Bachelorthesis

Comparing different state-of-the-art solutions for image prediction using time-series analysis

Sören Dittrich
soeren.dittrich@uni-hildesheim.de

September 2020

Abstract

The thesis will compare different state-of-the-art solutions for image-/video-prediction 2.1. The main module of those solutions, which is the core aspect of this work, is the LSTM (Long short-term memory) 2.5. This module was invented by Hochreiter and Schmidhuber [9] in 1997 and is used heavily in the field of image-& video-prediction since then, e.g. in Srivastava et al. [24]. During the time the module got many different add-ons and changes, which are described in different papers ([17], [14], [27], [26] and many more.). To have a valid comparison, I implement three different state-of-the-art solutions for image-/video-prediction ([23], [17] and [14]). All of them use a ConvLSTM solution as recurrent sub-module, which is changed during the experiments with another, more advanced solution named PredRNN [27]. The algorithms are re-implemented in PyTorch [16], as well as the „standard“ConvLSTM and PredRNN.

The thesis starts by introducing necessary topics, followed by an in-depth comparison of different state-of-the-art solutions in the related work part. Then the implementation of the different baselines is described, followed by the experimental results. Afterwards there is a comprehensive discussion about the different approaches to perform image-/video-prediction and the experimental results. Lastly there is a summary to conclude the thesis.

Contents

1 Scientific questions

1. What different types of image prediction architectures exist?

2. How important is the choice of the recurrent module for the runtime and performance of the algorithm?

2 Introduction

The task of this chapter is to give the reader the basic knowledge, which is necessary to follow the rest of the thesis. In general, the reader should have basic knowledge about machine learning and neural networks. As this thesis is in the field of machine learning, more explicit neural networks and image-/video-prediction, this chapter will start giving basic knowledge about image-/video-prediction and specific neural network architectures, which are used in the thesis.

The necessary neural network architectures are Autoencoder 2.2, CNN 2.3 RNN 2.4, LSTM 2.5 and ConvLSTM 2.6, they are described briefly for the reader. Afterwards I describe the backpropagation algorithm 2.7 and BPTT 2.8 at a very basic level. Those two algorithms are the most used algorithms in training neural networks and are used in this thesis.

2.1 Image-Prediction / Video-Prediction

Image-/Video-prediction is a field inside machine learning, where the key is to predict future images, given a sequence of images. The image sequence X is of length n, $(x_0, \ldots, x_{n-1})$.

One possible use-case is the **one-frame prediction**, where one predicts x_n, given the the sequence X. Another common use-case is **multi-frame prediction**, where the key is to predict $t > 1$ many frames into the future $(x_n, \ldots, x_{n+t-1})$. This is often performed using sequence-to-sequence learning [25]. The first frames will look much better then the last frames, as ground-truth is missing, The predicted frames are only approximated, which means they contain a certain level of error, so using them as input to perform **multi-frame prediction** will increase the level of error for the following frames.

2.2 Autoencoder

The autoencoder is a network architecture, which consists of two neural networks chained together. The first network is called Encoder. This Encoder gets the input x and outputs the code h. Often the output layer of the Encoder is named bottleneck-layer. The second network is called Decoder. It gets the code h as input and outputs $x\prime$. This architecture is used for reconstruction, where $x \approx x\prime$. To prevent the architecture to simply copy the input directly to the output (which would be an interpolation and not the goal of any machine learning algorithm.), there are different techniques to have the autoencoder to instead approximate the output.

$$E(x) = h \tag{1}$$

$$D(h) = x\prime \tag{2}$$

The simplest autoencoder architecture is the so named undercomplete autoencoder [7], in which the output of the bottleneck-layer h is smaller then the input x. Therefore the architecture needs to learn how to extract useful features from the input x, because it is not able to simply copy the input x to the output $x\prime$, because h is a reduced representation of x. For example a undercomplete convolutional autoencoder, which is just the autoencoder architecture using convolutional layer, gets an input image, which it should reconstruct. If the code is smaller then the input image, the network needs to abandon information from the input. The idea is, that during the training, the network learns what type of information is obsolete and what type of information is necessary and should be stored in the code, to get a useful approximated $x\prime$. There are many different ideas of using the autoencoder architecture, which are described more in-depth in e.g. Goodfellow et al. [7].

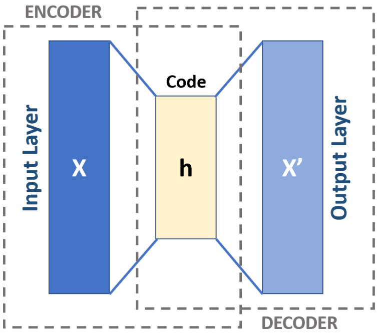


Figure 1: Autoencoder schema [15]

2.3 CNN

CNN (Convolutional neural network) is a type of network, which uses the mathematical operation convolution, instead of multiplication and addition. A CNN typically consists of three stages:

1. Convolutional operation
2. Non-linearity (ReLU, sigmoid, ...)
3. Pooling

The convolutional operation used is more precisely a discrete convolutional operation. Convolutional operations are applicable on two or three dimensional data. As this thesis will always stick to image data, the equation is given for the two dimensional convolution. In general the network is useful for e.g. time-series, as they can be represented in two or three dimensional tensors and images.

$$(I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n) \tag{3}$$

I is the two dimensional image.
K is the two dimensional kernel.
$*$ is the commonly used sign for the convolution operation.
The first stage looks like:

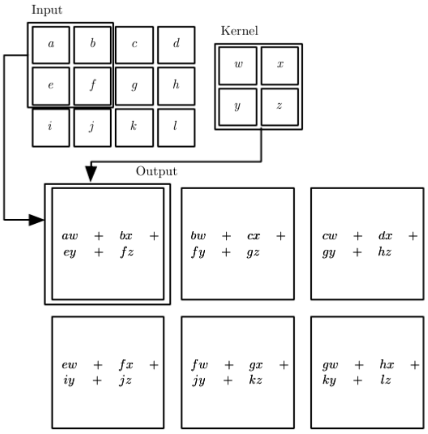


Figure 2: Two dimensional convolutional operation performed using a kernel [7]

In figure 2, one can see the idea of applying the kernel on the input data. In this example the stride is 0 and the padding is 0.
The second stage is applying a non-linear function on the output, e.g. ReLU or simgoid. Lastly there is the pooling layer, in which a pooling function is applied on the output of the second stage. This is done to make the output of the second stage invariant to the input. The pooling layer reduces the output size by at least two. For example, an input image of size $(32 \times 32 \times 1)$, $(Height \times Width \times Channel)$ is then outputted at a maximum size of $(16 \times 16 \times 1)$.

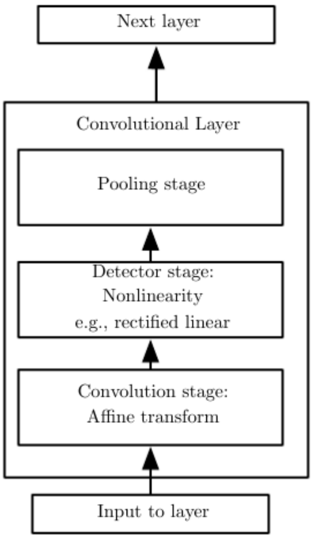


Figure 3: Stages of a CNN [7]

2.4 RNN

RNN (Recurrent neural network) is a network type, which is able to handle sequential data $X = (x_0, \ldots, x_{t-1})$, $|X| = t$. Therefore this type of network is used for e.g. time-series analysis and image-/video-prediction 2.1.

Despite a standard feed-forward neural network, a RNN will not only get a new input at a time-step, but also the output of the RNN of the last time-step. This requires a RNN to be initialized at start, because there is no output of the last time-step available. This last time-step input is often initialized as 0. A standard feed-forward neural network looks like:

$$\hat{y} = f_\theta(x) \tag{4}$$

Every approximated output $\hat{y}$ is only dependent of the input x and the computation inside the network. The RNN looks like:

$$\hat{y}^t = f_\theta(\hat{y}^{t-1}; x^t) = f_\theta(f_\theta(\hat{y}^{t-2}; x^{t-1}); x^t) = \dots \tag{5}$$

The approximated output $\hat{y}^t$ depends on the input x^t, but also on all previous outputs. In the literature the RNN is often schemed using a folded and an unfolded graph, to illustrate how the network architecture works.

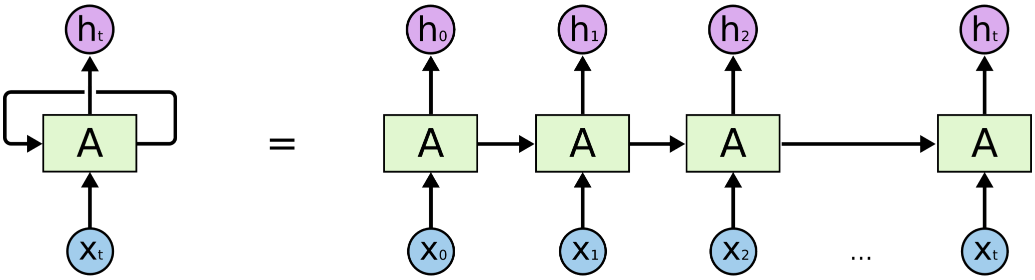


Figure 4: RNN schema. **Left:** Folded graph, **Right:** Unfolded graph [1]

The output of a RNN is often denoted as h for hidden-unit, because it is not only the output of the time-step, but also the input for the next time-step.
This networks are not learned with simple backpropagation 2.7, but often with BPTT (Backpropagation through time) 2.8.

2.5 LSTM

LSTM (Long Short-term Memory), invented by Hochreiter and Schmidhuber [9] is a form of RNN, which avoids a critical problem of standard RNN: Saving **Long-term dependencies** [7]. The architecture consists of different submodules: An inpute-gate, forget-gate, cell-state and output-gate.

$$i_t = \sigma(w_{x_i}x_t + w_{h_i}h_{t-1} + b_i) \tag{6}$$

$$f_t = \sigma(w_{x_f}x_t + w_{h_f}h_{t-1} + b_f) \tag{7}$$

$$c_t = f_t c_{t-1} + i_t tanh(w_{x_c}x_t + w_{h_c}h_{t-1} + b_c) \tag{8}$$

$$o_t = \sigma(w_{x_o}x_t + w_{h_o}h_{t-1} + b_o) \tag{9}$$

$$h_t = o_t tanh(c_t) \tag{10}$$

w is the weight of the layer
σ the sigmoid function
b the layer bias.
h_t is the output, in RNN's the output is often denoted as hidden 2.4.

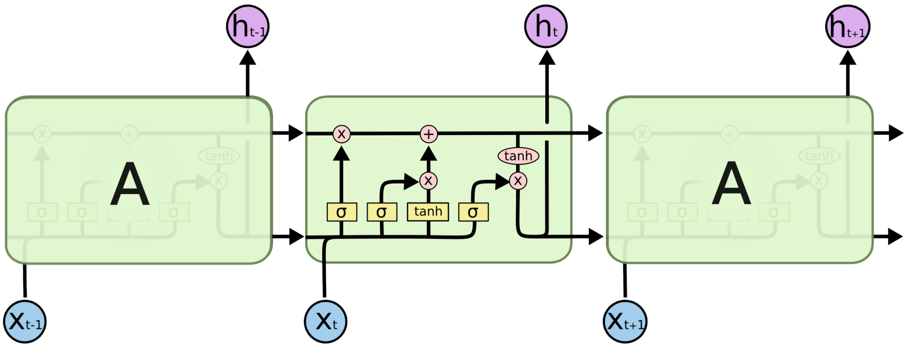


Figure 5: LSTM Architecture [1]

It is much better in saving long-term dependencies as a standard RNN, because of the cell-state. This cell-state can be seen as the main memory of the LSTM, where every iteration saves important information and deletes unnecessary information. The cell-state can be seen as the upper horizontal line in figure 6.

The given equations for the LSTM are for the very basic LSTM without peephole.
The peephole is an idea from Gers and Schmidhuber from the year 2000 [6], where they augmented the LSTM with peephole connections, which gave them an advantage in learning spike trains[1], because a LSTM without peephole was not able to learn those spike trains. The equations looks very similar, with only few changes:

$$i_t = \sigma(w_{x_i} x_t + w_{c_i} c_{t-1} + b_i) \tag{11}$$

$$f_t = \sigma(w_{x_f} x_t + w_{c_f} c_{t-1} + b_f) \tag{12}$$

$$c_t = f_t c_{t-1} + i_t tanh(w_{x_c} x_t + b_c) \tag{13}$$

$$o_t = \sigma(w_{x_o} x_t + w_{c_o} c_t + b_o) \tag{14}$$

$$h_t = o_t tanh(c_t) \tag{15}$$

2.6 ConvLSTM

The convolutional LSTM, invented by Shi et al. [23] is a LSTM with peephole using convolutional layer instead of fully connected ones. Therefore the formulas look very similar to the ones in section 2.5.

$$i_t = \sigma(x_t * w_{x_i} + h_{t-1} * w_{h_i} + w_{i_b}) \tag{16}$$

$$f_t = \sigma(x_t * w_{x_f} + h_{t-1} * w_{h_f} + w_{f_b}) \tag{17}$$

$$\tilde{c}_t = tanh(x_t * w_{x_{\tilde{c}}} + h_{t-1} * w_{h_{\tilde{c}}} + w_{c_{\tilde{b}}}) \tag{18}$$

$$c_t = \tilde{c}_t \odot i_t + c_{t-1} \odot f_t \tag{19}$$

[1] Spike trains are time series of 0s and 1s, where 1 is defined as a spike.

$$o_t = \sigma(x_t * w_{x_o} + h_{t-1} * w_{h_o} + w_{o_b}) \tag{20}$$

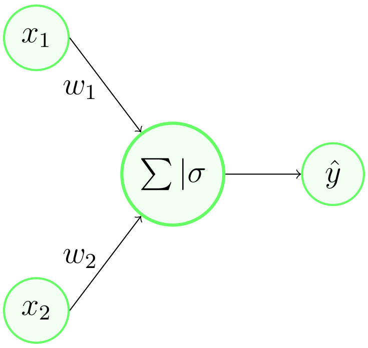
$$h_t = o_t \odot tanh(c_t) \tag{21}$$

$*$ is the commonly used sign for the convolution operation.
$\odot$ is the hadamard product (point-wise multiplication).

There is also a ConvLSTM without peephole, which is used in Patraucean et al. [17], which is implemented in section 4.

2.7 Backpropagation

Backpropagation, invented in 1986 by Rumelhart et al. [21], is the most used algorithm for training neural networks, due to it's simplicity. The algorithm should be already known to the reader, therefore I will only give a very simple example of it at the most simple neural network, the Perceptron [19].



The usage of backpropagation is very straight forward. Neural networks are representation learning algorithms, which means, that they learn the representation of the data over time, without having the need of an expert doing supervision. It only needs a valid training and testing dataset, where one has ground-truth knowledge of the output of the data. One then leverages the forward pass of the algorithm to produce our approximated output.

1. **Forward pass:**

$$\hat{y} = \sigma(\sum_{i=1}^{2} x_i w_i) \tag{22}$$

   After computing the regarding output, one will compare the computed output with the ground-truth output with some kind of error-function, e.g. **MSE**:

$$L(y, \hat{y}) = \frac{1}{N}||y - \hat{y}||_2^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{23}$$

   This error is then propagated back, using the chain-rule through the graph to update the weights of the network. This is done in the backward pass.

2. **Backward pass:**

$$\frac{\partial L}{\partial \hat{y}} = \frac{2}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i) \tag{24}$$

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x)) \tag{25}$$

$$\frac{\partial L}{\partial \sum_{i=1}^{2} x_i w_i} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \sum_{i=1}^{2} x_i w_i} = \frac{2}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i) \cdot \sigma(\sum_{i=1}^{2} x_i w_i)(1 - \sigma(\sum_{i=1}^{2} x_i w_i)) \tag{26}$$

$$\frac{\partial L}{\partial w_1} = \ldots = \frac{2}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i) \cdot \sigma(\sum_{i=1}^{2} x_i w_i)(1 - \sigma(\sum_{i=1}^{2} x_i w_i)) \cdot x_1 \tag{27}$$

3. **Update weights using Gradient Descent:**

$$w_1 = w_1^{old} - \lambda \cdot \frac{\partial L}{\partial w_1} \tag{28}$$

λ is the learning rate and is set as a hyperparameter.

Those three steps are performed iterative, as long as the error value is higher then an artificially set value ϵ.
It is also useful to use a second derivative method e.g. Newton method, instead of Gradient Descent, but in most cases Gradient Descent is used, due to it's simplicity and parallelization properties [21].

2.8 BPTT

BPTT (Backpropagation through time), invented by Paul Werbos [28], is a only slightly different algorithm, compared to simple backpropagation 2.7. This algorithm is explicitly designed for RNN's and as the name already denotes, able to backpropagate through the time. To make it even more clear, the algorithm does nothing more, then unfolding the graph during the backward pass and backpropagates through all unfolded time-steps performed by the recurrent module. Let's make a simple example to fully understand the idea. Therefore I will use a simple RNN architecture found in Goodfellow [7]. To have a basic comparison to the backpropagation in section 2.7, I changed the softmax function to σ, as this derivation is already known with equation 25.

1. **Forward pass:**

$$h_t = tanh(Wh_{t-1} + Ux_t + b_1) \tag{29}$$

$$o_t = Vh_t + b_2 \tag{30}$$

$$\hat{y}_t = \sigma(o_t) \tag{31}$$

U, V, W are the weight matrices for input-to-hidden, hidden-to-output and hidden-to-hidden.
After computing the output, one again needs an error function, e.g. MSE (known from

equation 23.)
This time, it is also necessary to have this error function for every iteration.

$$L(y, \hat{y}) = \sum_t (L_t(y_t, \hat{y}_t)) \tag{32}$$

After computing the MSE for all time-steps and the overall error, one again starts the backward pass.

2. **Backward pass:**

$$\frac{\partial L}{\partial V} = \sum_t \frac{\partial L_t}{\partial V} \tag{33}$$

$$\frac{\partial L_t}{\partial \hat{y}_t} = \frac{2}{N} \sum_{i=1}^{N} (y_t^i - \hat{y}_t^i) \tag{34}$$

$$\frac{\partial L_t}{\partial o_t} = \frac{\partial L_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial o_t} = \frac{2}{N} \sum_{i=1}^{N} (y_t^i - \hat{y}_t^i) \cdot \sigma(o_t)(1 - \sigma(o_t)) \tag{35}$$

$$\frac{\partial L_t}{\partial V} = \frac{\partial L_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial o_t} \cdot \frac{\partial o_t}{\partial V} = \frac{2}{N} \sum_{i=1}^{N} (y_t^i - \hat{y}_t^i) \cdot \sigma(o_t)(1 - \sigma(o_t)) \cdot h_t \tag{36}$$

$$\frac{\partial L}{\partial V} = \sum_t \frac{\partial L_t}{\partial V} \tag{37}$$

3. **Update weights using Gradient Descent:**

$$V = V^{old} - \lambda \cdot \frac{\partial L}{\partial V} \tag{38}$$

3 Related work

This section will describe a range of state-of-the-art architectures for image prediction. Image prediction is a very broad field, but almost all state-of-the-art solutions for image prediction share one common part, the recurrent module. LSTM's are the most used modules in image prediction, as they are able to store information over a long period of time, despite the standard RNN (recurrent neural network). All algorithms described here have a different way to perform image prediction, but all use a type of LSTM to store the time-series information. It is very common for this type of papers, that the authors start their experiments by using a synthetic dataset. In the most cases this is MovingMNIST [13] and then afterwards performing tests on natural images. For natural images, the authors often use action-recognition datasets, because the camera is fixed and only objects/people in the scenery are moving. Another approach is using natural image datasets, where the camera is also moving through the scenery, for example Kitti dataset [5]. This dataset consists of natural images from different car drives through the city and residential area of Karlsruhe in Germany. In this dataset not only objects in the scenery are moving, but also the camera has a self-motion, which can be very tricky for image prediction algorithms to „understand“.

3.1 LSTM Autoencoder

The paper „Unsupervised Learning of Video Representations using LSTMs“by Srivastava et al. [24] is using a LSTM 2.5 from Graves [8] in an autoencoder architecture 2.2 for image reconstruction and future image prediction. The architecture is often used as a baseline in newer and more advanced architectures, because it consists of the standard LSTM as recurrent module. Due to the fact, that the LSTM module is not able to handle multi-dimensional data as is, the images need to be reshaped at the input and also at the output again. The authors use MovingMNIST [13] as synthetic dataset, where every image is of size $(64 \times 64 \times 1)$. Therefore the image is vectorized into $(64 \cdot 64 \times 1) = (4096 \times 1)$. This MovingMNIST implementation consists of two digits inside every frame. The authors input 10 images and output the next 10 images. The model is end-to-end differentiable and trained using BPTT (backpropagation through time) [28]. For the synthetic dataset, the model is trained using cross-entropy loss with logits **??**, for natural image datasets using MSE (mean-squared error) [29].

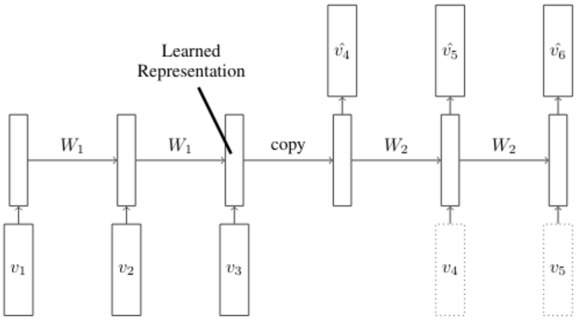


Figure 6: Architecture for future image prediction [24]

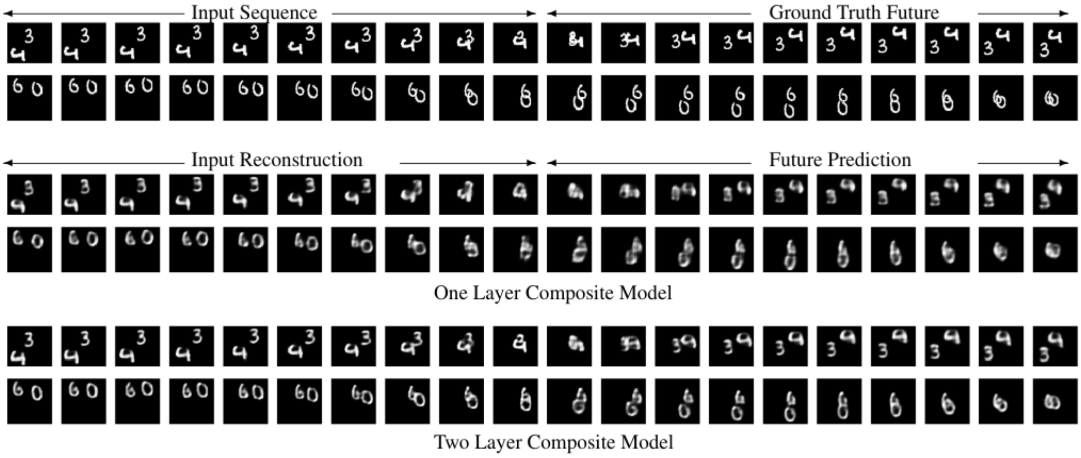


Figure 7: Results for MovingMNIST [24]

3.2 ConvLSTM Autoencoder

The paper „Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting“by Shi et al. [23] is using a similar architecture as Srivastava et al. in section 6, but instead of using the standard LSTM, they use a novel ConvLSTM 2.6. The ConvLSTM used in this architecture is with peephole **??**. This architecture outperforms the implementation of Srivastava et al., because it „captures spatiotemporal correlations better“. This model is, same as LSTM Autoencoder 3.1 end-to-end differentiable and trained using BPTT. It also uses the cross-entropy loss with logits for the synthetic dataset (MovingMNIST) experiment. In this MovingMNIST implementation, every frame consists of three digits. As in the architecture above, the authors here input 10 images and output 10.

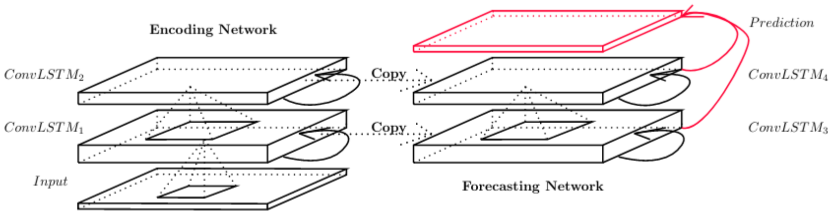


Figure 8: Future image prediction model [23]

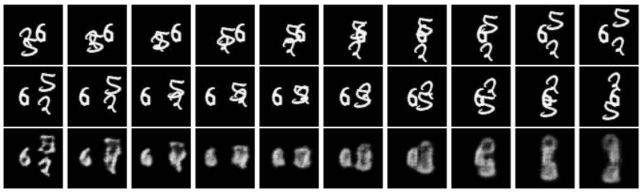


Figure 9: Results for MovingMNIST. **First Row:** 10 input images, **Second Row:** Ground truth next images, **Third Row:** Prediction of 3-layer implementation [23]

3.3 Spatio-temporal Video Autoencoder

The paper „Spatio-temporal Video Autoencoder With Differentiable Memory“by Patraucean et al. [17] describes a more complex architecture, in which the authors nest a temporal autoencoder inside a spatial autoencoder. The spatial autoencoder is a simple undercomplete autoencoder 2.2, where the decoder uses nearest-neighbor upsampling to get the output image size correct. The temporal autoencoder consists of a ConvLSTM (A ConvLSTM without peephole **??**, which works as the temporal encoder and and an optical flow convolutional module, which works as the temporal decoder. The network idea is to insert the image sequence X, which will create an optical flow map. This optical flow map is then applied on the last given image, to shift every pixel to it's new position. This will create the next image. The idea behind this is given in „Spatial Transformer Networks“by Jadeberg et al. [10]. The model is end-to-end differentiable and trained using BPTT. The authors also used MovingMNIST as synthetic dataset and use the cross-entropy loss with logits as reconstruction error for it. The MovingMNIST implementation is the same as in LSTM Autoencoder 3.1 (With two digits per frame.). In contrast to the other algorithms, this architecture is only capable of doing one-frame prediction as is. The authors input 19 images and output 1 image.

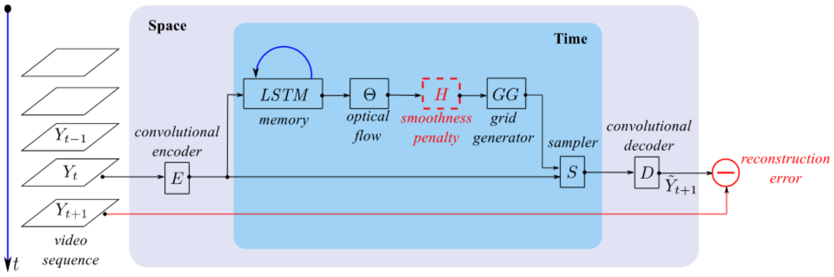


Figure 10: Spatio-temporal Video Autoencoder [17]

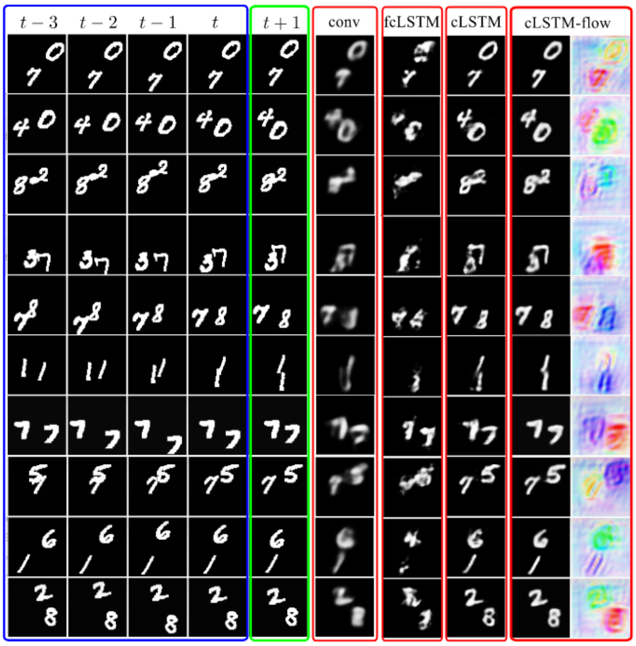


Figure 11: Results for MovingMNIST. **conv:** Is a simple convolutional autoencoder, without recurrent module, **fcLSTM:** LSTM Autoencoder 3.1, **cLSTM:** ConvLSTM Autoencoder 3.2, **cLSTM-flow:** Spatio-temporal Video Autoencoder with extra output of flow-map. [17]

3.4 PredNet

The paper „Deep Predictive Coding Networks For Video Prediction And Unsupervised Learning“by Lotter et al. [14] composes an architecture, which is informally named **PredNet**. It describes a network architecture based on the concept of „predictive coding“[18], [4]. It is the baseline for the experiments performed in section 5. The network consists of an arbitrary amount of layers, the amount of layers (depth of the network) can be treated as a hyperparameter. Every layer consists of an input module ${A_l}^t$, prediction module $\hat{A_l}^t$, representation module ${R_l}^t$ and error module ${E_l}^t$. l is the corresponding layer, t the timestamp.

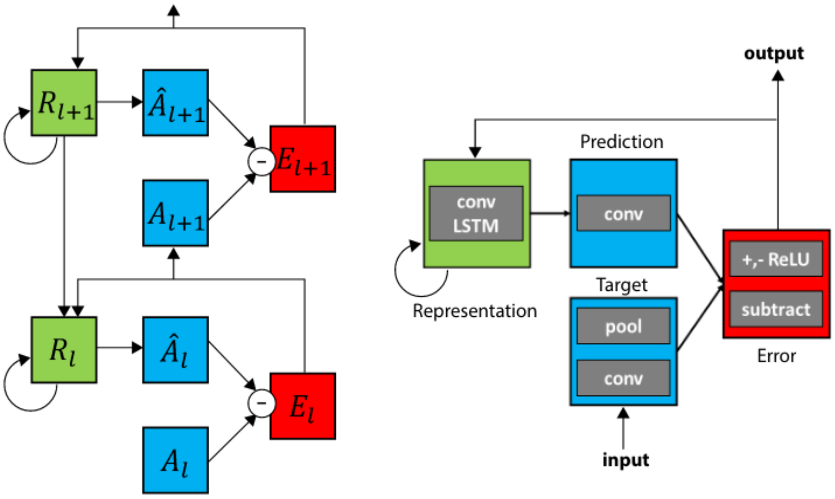


Figure 12: **Left:** PredNet architecture with two layers, **Right:** „Module operations for case of video sequences“[14]

$$ {A_l}^t = \begin{cases} x_t & l = 0 \\ MaxPool(ReLU(Conv(E^t_{l-1}))) & l > 0 \end{cases} \tag{39} $$

$$ \hat{A}^t_l = ReLU(Conv(R^t_l) \tag{40} $$

$$ E^t_l = [ReLU(\hat{A}^t_l - A^t_l); ReLU(A^t_l - \hat{A}^t_l)] \tag{41} $$

$$ R^t_l = ConvLSTM(E^{t-1}_l, R^{t-1}_l, Upsample(R^t_{l+1}) \tag{42} $$

3.5 PredRNN

The paper „PredRNN: Recurrent Neural Networks for Predictive Learning using Spatiotemporal LSTMs“by Wang et al. [27].

4 Implementation

This section describes the structure and usage of the implemented baselines (Shi et al. [23], Patraucean et al. [17], Lotter et al. [14]), so that the reader is able to understand the given code and is able to redo the experiments and even extend the code to his own needs.

4.1 Structure

The code is structured in a way, that everyone without specific knowledge of coding should be able to get the necessary files and results as easy as possible and everyone with more experience in coding should have a nice structure to add or remove certain methods easily.

First important files are the setup.py and requirements.txt. Both of them are useful to install every necessary requirement to run the code without any further problems on someones local machine or on any machine learning cloud computing platform. This was tested by me during the experiments on my private computer, on Floydhub and Google Colab. For Google Colab one needs to add a Jupyter notebook file [12], which is not included here.

The part where all comes together is the main file in the PredNet folder, in which the main method is run. This main method controls everything, such as initializing the network, the optimizer (Adam [11] or RMSProp [20]) and starting the training or testing. Testing, training and validation methods are seperated in own files, so the user has more control over the methods itself (For example, the user wants to validate on more then one error, but wants to test on only one, he can simply add the change to the validation file, without interchanging with any other method.). Everything related to the actual network models is stored in the model folder, which stores the three baselines and the folder for all submodules of which they consist (For example, PredNet has the error module, which is stored in model/modules/error.py.). Many useful methods, e.g. the choose of a certain error function can be found in the helper folder. The dataset files are stored in data, as well as some simple scripts to fetch and pre-process the data. There is also the dataset folder, which holds the PyTorch dataloader files. The code is able to save and load models, so one is able to continue training at a certain point, or test the network after training, those network files are stored in the mdl folder. To overcome the problem of having the need to change network parameters always inside the code files, I implemented a solution, which offers the user to simply add yml-files which contain every necessary information about the network parameter, where to store the model, and the logs, using debug mode and where the dataset files are stored. Those files are stored in the yml folder. To log the training, validation and test results, I am using Tensorboard [2]. Those files are stored in the log folder. Lastly one is able to create the backpropagation graph from the network one is using, this is outputted into the graph folder. A tree graph of the structure is given in the appendix 8.

4.2 Usage

The usage is also structured in a way, that everyone should be able to change mostly every parameter without having the need of changing lines of code. The code is started using the console with plenty of different mandatory and optional parameters. I will demonstrate this, giving an example for training and testing the PredNet model, using ConvLSTM. The first important thing to do, is to check if one already has a valid yml-file in the yml folder. As described in section 4.1, the user adds the network parameter, by adding a yml-file, which is then loaded by giving the corresponding file-path as console parameter. The training could look like:

```
python3.8 main.py -M 'prednet' -p 'yml/prednet_mnist_l0.yml' -d 'mnist'
          -e 1 -b 1 -s 10 -o 'adam' -L 'mse' -v 1 -m 'error' -r 0.001 -S
          -i 5000 -E 0 -n
```

The testing could then look like:

```
python3.8 main.py -M 'prednet' -p 'yml/prednet_mnist_l0.yml' -d 'mnist'
     -e 1 -b 1 -s 10 -o 'adam' -L 'mse' -v 1 -m 'prediction' -r 0.001 -l
     -i 5000 -E 0 -n -t
```

The describtion for the console parameter is given in the appendix 8.

5 Experiments

This section describes the experiments performed on the three networks and the results those experiments gave. The experiments were performed, to see if the implemented networks are able to perform better, when changing the recurrent sub-module with a more advanced recurrent module. The sections starts with the description of the experimental setup, followed by the experimental results.

5.1 Experimental Setup

The experiments were performed on three different datasets. First of all, I started using a synthetic dataset. This has the advantage of having full knowledge about the underlying structure of the data and that one has, in theory, an infinite amount of data for training and testing. This is also a very common approach and performed in all papers, compared in section 3. For the synthetic dataset I used MovingMNIST, which is a dataset created using handwritten digits from the famous handwritten dataset MNIST [13]. The idea is to use a pre-defined number of digits, which are spawned at a random position inside a given frame. Those digits are then moved through the frame, given a velocity and momentum. If a digit will touch the boundary of the frame, the digit will bounce back.
The other two used datasets are real datasets. The first one is KTH [22], an action recognition dataset, which consists of videos where people do different actions, such as hand waving, running and jogging. The second one is Kitti [5], which is an autonomous driving dataset, where the authors drove a car through different areas in Karlsruhe, Germany. For example through residential areas and the city.

I pre-processed MovingMNIST, to have a frame size of $(1 \times 64 \times 64)$, two digits per frame and ten frames per sequence. The training set consists of 10000 sequences and the test set of 3000 sequences.
KTH was pre-processed to gray-scaled images and cropped to the frame size $(1 \times 80 \times 60)$.
Kitti was also cropped to a specific frame size $(3 \times 160 \times 128)$.

To have a valid comparison, I decided to fix the amount of epochs and iterations per dataset, as well as all frames are normalized and having MSE as the error function for training and testing.
For MovingMNIST, I used one epoch with 5000 iterations, for KTH 20 epochs with 500 iterations per epoch and for Kitti 50 epochs with 500 iterations per epoch. The values for MovingMNIST are inspired by Elsayed et al. [3], where they develop a novel ConvLSTM module for PredNet and train PredNet on the MovingMNIST dataset (Which is not done by the PredNet authors.). The values for KTH and Kitti are inspired by Lotter et al. [14]. In the PredNet paper the authors use 150 epochs with 500 iterations per epoch to train PredNet on Kitti dataset, but I reduced the amount of epochs to 50 because the training would otherwise take more then 20 hours, which was not applicable for me. Because KTH is a more „simple“dataset (Only objects in the scenery are moving, not the scenery as in Kitti.), I artificially chose 20 as the amount of epochs. In general those are very important hyperparameters and one would need to implement concepts like early-stopping to get the correct amount of epochs to train the net on.

I used Adam [11] as optimizer for all networks, always with the same learning rate, of 0.001, and the same learning rate scheduler (Dividing the learning rate by factor ten after 50% of epochs.), even tho Patraucean et al. [17] are using RMSProp [20], in their paper, as optimizer and a different learning rate scheduler. The values are used by Lotter et al. [14] for PredNet, so I chose them as the hyperparameter for this experiment. But the idea behind those fixed values is simply, that all networks get the same amount of data with the same optimizer, the same learning rate scheduler without any unfair advantage, because the experiment is the comparison of the network using two different recurrent submodules. So both implementations have the same starting point, even tho the hyperparameter might not be the best for the network to reach the best performance.

In general there would be so many different additions to add to the experiments, such as the already named early stopping, random restarts, grid- or random- search, the usage of dropout and trying different optimizer and learning rate scheduler, but all those additions take plenty of computing time, which was not applicable for me and the scope of this thesis.

5.2 Experimental Results

This section is divided into the three different datasets, but all those subsections is structured the same way. It starts by comparing the amount of paramters for the dataset, because the amount of parameters is directly correlated to the computing time, the network needs for training and testing. It then shows some training graphs, to proove that the networks really learned something, followed by examples of the testing output at the end and a table of mean MSE error.

5.2.1 MovingMNIST

Model	**ConvLSTM**	**PredRNN**
Autoencoder (Depth 2)	537.411	773.018
PredNet	6.909.818	12.421.090
Spatiotemp	1.001.324	1.415.639

Table 1: Number of trainable parameter for MovingMNIST.

Model	**ConvLSTM**	**PredRNN**
Autoencoder (Depth 2)	0.027	0.028
PredNet	0.035	0.041
Spatiotemp	0.024	0.022

Table 2: Mean MSE for MovingMNIST.

5.2.2 KTH

Model	ConvLSTM	PredRNN
Autoencoder (Depth 2)	542.531	781.760
PredNet	850.325	1.285.730
Spatiotemp	1.007.598	1.421.913

Table 3: Number of trainable parameter for KTH.

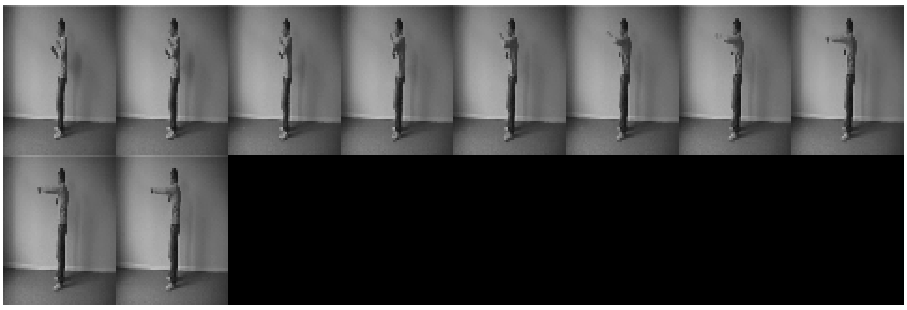


Figure 13: Ground truth frames

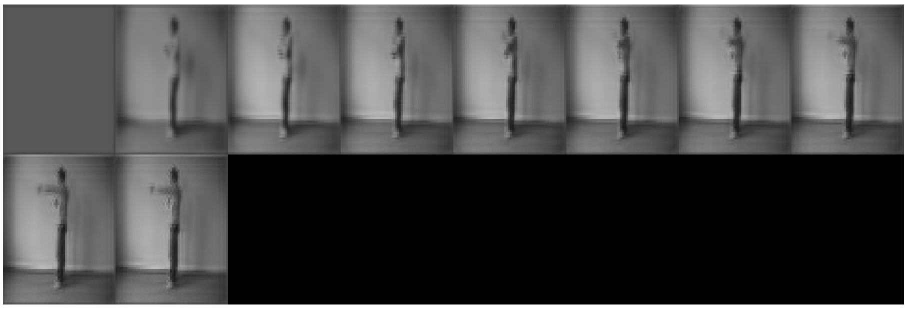


Figure 14: Predicted frames using ConvLSTM

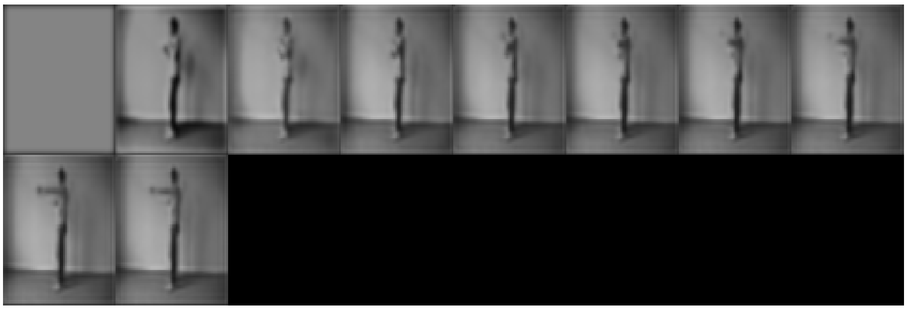


Figure 15: Predicted frames using PredRNN

Model	ConvLSTM	PredRNN
Autoencoder (Depth 2)	$1.55e-3$	0.05 (didn't converged)
PredNet	$1.95e-3$	$1.93e-3$
Spatiotemp	$3.1e-3$	0.025 (didn't converged)

Table 4: Mean MSE for KTH.

5.2.3 Kitti

Model	ConvLSTM	PredRNN
Autoencoder (Depth 2)	542.531	781.760
PredNet	8.222.559	12.430.626
Spatiotemp	1.640.321	2.200.641

Table 5: Number of trainable parameter for Kitti.

Model	ConvLSTM	PredRNN
Autoencoder (Depth 2)	0.02	0.013
PredNet	0.019	0.02
Spatiotemp	0.018	0.017

Table 6: Mean MSE for Kitti.

6 Discussion

7 Conclusion

8 Explanation

Erklärung über das selbstständige Verfassen von „Comparing different state-of-the-art solutions for image prediction using time-series analysis“
Ich versichere hiermit, dass ich die vorstehende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der obigen Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich in jedem einzelnen Fall durch die Angabe der Quelle bzw. der Herkunft, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet und anderen elektronischen Text-und Datensammlungen und dergleichen. Die eingereichte Arbeit ist nicht anderweitig als Prüfungsleistung verwendet worden oder in deutscher oder in einer anderen Sprache als Veröffentlichung erschienen. Mir ist bewusst, dass wahrheitswidrige Angaben als Täuschung behandelt werden.

Datum, Ort Unterschrift

References

[1] *Understanding LSTM Networks.* https://colah.github.io/posts/2015-08-Understanding-LSTMs/. – Accessed: 2020-07-13

[2] Abadi, Martín ; Agarwal, Ashish ; Barham, Paul ; Brevdo, Eugene ; Chen, Zhifeng ; Citro, Craig ; Corrado, Greg S. ; Davis, Andy ; Dean, Jeffrey ; Devin, Matthieu ; Ghemawat, Sanjay ; Goodfellow, Ian ; Harp, Andrew ; Irving, Geoffrey ; Isard, Michael ; Jia, Yangqing ; Jozefowicz, Rafal ; Kaiser, Lukasz ; Kudlur, Manjunath ; Levenberg, Josh ; Mané, Dandelion ; Monga, Rajat ; Moore, Sherry ; Murray, Derek ; Olah, Chris ; Schuster, Mike ; Shlens, Jonathon ; Steiner, Benoit ; Sutskever, Ilya ; Talwar, Kunal ; Tucker, Paul ; Vanhoucke, Vincent ; Vasudevan, Vijay ; Viégas, Fernanda ; Vinyals, Oriol ; Warden, Pete ; Wattenberg, Martin ; Wicke, Martin ; Yu, Yuan ; Zheng, Xiaoqiang: *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* 2015. – URL https://www.tensorflow.org/. – Software available from tensorflow.org

[3] Elsayed, Nelly ; Maida, Anthony S. ; Bayoumi, Magdy A.: Reduced-Gate Convolutional LSTM Using Predictive Coding for Spatiotemporal Prediction. In: *CoRR* abs/1810.07251 (2018). – URL http://arxiv.org/abs/1810.07251

[4] Friston, Karl: A Theory of Cortical Responses. In: *Philosophical transactions of the Royal Society of London. Series B, Biological sciences* 360 (2005), 05, S. 815–36

[5] Geiger, Andreas ; Lenz, Philip ; Stiller, Christoph ; Urtasun, Raquel: Vision meets Robotics: The KITTI Dataset. In: *International Journal of Robotics Research (IJRR)* (2013)

[6] Gers, F. A. ; Schmidhuber, J.: Recurrent nets that time and count. In: *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000.*

Neural Computing: New Challenges and Perspectives for the New Millennium Bd. 3, 2000, S. 189–194 vol.3

[7] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. MIT Press, 2016. – `http://www.deeplearningbook.org`

[8] Graves, Alex: Generating Sequences With Recurrent Neural Networks. In: *CoRR* abs/1308.0850 (2013). – URL `http://arxiv.org/abs/1308.0850`

[9] Hochreiter, Sepp ; Schmidhuber, Jürgen: Long Short-term Memory. In: *Neural computation* 9 (1997), 12, S. 1735–80

[10] Jaderberg, Max ; Simonyan, Karen ; Zisserman, Andrew ; Kavukcuoglu, Koray: Spatial Transformer Networks. In: *CoRR* abs/1506.02025 (2015). – URL `http://arxiv.org/abs/1506.02025`

[11] Kingma, Diederik P. ; Ba, Jimmy: Adam: A Method for Stochastic Optimization. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, URL `http://arxiv.org/abs/1412.6980`, 2015

[12] Kluyver, Thomas ; Ragan-Kelley, Benjamin ; Pérez, Fernando ; Granger, Brian ; Bussonnier, Matthias ; Frederic, Jonathan ; Kelley, Kyle ; Hamrick, Jessica ; Grout, Jason ; Corlay, Sylvain ; Ivanov, Paul ; Avila, Damián ; Abdalla, Safia ; Willing, Carol: Jupyter Notebooks – a publishing format for reproducible computational workflows. In: Loizides, F. (Hrsg.) ; Schmidt, B. (Hrsg.): *Positioning and Power in Academic Publishing: Players, Agents and Agendas* IOS Press (Veranst.), 2016, S. 87 – 90

[13] Lecun, Y. ; Bottou, L. ; Bengio, Y. ; Haffner, P.: Gradient-based learning applied to document recognition. In: *Proceedings of the IEEE* 86 (1998), Nr. 11, S. 2278–2324

[14] Lotter, William ; Kreiman, Gabriel ; Cox, David D.: Deep Predictive Coding Networks for Video Prediction and Unsupervised Learning. In: *CoRR* abs/1605.08104 (2016). – URL `http://arxiv.org/abs/1605.08104`

[15] Massi, Michela: *Autoencoder schema — Wikipedia, The Free Encyclopedia*. 2019. – URL `https://en.wikipedia.org/wiki/Autoencoder#/media/File:Autoencoder_schema.png`. – [Online; accessed 21-Juli-2020]

[16] Paszke, Adam ; Gross, Sam ; Massa, Francisco ; Lerer, Adam ; Bradbury, James ; Chanan, Gregory ; Killeen, Trevor ; Lin, Zeming ; Gimelshein, Natalia ; Antiga, Luca ; Desmaison, Alban ; Kopf, Andreas ; Yang, Edward ; DeVito, Zachary ; Raison, Martin ; Tejani, Alykhan ; Chilamkurthy, Sasank ; Steiner, Benoit ; Fang, Lu ; Bai, Junjie ; Chintala, Soumith: PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: Wallach, H. (Hrsg.) ; Larochelle, H. (Hrsg.) ; Beygelzimer, A. (Hrsg.) ; Alché-Buc, F. d(Hrsg.) ; Fox, E. (Hrsg.) ; Garnett, R. (Hrsg.): *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, S. 8024–8035. – URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`

[17] Patraucean, Viorica ; Handa, Ankur ; Cipolla, Roberto: Spatio-temporal video autoencoder with differentiable memory. In: *CoRR* abs/1511.06309 (2015). – URL `http://arxiv.org/abs/1511.06309`

[18] Rao, Rajesh P. N. ; Ballard, Dana H.: Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects. In: *Nature Neuroscience* 2 (1999), Jan, Nr. 1, S. 79–87. – URL `https://doi.org/10.1038/4580`. – ISSN 1546-1726

[19] Rosenblatt, Frank: The Perceptron — A Perceiving and Recognizing Automaton. In: *Tech. Rep. 85-460-1* (1957)

[20] Ruder, Sebastian: An overview of gradient descent optimization algorithms. In: *CoRR* abs/1609.04747 (2016). – URL `http://arxiv.org/abs/1609.04747`

[21] Rumelhart, David E. ; Hinton, Geoffrey E. ; Williams, Ronald J.: Learning representations by back-propagating errors. In: *Nature* 323 (1986), Oct, Nr. 6088, S. 533–536. – URL `https://doi.org/10.1038/323533a0`. – ISSN 1476-4687

[22] Schüldt, Christian ; Laptev, Ivan ; Caputo, Barbara: Recognizing human actions: A local SVM approach, 09 2004, S. 32 – 36 Vol.3. – ISBN 0-7695-2128-2

[23] SHI, Xingjian ; Chen, Zhourong ; Wang, Hao ; Yeung, Dit-Yan ; Wong, Wai-kin ; WOO, Wang-chun: Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting. In: Cortes, C. (Hrsg.) ; Lawrence, N. D. (Hrsg.) ; Lee, D. D. (Hrsg.) ; Sugiyama, M. (Hrsg.) ; Garnett, R. (Hrsg.): *Advances in Neural Information Processing Systems 28*. Curran Associates, Inc., 2015, S. 802–810. – URL `http://papers.nips.cc/paper/5955-convolutional-lstm-network-a-machine-learning-approach-for-precipitation-nowca` `pdf`

[24] Srivastava, Nitish ; Mansimov, Elman ; Salakhutdinov, Ruslan: Unsupervised Learning of Video Representations using LSTMs. In: *CoRR* abs/1502.04681 (2015). – URL `http://arxiv.org/abs/1502.04681`

[25] Sutskever, Ilya ; Vinyals, Oriol ; Le, Quoc V.: Sequence to Sequence Learning with Neural Networks. In: *CoRR* abs/1409.3215 (2014). – URL `http://arxiv.org/abs/1409.3215`

[26] Wang, Yunbo ; Gao, Zhifeng ; Long, Mingsheng ; Wang, Jianmin ; Yu, Philip S.: PredRNN++: Towards A Resolution of the Deep-in-Time Dilemma in Spatiotemporal Predictive Learning. In: *CoRR* abs/1804.06300 (2018). – URL `http://arxiv.org/abs/1804.06300`

[27] Wang, Yunbo ; Long, Mingsheng ; Wang, Jianmin ; Gao, Zhifeng ; Yu, Philip S.: PredRNN: Recurrent Neural Networks for Predictive Learning using Spatiotemporal LSTMs. In: *NIPS*, 2017

[28] Werbos, Paul: Backpropagation through time: what it does and how to do it. In: *Proceedings of the IEEE* 78 (1990), 11, S. 1550 – 1560

[29] Zhao, H. ; Gallo, O. ; Frosio, I. ; Kautz, J.: Loss Functions for Image Restoration With Neural Networks. In: *IEEE Transactions on Computational Imaging* 3 (2017), Nr. 1, S. 47–57

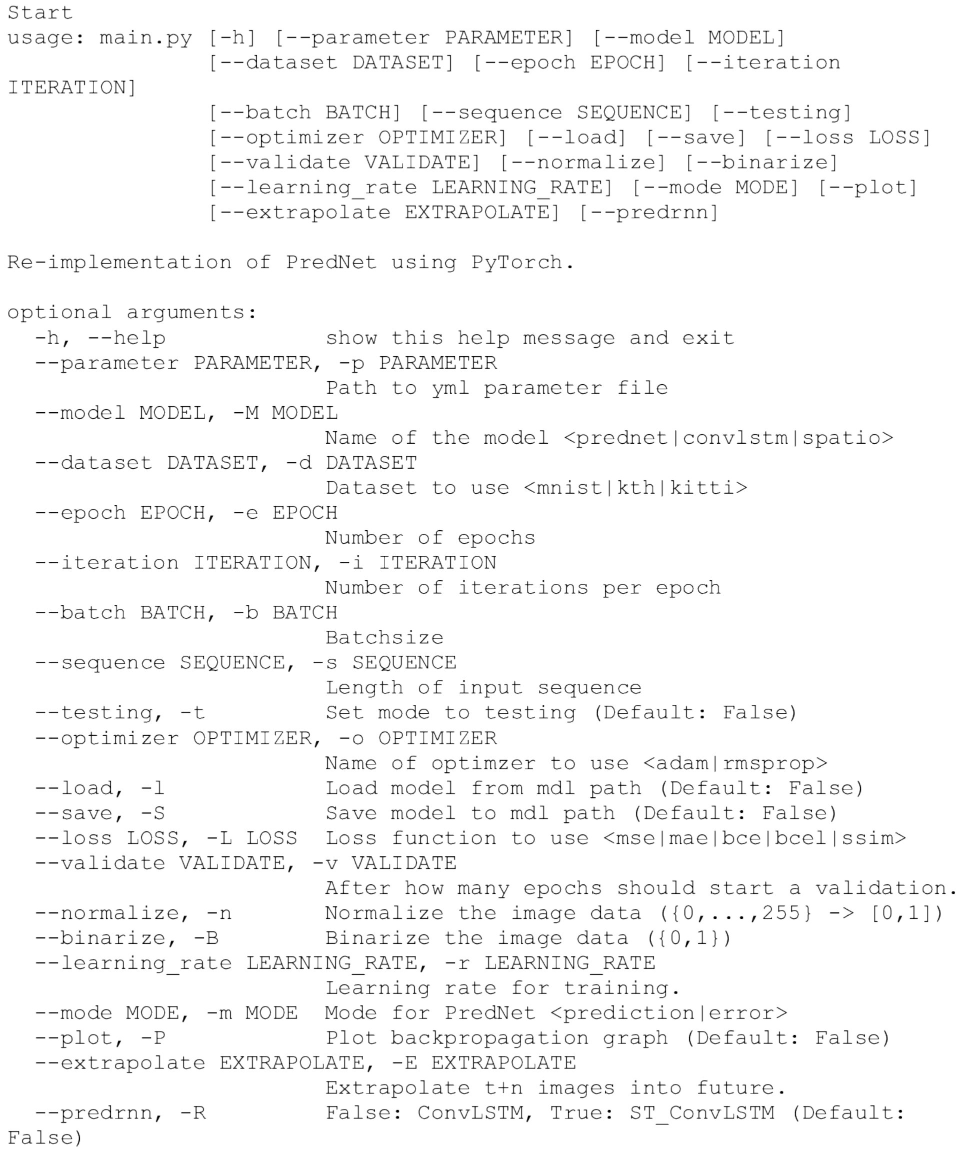
Appendix

Code structure

```
PredNet
├── data
│   ├── files
│   │   ├── caltech_test.hkl
│   │   ├── caltech_test_sources.hkl
│   │   ├── caltech_train.hkl
│   │   ├── caltech_train_sources.hkl
│   │   ├── kitti_sources_test.hkl
│   │   ├── kitti_sources_train.hkl
│   │   ├── kitti_sources_val.hkl
│   │   ├── kitti_X_test.hkl
│   │   ├── kitti_X_train.hkl
│   │   ├── kitti_X_val.hkl
│   │   ├── kth_dev.hkl
│   │   ├── kth_test.hkl
│   │   ├── kth_train.hkl
│   │   ├── moving_mnist_test.hkl
│   │   └── moving_mnist_train.hkl
│   └── scripts
│       ├── 00sequences.txt
│       ├── data_utils_caltech.py
│       ├── data_utils_kth.py
│       ├── download_caltech.sh
│       ├── download_kth.sh
│       └── moving_mnist.py
├── dataset
│   ├── Caltech.py
│   ├── __init__.py
│   ├── Kitti.py
│   ├── Kth.py
│   └── MovingMNIST.py
├── graph
├── helper
│   ├── arguments.py
│   ├── datetime.py
│   ├── __init__.py
│   ├── logger.py
│   ├── loss.py
│   ├── mdl.py
│   ├── plot_graph.py
│   ├── tensorboard_log.py
│   ├── transformation.py
│   └── yaml_parser.py
├── __init__.py
├── log
│   ├── convlstm
│   ├── prednet
│   └── spatiotemp
├── main.py
├── mdl
├── model
│   ├── autoenc.py
│   ├── __init__.py
│   ├── modules
│   │   ├── activation.py
│   │   ├── conv_lstm.py
│   │   ├── error.py
│   │   ├── grid_generator.py
│   │   ├── huber_penalty.py
│   │   ├── __init__.py
│   │   ├── input.py
│   │   ├── optical_flow.py
│   │   ├── prediction.py
│   │   ├── predrnn.py
│   │   └── spatial_autoencoder.py
│   ├── prednet.py
│   └── spatiotemp.py
├── output.txt
├── README
├── requirements.txt
├── setup.py
├── test.py
├── train.py
├── tree.txt
├── validate.py
└── yml
    ├── convlstm_kitti.yml
    ├── convlstm_kth.yml
    ├── convlstm_mnist.yml
    ├── prednet_kitti_l0.yml
    ├── prednet_kitti_lall.yml
    ├── prednet_kth_l0.yml
    ├── prednet_kth_lall.yml
    ├── prednet_mnist_l0_diff.yml
    ├── prednet_mnist_l0.yml
    ├── prednet_mnist_lall_diff.yml
    ├── prednet_mnist_lall.yml
    ├── spatiotemp_kitti.yml
    ├── spatiotemp_kth.yml
    └── spatiotemp_mnist.yml
```

Figure 16: Tree of Implementation

Code usage



```
Start
usage: main.py [-h] [--parameter PARAMETER] [--model MODEL]
               [--dataset DATASET] [--epoch EPOCH] [--iteration
ITERATION]
               [--batch BATCH] [--sequence SEQUENCE] [--testing]
               [--optimizer OPTIMIZER] [--load] [--save] [--loss LOSS]
               [--validate VALIDATE] [--normalize] [--binarize]
               [--learning_rate LEARNING_RATE] [--mode MODE] [--plot]
               [--extrapolate EXTRAPOLATE] [--predrnn]

Re-implementation of PredNet using PyTorch.

optional arguments:
  -h, --help            show this help message and exit
  --parameter PARAMETER, -p PARAMETER
                        Path to yml parameter file
  --model MODEL, -M MODEL
                        Name of the model <prednet|convlstm|spatio>
  --dataset DATASET, -d DATASET
                        Dataset to use <mnist|kth|kitti>
  --epoch EPOCH, -e EPOCH
                        Number of epochs
  --iteration ITERATION, -i ITERATION
                        Number of iterations per epoch
  --batch BATCH, -b BATCH
                        Batchsize
  --sequence SEQUENCE, -s SEQUENCE
                        Length of input sequence
  --testing, -t         Set mode to testing (Default: False)
  --optimizer OPTIMIZER, -o OPTIMIZER
                        Name of optimzer to use <adam|rmsprop>
  --load, -l            Load model from mdl path (Default: False)
  --save, -S            Save model to mdl path (Default: False)
  --loss LOSS, -L LOSS  Loss function to use <mse|mae|bce|bcel|ssim>
  --validate VALIDATE, -v VALIDATE
                        After how many epochs should start a validation.
  --normalize, -n       Normalize the image data ({0,...,255} -> [0,1])
  --binarize, -B        Binarize the image data ({0,1})
  --learning_rate LEARNING_RATE, -r LEARNING_RATE
                        Learning rate for training.
  --mode MODE, -m MODE  Mode for PredNet <prediction|error>
  --plot, -P            Plot backpropagation graph (Default: False)
  --extrapolate EXTRAPOLATE, -E EXTRAPOLATE
                        Extrapolate t+n images into future.
  --predrnn, -R         False: ConvLSTM, True: ST_ConvLSTM (Default:
False)
```

Figure 17: Usage of Implementation

Class diagram

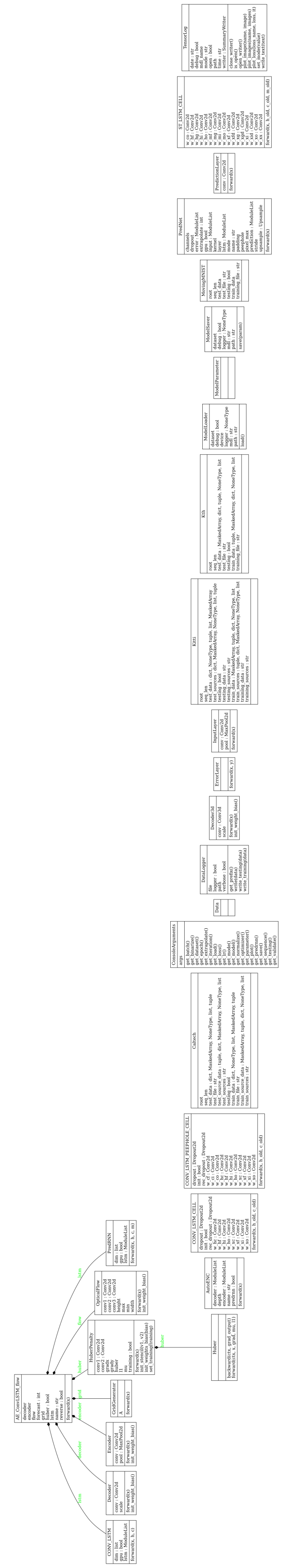


Figure 18: Class diagram, automatically provided by pyreverse. (pyreverse is not able find connections if loops are used.)

Package diagram

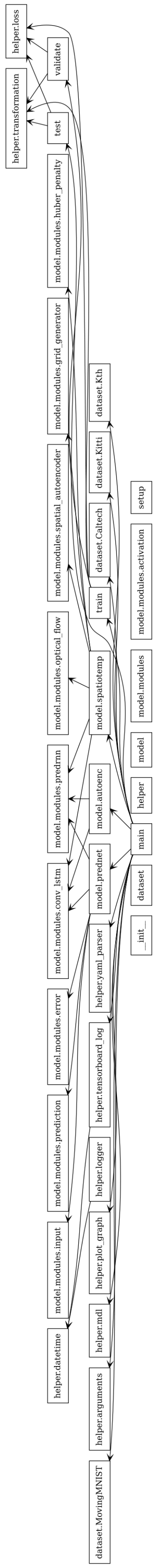


Figure 19: Package diagram, automatically provided by pyreverse