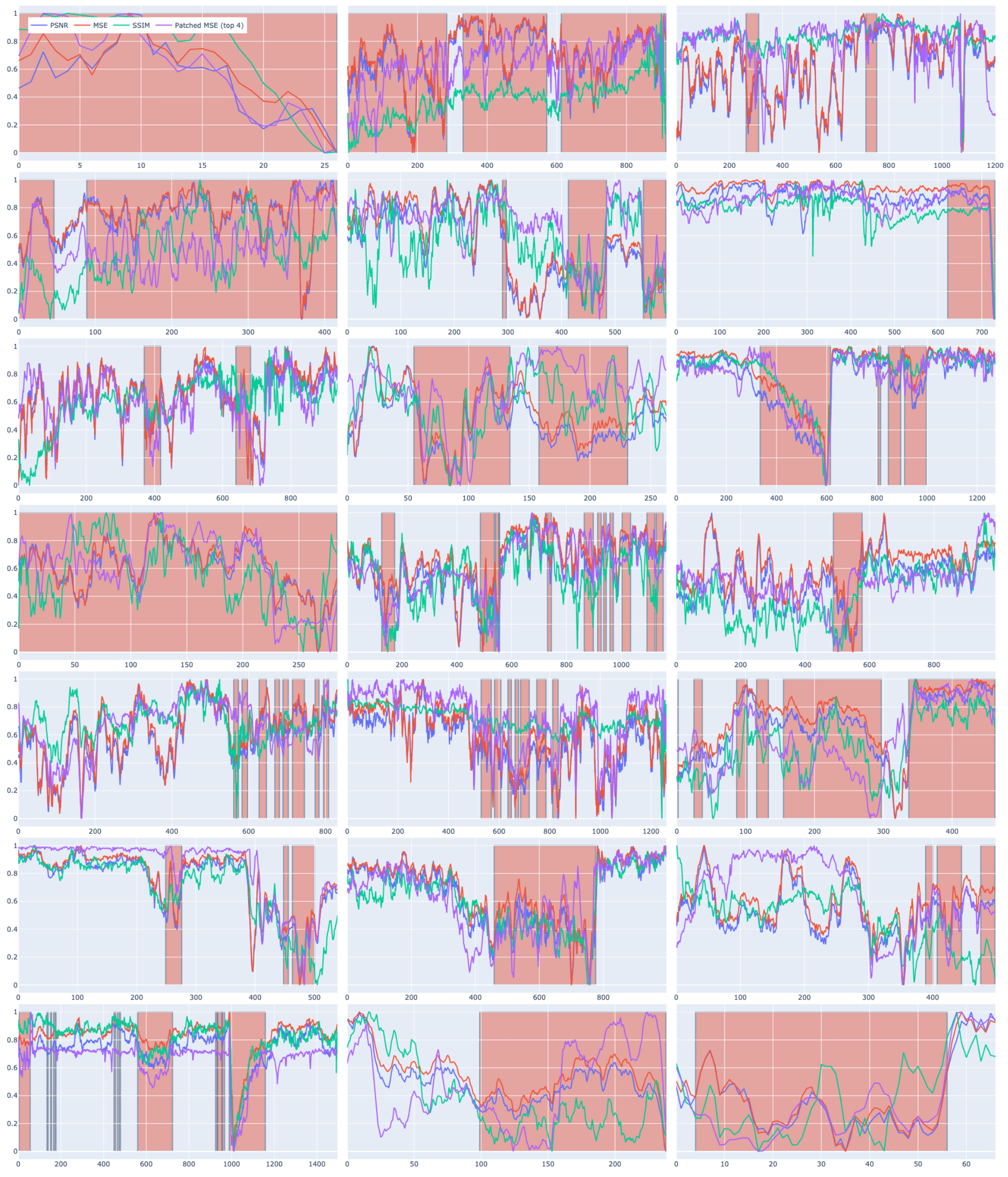


Figure A.2: Spatio Temporal Convolutional Autoencoder evaluation results on the *Avenue* test dataset.

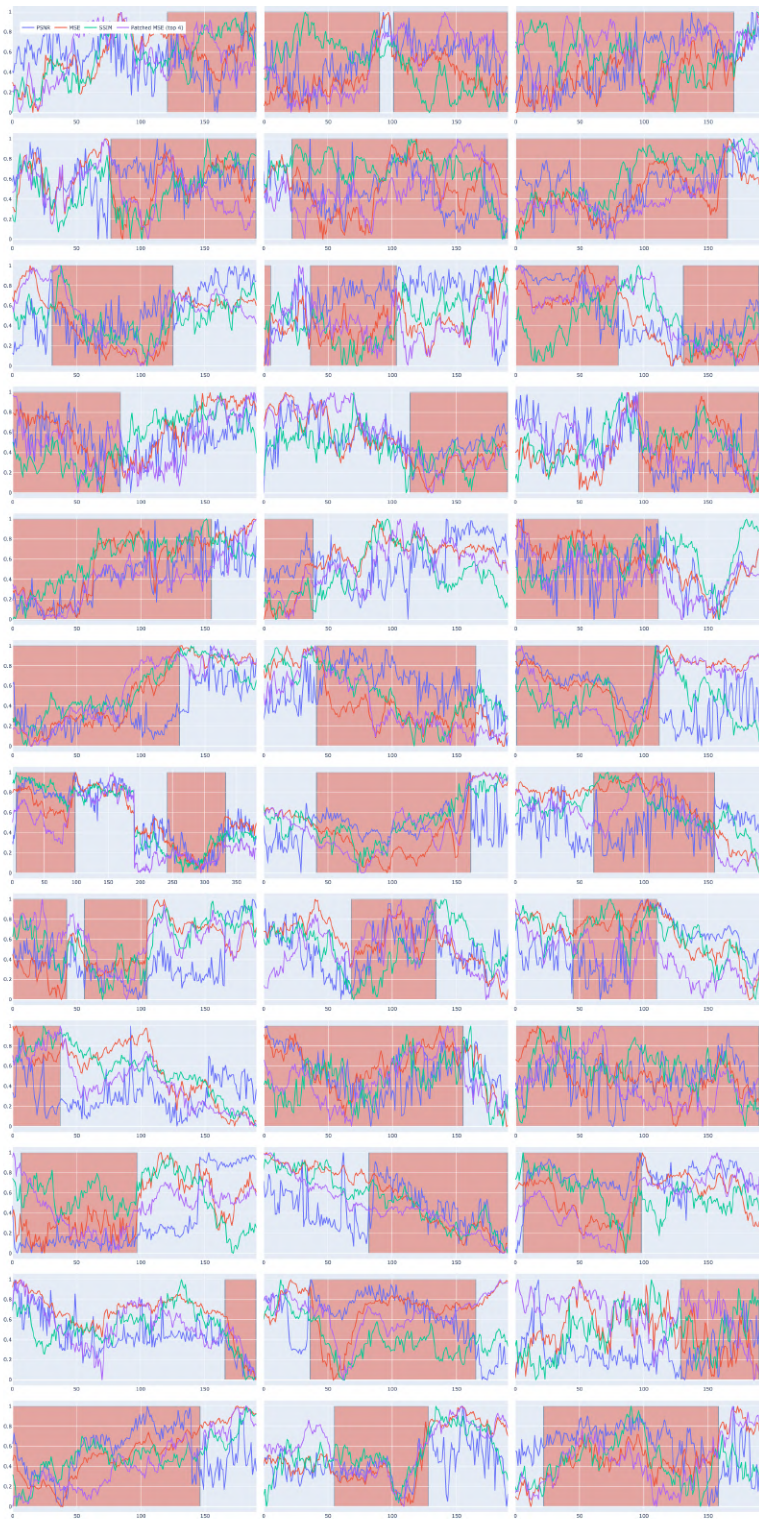


Figure A.3: Spatio Temporal Convolutional Autoencoder evaluation results on the *UCSD 1* test dataset.

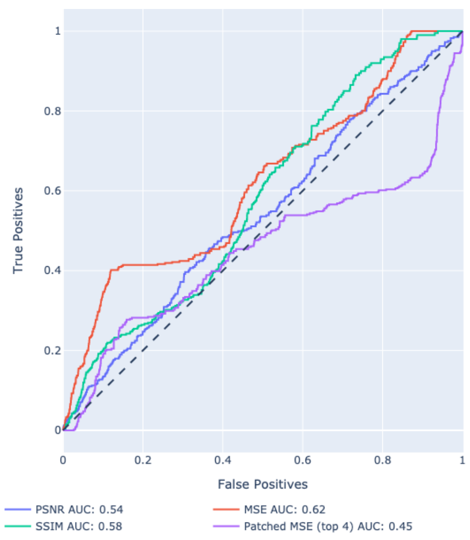
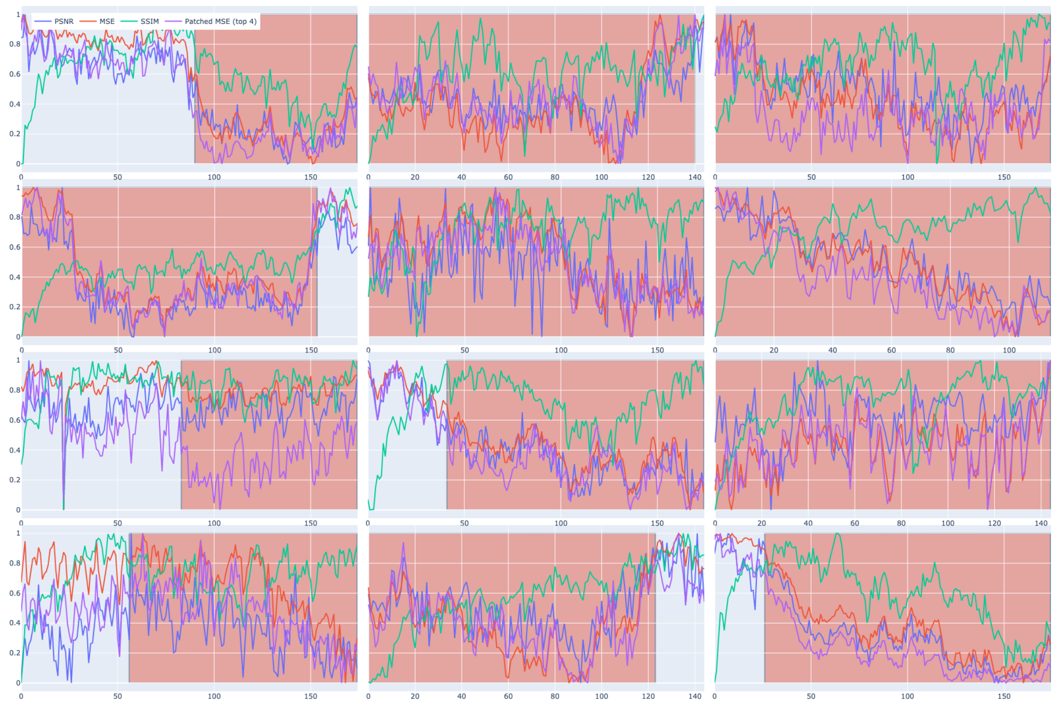
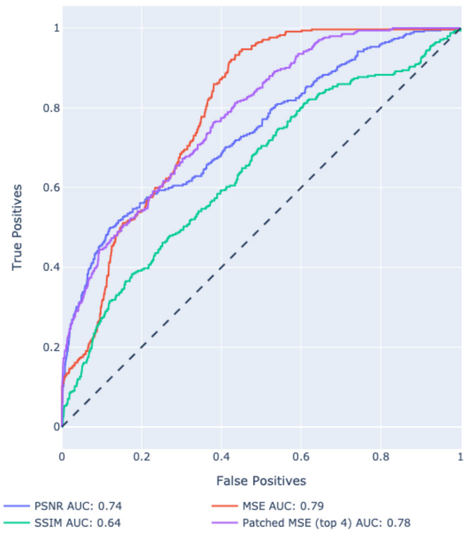


Figure A.4: Spatio Temporal Convolutional Autoencoder evaluation results on the *UCSD 2* test dataset.

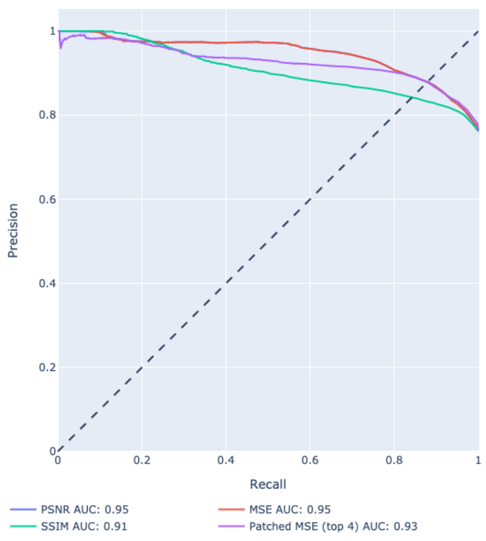
A.2 FastAno

A.2.1 UCSD

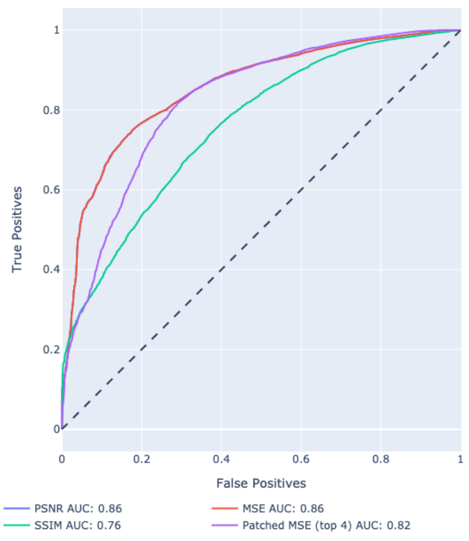




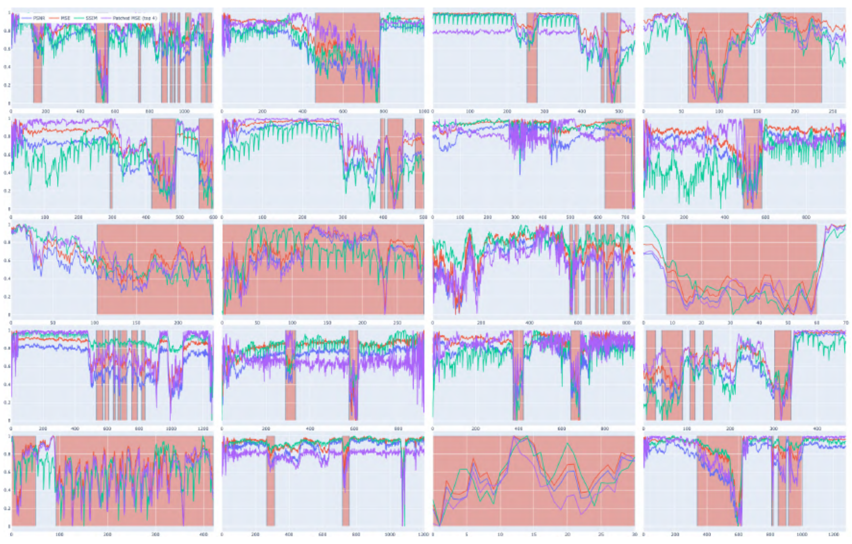
A.2.2 Avenue



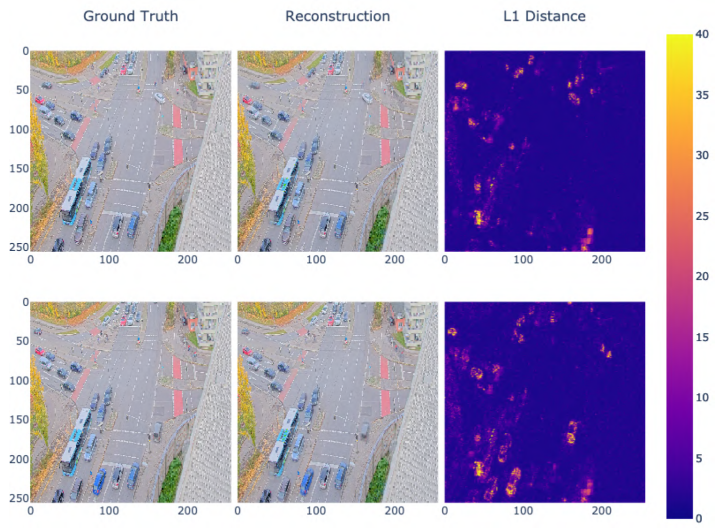
Precision
Recall
PSNR AUC: 0.95
MSE AUC: 0.95
SSIM AUC: 0.91
Patched MSE (top 4) AUC: 0.93



True Positives
False Positives
PSNR AUC: 0.86
MSE AUC: 0.86
SSIM AUC: 0.76
Patched MSE (top 4) AUC: 0.82



A.2.3 FKK



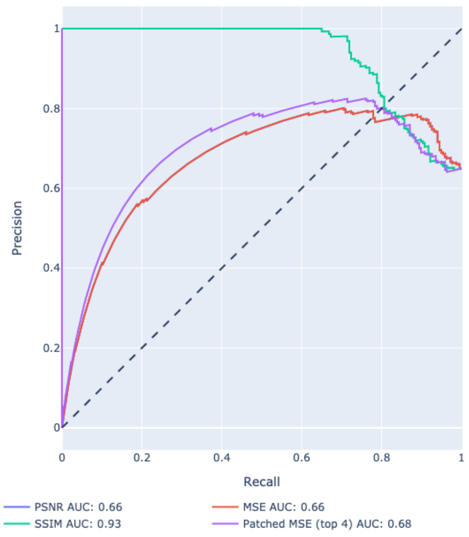


Figure A.5: Precision recall curve resulting from an evaluation on the *e:fs FKK* dataset.

A.3 Convolutional LSTM Autoencoder

A.3.1 UCSD Anomaly Scores

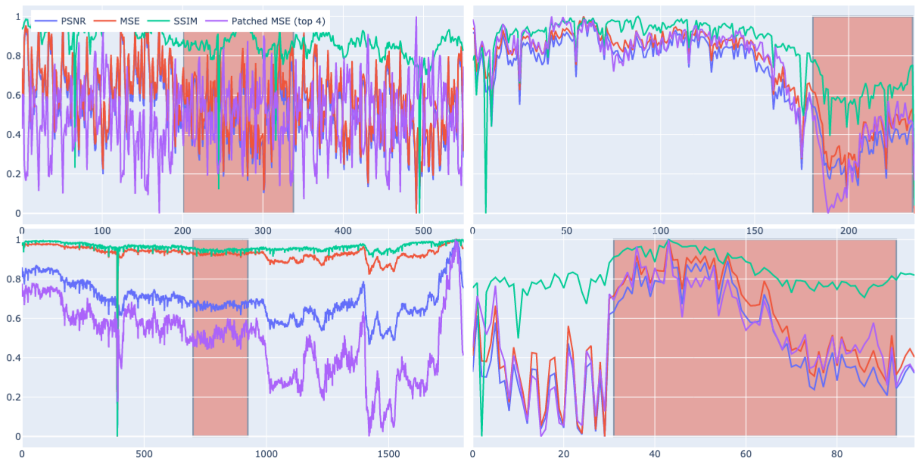


Figure A.6: Inferred regularity score graphs by the *FastAno* neural network architecture for the *e:fs FKK* dataset.

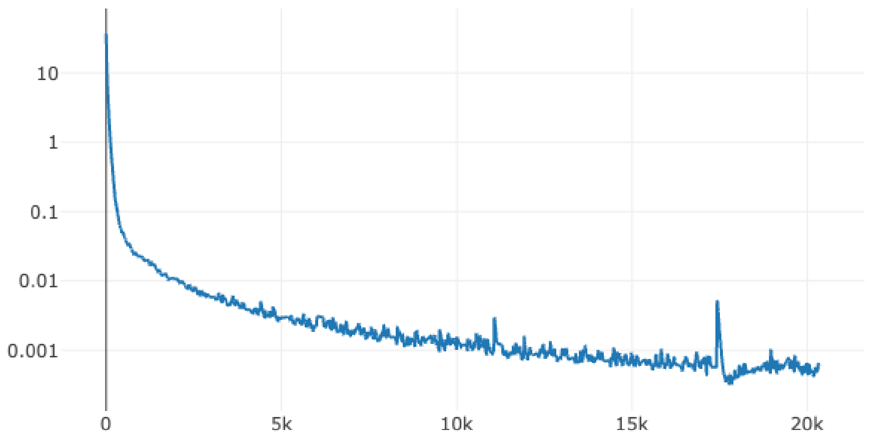


Figure A.7: *MSE* reconstruction loss on the *UCSD 1* dataset.

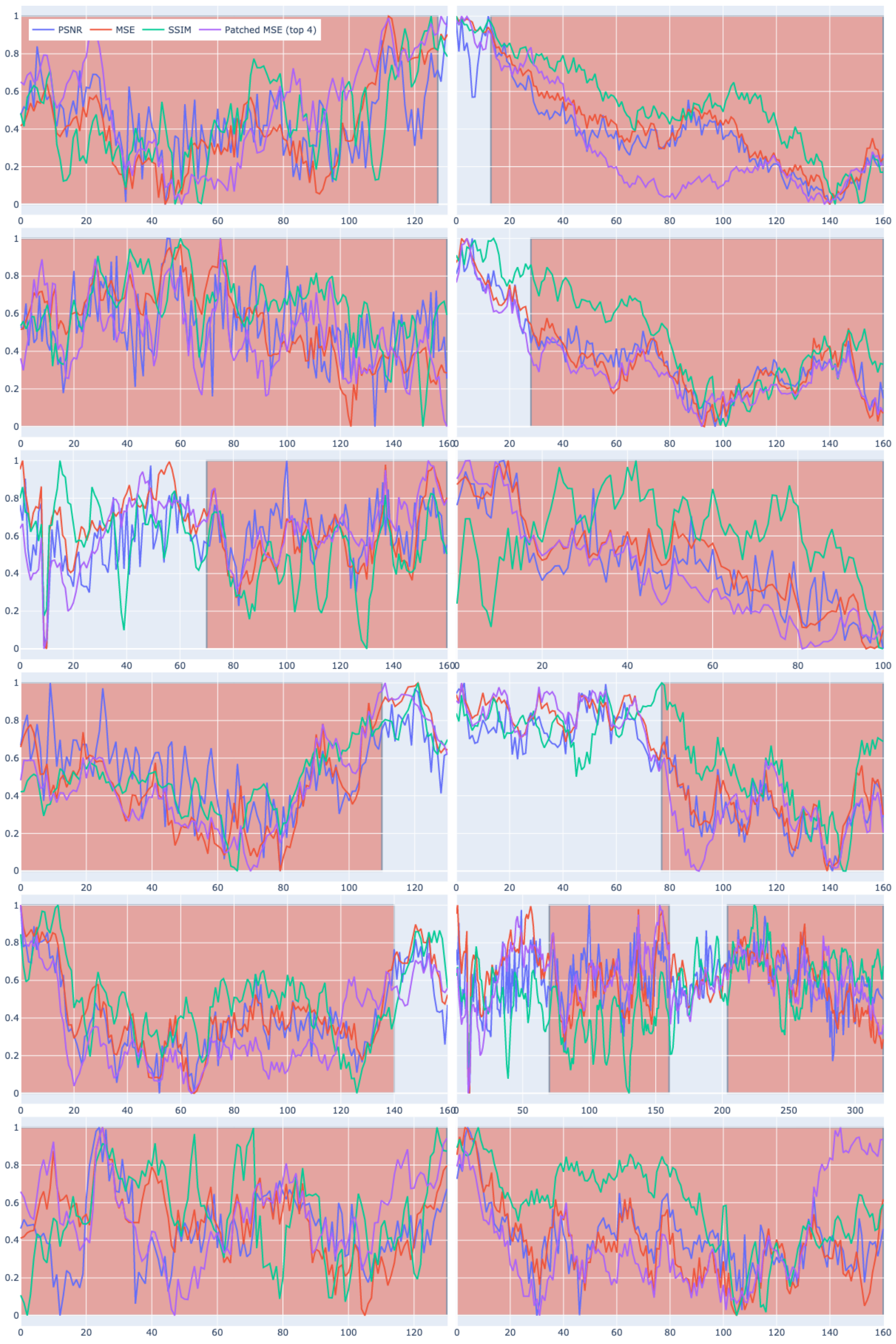


Figure A.8: Regularity scores as a function of time.

A.4 Convolutional LSTM Autoencoder with Lateral Connections

A.4.1 UCSD Pedestrian 1 & 2 Dataset

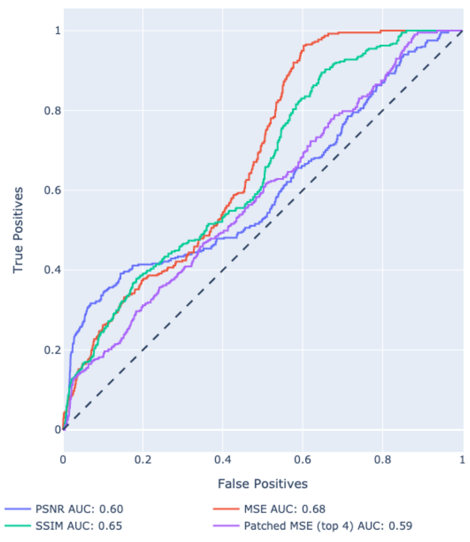


Figure A.9: Evaluation results on the *UCSD Pedestrian 2* dataset.

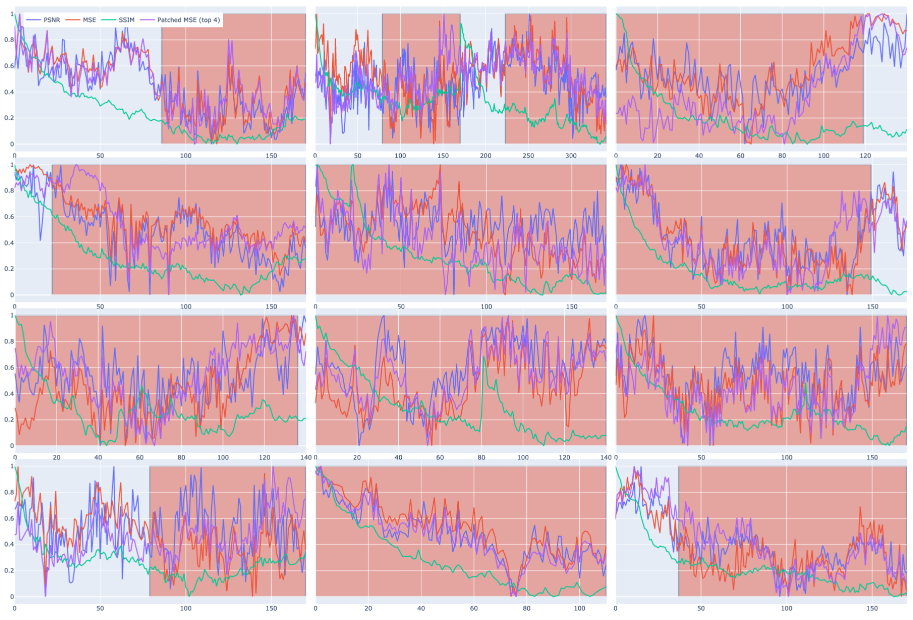


Figure A.10: Autoencoder architecture with convolutional *LSTM* lateral connections. Inference results of the *UCSD 1* test dataset.

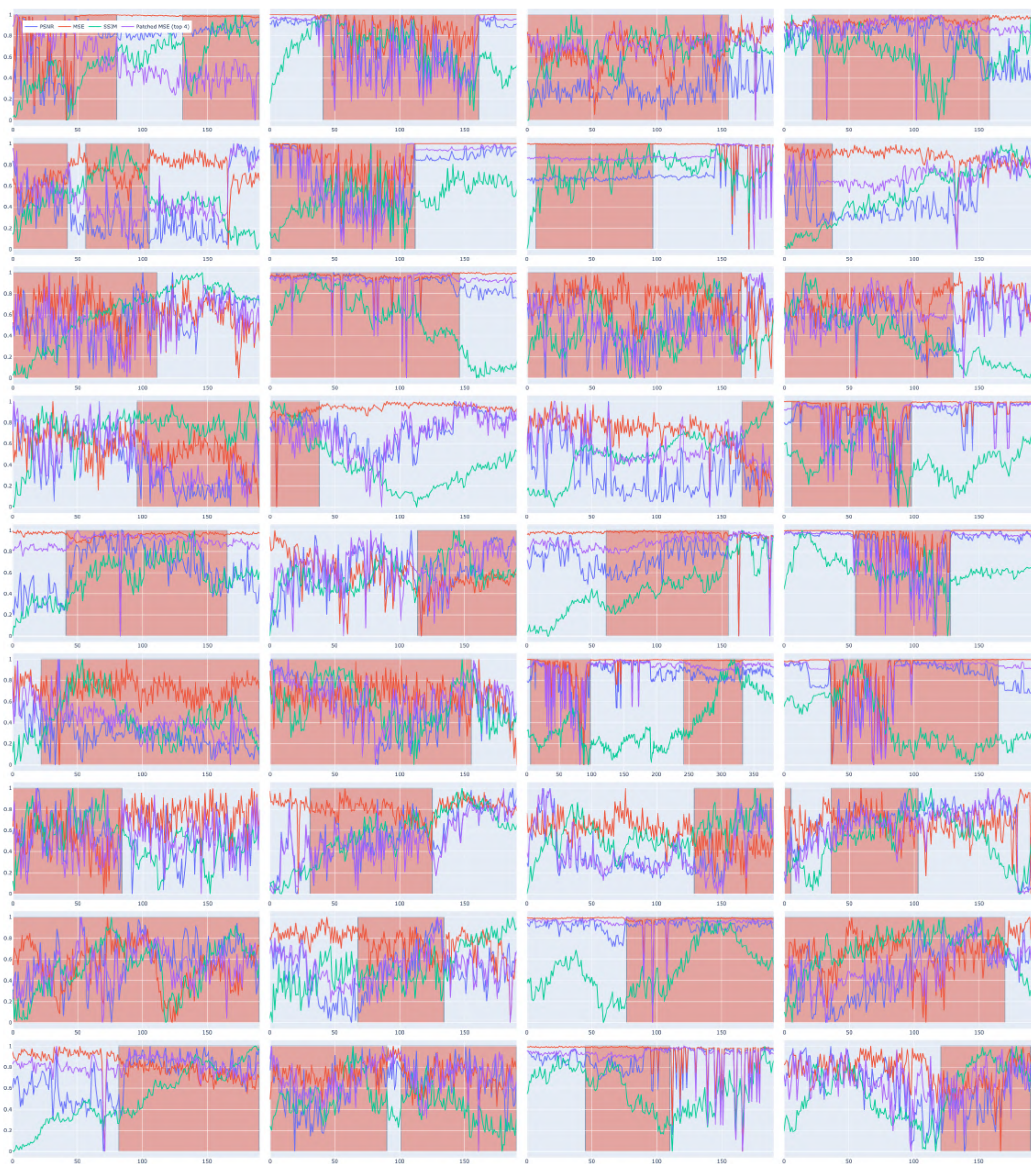


Figure A.11: Autoencoder architecture with convolutional *LSTM* lateral connections. Inference results of the *UCSD 2* test dataset.

A.4.2 Avenue

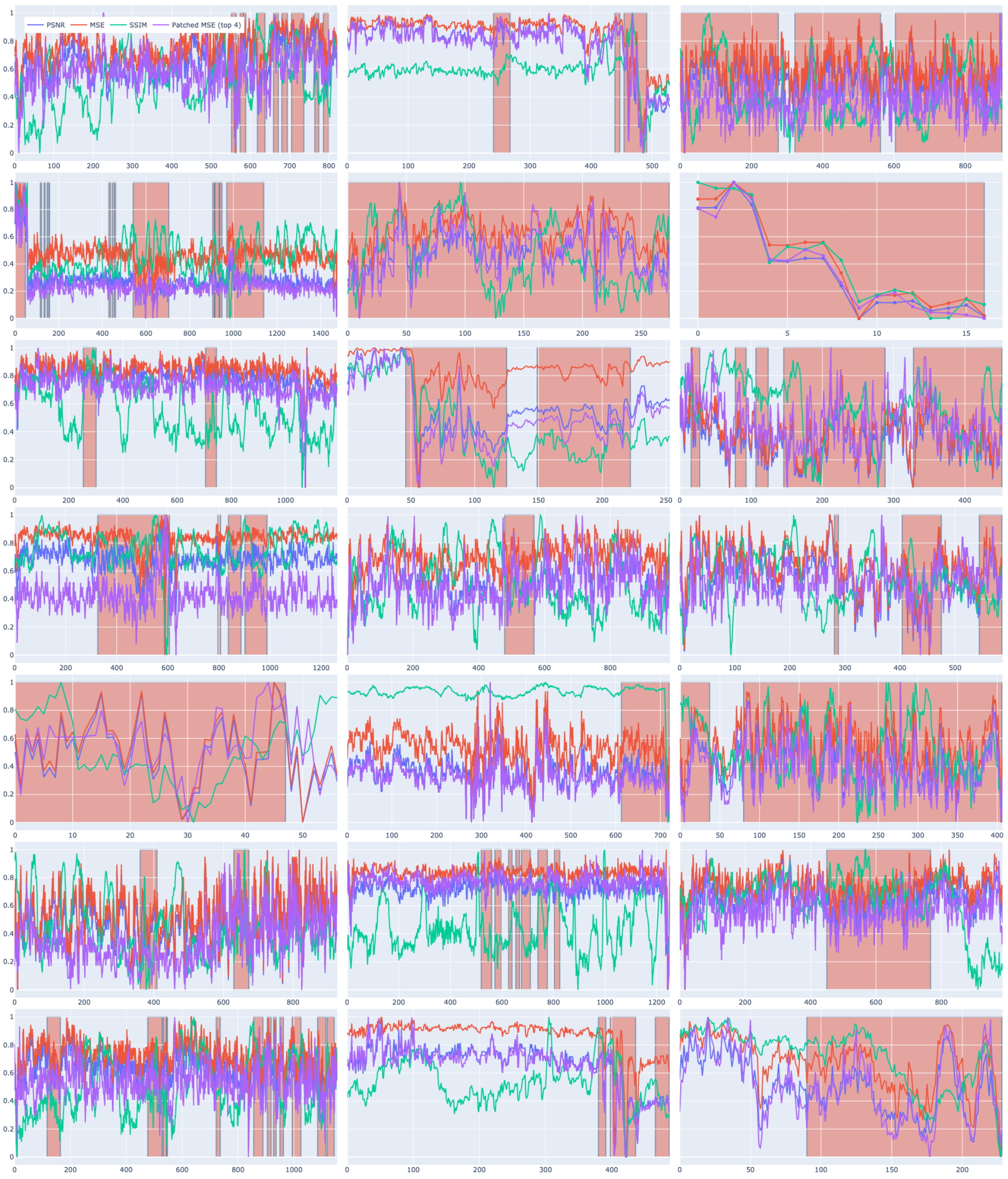
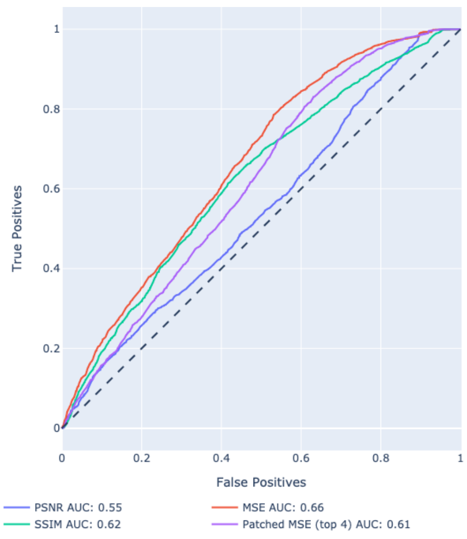
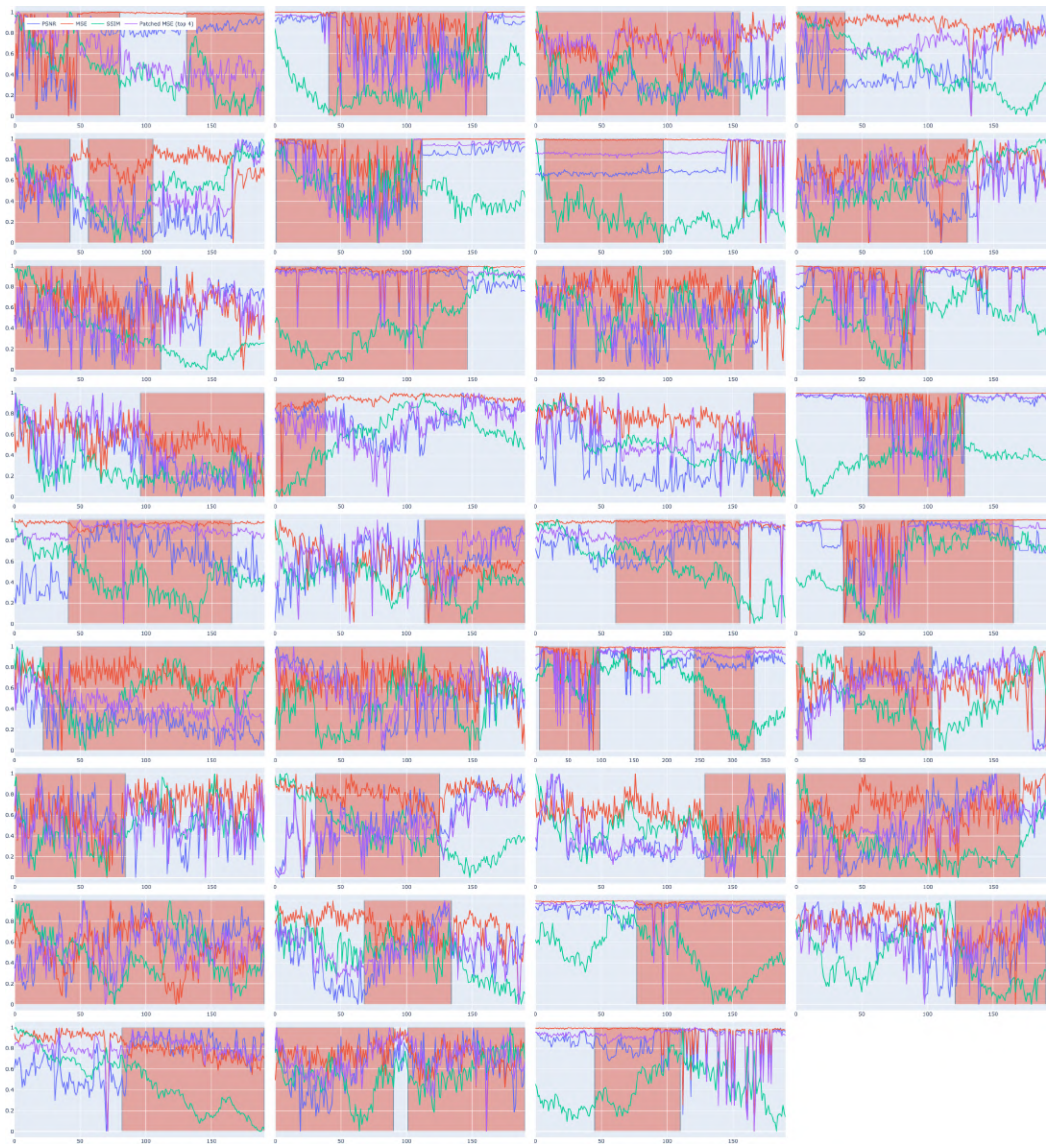


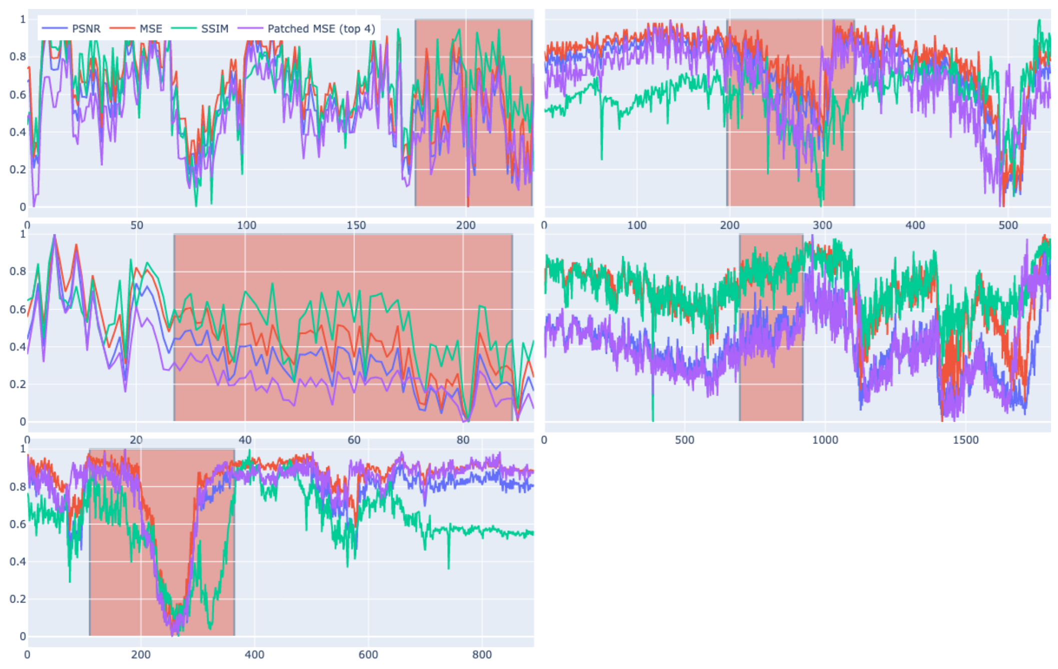
Figure A.12: Inference results on the Avenue test dataset.

A.4.3 Street Scene

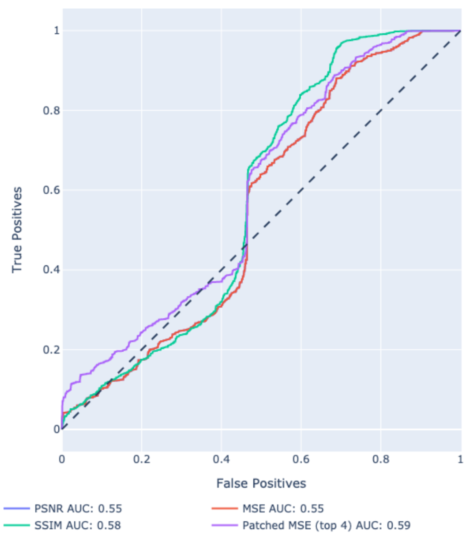




A.4.4 FKK



PSNR
MSE
SSIM
Patched MSE (top 4)



True Positives
False Positives
PSNR AUC: 0.55
MSE AUC: 0.55
SSIM AUC: 0.58
Patched MSE (top 4) AUC: 0.59

A.5 Transformer

A.5.1 UCSD

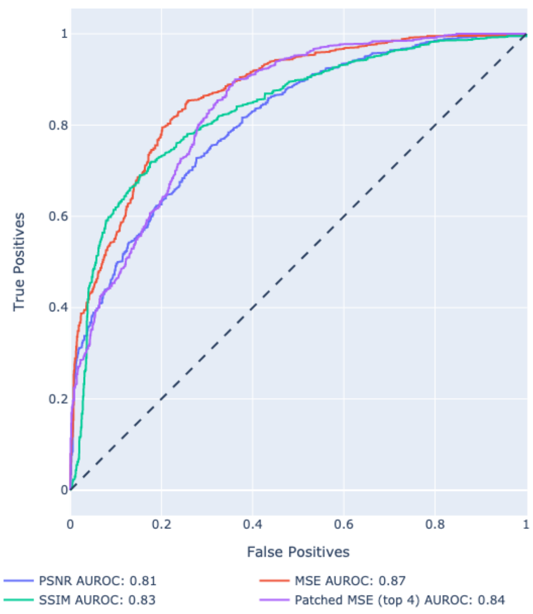


Figure A.13: Transformer candidate model evaluation results on the *UCSD 1* test dataset.

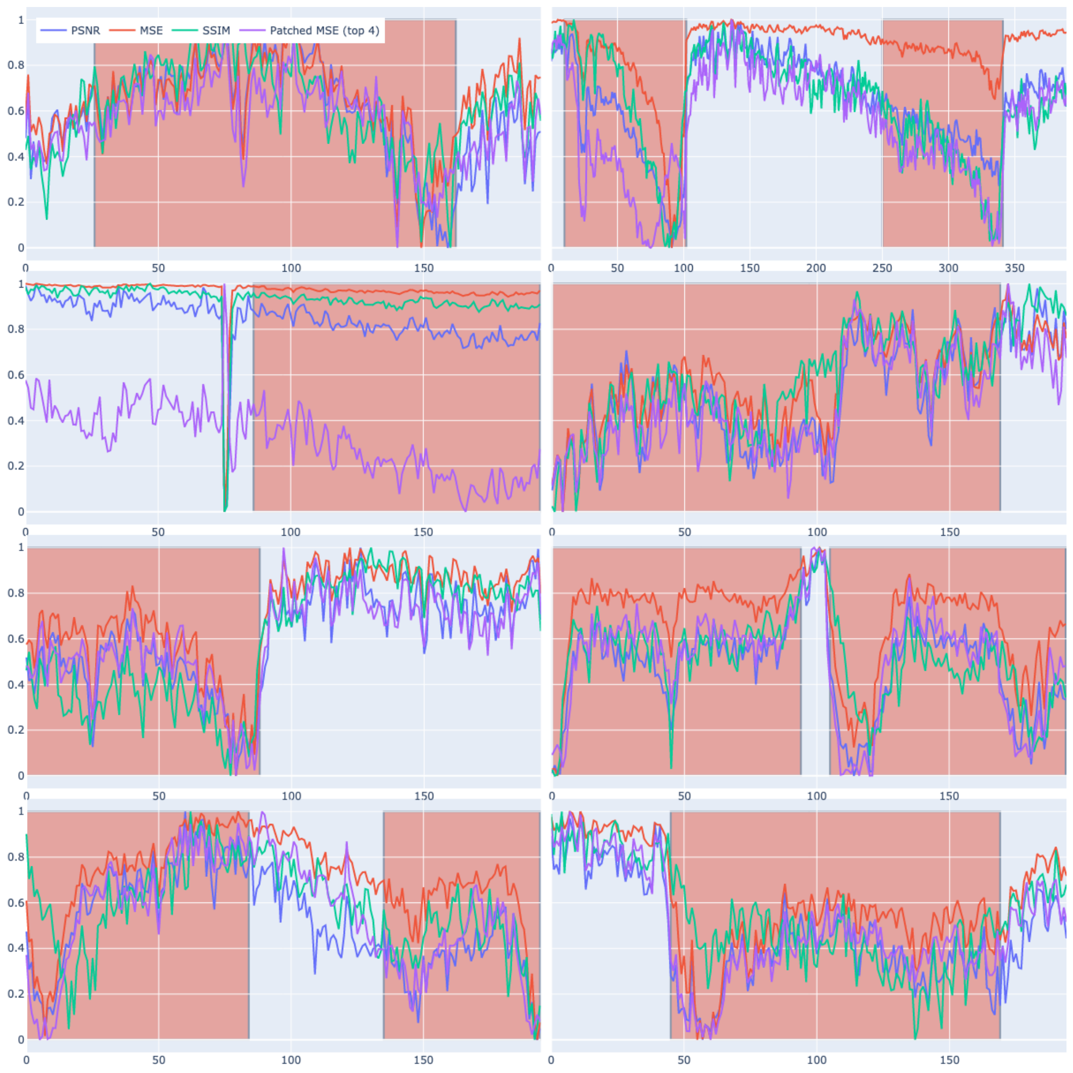


Figure A.14: Transformer test results on the *UCSD 1* test dataset.

A.5.2 Avenue

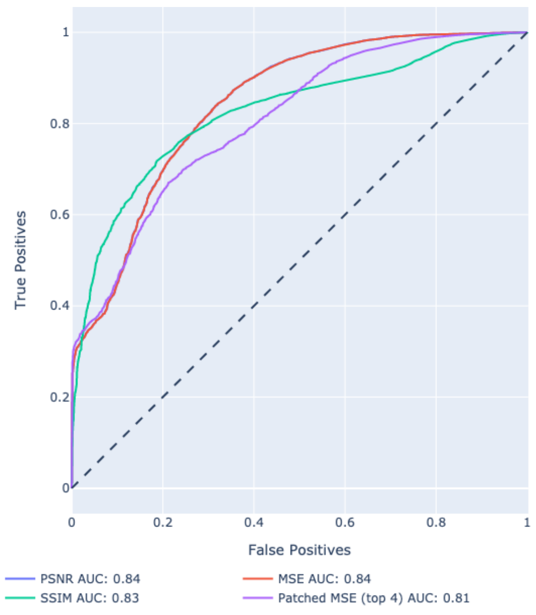


Figure A.15: ROC achieved by the transformer architecture on the *Avenue* test dataset.

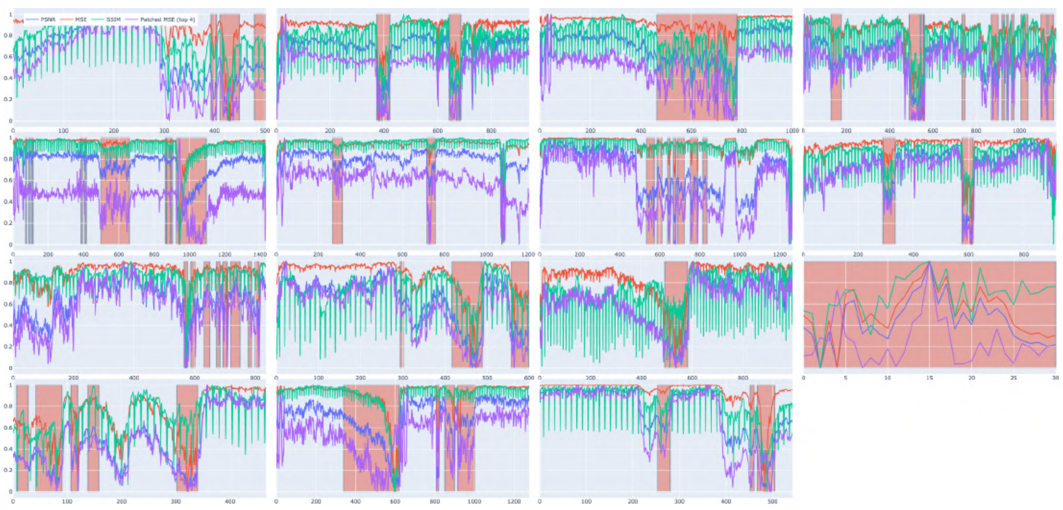
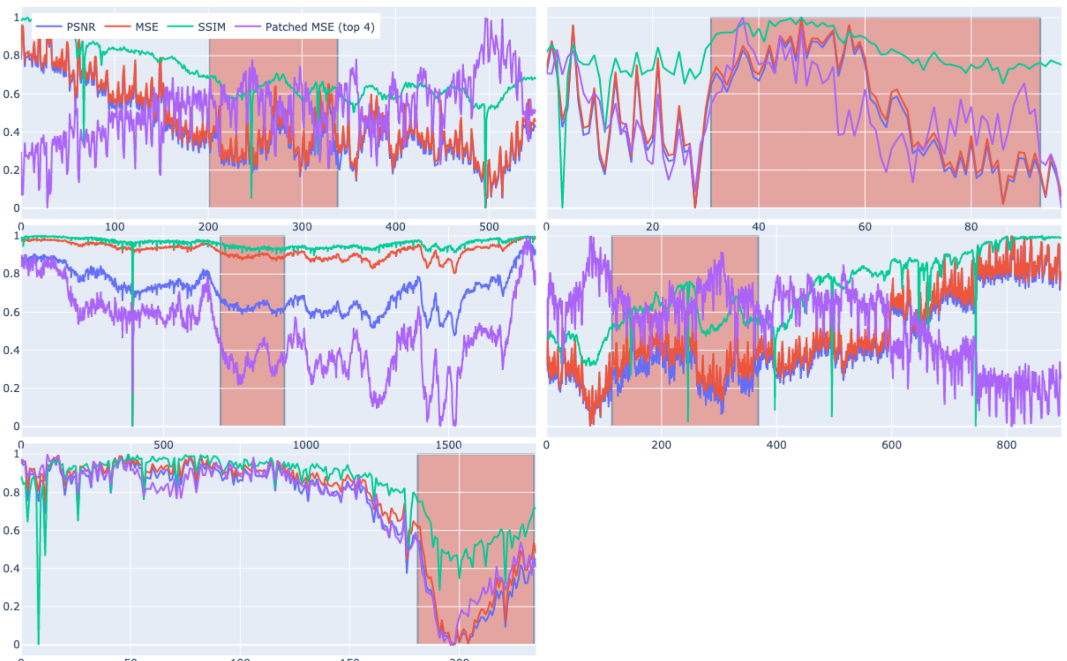


Figure A.16: Generated regularity score graphs from individual clips of the *Avenue* test dataset.

A.5.3 FKK Dataset



APPENDIX B

Appendix - Documentation

trafficanomalydetection.models.transformer module

Implementation inspired by ‘TransAnomaly: Video Anomaly Detection Using Video Vision Transformer’.

Source: https://ieeexplore.ieee.org/document/9525368

class **DecoderBlock**(*in_channels: int, out_channels: int, crop: int*) [source]

Bases: `Module`

A single U-Net like decoder block.

The architectecture is described in TranAnomaly.

Summary

1. First the main input X is propagated through a deconvolution layer (ConvTranspose2d).
2. The lateral connection is than stacked ontop the deconvolution output, which leads to feature maps of channel $2 \times C_{out}$.
3. Two convolutions each followed by a $ReLU$ non-linearity are used to reduce the stacked feature maps to the output space: $2 \times C_{out} \rightarrow C_{out}$.

forward(*X: Tensor, lateral: Optional[Tensor]*) [source]

Perform a deconvolution with an optional lateral connection.

Parameters:
- **X** (*torch.Tensor*) – A torch tensor with shape $[B, C_{in}, H, W]$
- **lateral** (*Optional[torch.Tensor]*) – Optionally specify a lateral input of shape $[B, C_{in}, H, W]$

Returns: A torch.Tensor of shape $[B, C_{out}, H, W]$

training: *bool*

class **TransformerBlock**(*input_size: tuple[int, int], token_dim: int = 512, temporal_layers: int = 1, spatial_layers: int = 3*) [source]

Bases: `Module`

The implementation of this module is based on ViViT.

Steps

1. First a patch generator is used to reduce each frame into patches of size patch-size and projected to a vector of shape token_dim via a linear projection layer.
2. An additionally randomly initialized timestep of shape $[1, N_P, D]$ is concatinated onto each batch element resulting in the following transformation: $[B, T, N_P, D] \rightarrow [B, T+1, N_P, D]$.
3. To retain temporal information a randomly initialized temporal embedding tensor is added (not concatinated!) to every timestep.

4. To let the transformer learn temporal relations for every patch, we collapse the batch and N_P into a single axis. We pass this tensor of shape $[B * N_P, T+1, D]$ into an AttentionModule featuring a MultiheadAttention module.
5. After the propagation through the temporal transformer module, we revert the dimension collapse described in step 4: $[B * N_P, T+1, D] \rightarrow [B, T+1, N_P, D]$
6. The tokens predicted by the temporal transformer for timestep $T+1$ can now be extracted :math:`pred = [B, T+1, dots] and we can focus on learning spatial relations between every patch.
7. The same AttentionModule is used to learn spatial relations from the predicted tensor of shape $[B, N_P, D]$ (no reshape is necessary because we got rid of the temporal dimension).

`add_prediction_token`(*X: Tensor*)→ Tensor [source]

Concatinates a prediction token for $T+1$ for every batch.

Parameters: **X** (*torch.Tensor*) – A batched tensor containing tokens of shape $[B, T, N, D]$

Returns: A tensor of shape $[B, T+1, N, D]$

`add_spatial_embeddings`(*X: Tensor*)→ Tensor [source]

Add the spatial position embeddings to the given tensor.

The spatial embeddings vector has the same length as the number of tokens (patches) for each timestep. The same spatial embedding value will be added for each value in a token.

`add_temporal_embeddings`(*X: Tensor*) [source]

Add temporal embeddings to tokens.

`forward`(*X: Tensor*) [source]

Propagates the input tensor X through a temporal and spatial transformer module.

For more information see module description.

`frame_count`: *int = 4*

`generate_patches`(*X: Tensor*) [source]

Split the input image sequence into windows of size P.

To set P use the class parameter $self.patch_size$.

Parameters: **X** (*torch.Tensor*) – Input tensor of shape $[B, C, W, H]$

`patch_size`: *int = 2*

`token_dim`: *int = 512*

class **`VideoTransformerAutoencoder`**(*config: dict*) [source]

Bases: `BaseModel`

Spatio-temporal convolutional transformer architecture based on TransAnomaly.

`timesteps`

how many timesteps T the model should use to predict frame $T + 1$

Type: int

`gradient_loss_weight`

loss function weight for the gradient loss

Type: float

`discriminator_loss_weight`

adversarial (discriminator) training loss weight

Type: float

`reconstruction_loss_weight`

reconstruction weight loss

Type: float

`difference_loss_weight`

difference loss weight

Type: float

`token_dim`

dimensionality of the latent space

Type: int

`temporal_transformer_layers`

how many temporal transformer layers the model should use

Type: int

`spatial_transformer_layers`

how many spatial transformer layers the model should use

Type: int

`configure_optimizers()` [source]

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns: Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note

The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note

Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer and learning rate scheduler as needed.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, **`training_step()`** will have an additional `optimizer_idx` parameter.
- If you use **`torch.optim.LBFGS`**, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the **`optimizer_step()`** hook.

`difference_loss_weight`: *float* = 0.1

discriminator_loss_weight: *float* = *1*

forward(*X: Tensor*) [source]

Process a series of images to predict the next frame.

Parameters: **X** (*torch.Tensor*) – input frames with shape $[B, T, C, H, W]$

Returns: the predicted frame at $T + 1$ with shape $[B, C, H, W]$

gradient_loss_weight: *float* = *1*

input_channels: *int* = *3*

io_normalizer: *Optional[list[trafficanomalydetection.data.preprocessing.scaling.Normalizer]]* = *None*

predict_step(*batch*, *batch_idx*) [source]

Predicts frame T+n+1 from a series of frame [T, T+n].

Parameters:
- **batch** (-) – The first item should contain the input frame series and the second element should be the groudn truth to the prediction.firs
- **batch_idx** (-) – current batch index

Dims:

- Input Tensor: [B, T (self.timesteps), C, W, H]
- GT Tensor: [B, C, W, H]

reconstruction_loss_weight: *float* = *1*

spatial_transformer_layers: *int* = *3*

temporal_transformer_layers: *int* = *3*

test_step(*batch*, *batch_idx*) [source]

Operates on a single batch of data from the test set. In this step you'd normally generate examples or calculate anything of interest such as accuracy.

```
# the pseudocode for these calls
test_outs = []
for test_batch in test_data:
    out = test_step(test_batch)
    test_outs.append(out)
test_epoch_end(test_outs)
```

Parameters:
- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_id** – The index of the dataloader that produced this batch. (only if multiple test dataloaders used).

–

Returns: Any of.

- Any object or value
- `None` - Testing will skip to the next batch

```
# if you have one test dataloader:
def test_step(self, batch, batch_idx):
    ...


# if you have multiple test dataloaders:
def test_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single test dataset
def test_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    test_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'test_loss': loss, 'test_acc': test_acc})
```

If you pass in multiple test dataloaders, `test_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple test dataloaders
def test_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note

If you don't need to test you don't need to implement this method.

Note

When the `test_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of the test epoch, the model goes back to training mode and gradients are enabled.

timesteps: *int*

token_dim: *int = 512*

training_step(*batch, batch_idx*) [source]

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters:

- **batch** (`Tensor` | (`Tensor`, …) | [`Tensor`, …]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (`int`) – Integer displaying index of this batch
- **optimizer_idx** (`int`) – When using multiple optimizers, this argument will also be present.
- **hiddens** (`Any`) – Passed in if :paramref:`~pytorch_lightning.core.lightning.LightningModule.truncated_bptt_ste > 0.

Returns:

Any of.

- `Tensor` - The loss tensor
- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`
- `None` - **Training will skip to the next batch. This is only for automatic optimizatic**

  This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note

The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

`validation_step`(*batch, batch_idx*) [source]

Predicts frame T+n+1 from a series of frame [T, T+n].

Parameters:
- **batch** (-) – The first item should contain the input frame series and the second element should be the groudn truth to the prediction.firs
- **batch_idx** (-) – current batch index

Dims:

- Input Tensor: [B, T (self.timesteps), C, W, H]
- GT Tensor: [B, C, W, H]

`anomaly_score_patched`(*prediction: ~torch.Tensor, ground_truth: ~torch.Tensor, patch_size: int = 64, p: int = 3, diff_fn: callable = <function mse_loss>*) [source]

Calculate the anomaly score of a single image by comparing the most different patches of two images.

Parameters:
- **prediction** (-) – Image of shape $[C, W, H]$
- **ground_truth** (-) – image of shape $[C, W, H]$
- **patch_size** (-) – the patch height and width
- **p** (-) – the top patches to take into account

Returns:
- the final PSNR value as a torch scalar
- a matrix with individual mse values for each patch $[T, T, 1]$

`difference_loss`(*x1: Tensor, y1: Tensor, x2: Tensor, y2: Tensor*) [source]

Calculate the difference loss as defined in TransAnomaly.

Parameters:
- **x1** (-) – predicted frame at $T + 1$
- **y1** (-) – ground_truth at $T + 1$
- **x2** (-) – predicted frame at $T + 2$
- **y2** (-) – ground_truth at $T + 2$

Returns: a single torch scalar as a Tensor

`gradient_loss`(*prediction: Tensor, ground_truth: Tensor*) [source]

Calculate the mean $L1$ difference between the gradients of two images or series of images.

Module contents

class `ModelType`*(value)* [source]

Bases: `Enum`

An enumeration.

`AUTOENCODER`= *'autoencoder'*

`FAST_ANO`= *'fastano'*

`FFP`= *'FFP'*

`LSTM_AUTOENCODER`= *'lstm_autoencoder'*

`TRANSFORMER`= *'transformer'*

`get_model`*(model: ModelType, config: TrainingConfig)* [source]

`get_preprocessor`*(model: ModelType, config: TrainingConfig)* [source]