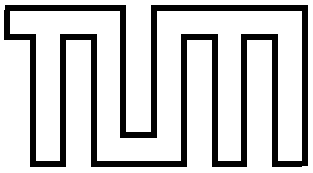


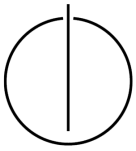
DEPARTMENT OF INFORMATICS

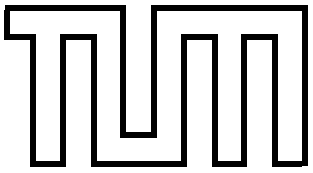
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Computer Science

Deep Learning Approaches to Predict Future Frames in Videos

Benjamin Sautermeister





DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Computer Science

Deep Learning Approaches to Predict Future Frames in Videos

Herangehensweisen in Deep Learning zur Vorhersage von Zukünftigen Bildern in Videos

Author:	Benjamin Sautermeister
Supervisor:	Prof. Dr. Daniel Cremers
Advisor:	Philip Häusser
Submission Date:	17th October, 2016

I confirm that this master's thesis in computer science is my own work and I have documented all sources and material used.

Munich, 17th October, 2016 Benjamin Sautermeister

Acknowledgments

I would like to express my sincere thanks to those who offered feedback on the content or made this thesis even possible. Beginning with *Prof. Dr. Daniel Cremers* for his valuable suggestions at the very beginning of my work. Next, my advisor *Philip Häusser* for offering this interesting topic, his continuous support as well as our fruitful discussion at Google Zurich. Furthermore, I would like to acknowledge *PD Dr. habil. Rudolph Triebel* for his tough but outstanding lecture *Machine Learning in Computer Vision* at TUM, which initiated my interest to do a thesis in this field.

Finally, I would also like to thank my family for their faith and positive mentality, who waited patiently for me to finish this work. And last but not least, *Nicole Frangi Doetzer*, *Tom Kingdom*, *Stefan Kraus*, *Patrick Mutter*, *Timo Partl*, *Andrej Prescher*, *Manuel Sautermeister* and *Lukas Wöhrl* in terms of investing their time for proofreading this thesis.

Abstract

Deep neural networks are becoming central in several areas of computer vision. While there has been a lot of research regarding the classification of images and videos, future frame prediction is still a rarely investigated approach, and even though many applications could make good use of the knowledge regarding the next frame of an image sequence in pixel-space. Examples include video compression and autonomous agents in robotics that have to act in natural environments. In fact, learning how to forecast the future of an image sequence requires the system to understand and efficiently encode the content and dynamics for a certain period of time. It is viewed as a promising avenue in which even supervised tasks could benefit from, since labeled video data is limited and hard to obtain. Therefore, this work gives an overview of scientific advances covering future frame prediction and proposes a recurrent network model which utilizes recent techniques from deep learning research. The presented architecture is based on the recurrent decoder-encoder framework with convolutional cells, which allows the preservation of spatio-temporal data correlations. Driven by perceptual motivated objective functions and a modern recurrent learning strategy, it is able to outperform existing approaches with respect to future frame generation in several video content types. All this can be achieved with fewer training iterations and model parameters.

Contents

1 Introduction

Since the classical era, people have dreamed of inventing machines that can act and think like humans. This opened the field of *artificial intelligence* (AI), which is still an active research topic and is used in many practical applications. The main focus of AI in early days was to solve problems that are hard to solve for humans, such as finding the shortest path to an arbitrary destination using the well-known *Dijkstra algorithm*[1]. Ironically, it turned out that tasks which can be solved by humans using pure intuition are actually extremely hard for computers to solve. As an example, it is hard or even impossible to write a program from scratch that is able to detect objects in pictures, recognize words in spoken text or to describe the events in a video scene. The reason is that classical computer programs in contrast have to be algorithmically expressed as a sequence of commands or a list of mathematical rules [GBC16]. But it is quite tough to apply this on multi-dimensional data such as pictures or videos that consists of an incoherent set of pixels including different color channels with a lot of noise and countless possibilities.

Humans handle this kind of data differently. They learn to recognize objects by experience and implicitly build hierarchies of relationships in their mind. This basic principle opened a new subfield, known as *machine learning* (ML). It covers a methodology where knowledge is acquired by extracting patterns from raw data and consequently allows to make reasonable decisions [GBC16]. But while this is able to cover many previously unsolvable problems, it requires that one can tell which features we would like to investigate, for instance to build a decision tree out of it. Coming back to our previous example, this is still hard to be applied on images or video data where it is known which features we are actuall looking for, but still cannot formally describe how these are represented. A field that deals with this issue is called *representation learning*, which tries to automatically build the representation by itself.

Having just a high-level representation might still not be enough. To break down the problem, *artificial neural networks* (ANN) have been introduced. They are biologically inspired by the structure of the human brain [Kar12] and can be trained to learn hierarchies of representations. For object recognition on images, one can think of edges that are detected on a very low level, which will be further composed to curves or shapes. Furthermore, these simple structures might be compound in a specific way,

[1] Also known as uniform-cost search

so that the neural network can identify distinct complex objects in it. The rise of computational power allows to create networks that are even deeper and therefore learn more and more complex representations hierarchies. This principle caused its contemporary name, known as *deep learning*.

1.1 Motivation

In recent years, the field of deep learning achieved considerable success and according to its underlying philosophy: "*if we have a reasonable end-to-end model and sufficient data for training it, we are close to solving the problem.*" [Shi+15]. But while there has been a lot of studies and practical applications of object recognition on static images or speech recognition, the application of these concepts on video data are just about to make their first steps in research.

Early deep learning approaches dealing with video data or simple image sequences address problems like human action recognition [Ji+13], [SZ14], [Don+14] or video classification [Kar+14]. Another example is optical flow prediction [Fis+15] in order to detect the visual flow from one frame to the next. Most of these approaches require lots of labeled data to be able to train a network. The effortful labeling process and thus the resulting low availability of such data might be the main reason why this topic has not been covered that well so far. On the contrary, online services like *YouTube* provide a seemingly endless, but unlabeled source of videos to learn from.

1.2 Problem Statement

Throughout this work, is is investigated whether deep learning techniques can be successfully applied on videos to learn a meaningful representation in a completely unsupervised fashion. In detail, it is examined if such a representation is suited to continue a video even after it has finished. Hence, to learn a notion of the spatial and temporal evolution within a sequence of images as well as to get an idea of motion and dynamics of a scene. Such a high-level understanding would be helpful for autonomous intelligent agents that have to act and therefore understand our environment including its physical and temporal constrains [SMS15]. Other application areas might be for instance video compression [ABP05], visual systems for autonomous cars or as a replacement for optical flow in causal video segmentation [Cou+13]. Aside from that, other supervised learning tasks like human action recognition could benefit from such a pre-trained network in order to improve the overall performance or to reduce the training time. Needless to say, other forms of *transfer learning* are easily conceivable as well.

Figure 1.1: Example of an image sequence with an unknown future frame. The sequence is starting from the left and is taken from UCF-101.

Like in the first example of object recognition on static pictures, this task might sound trivial for humans once more, since we already have built an intuition regarding motion and our environment. When we have a look at the image sequence in Figure 1.1, we have a strong idea about how this sequence might continue. At least for a couple of time steps. The boy in the foreground probably will lift his left foot towards the ball, while the ball continues to fall down due to gravitation. In contrast, the background will stay almost unchanged.

Developing a deep learning approach to tackle this is indeed a non-trivial task, since it has to model both spatial and temporal features in combination, as well as the search space grows exponential in case of multi-step forecasting. Additionally, related issues like evolving an effective training process or quantifying the perceptual image similarity between predicted and the ground truth frames have to be addressed as well. Moreover, existing state-of-the-art implementations that deal with frame prediction have to be reviewed and analyzed in detail in order to learn from their strength and weaknesses.

1.3 Contributions

This thesis consists of several contributions. Firstly, it provides a dense *overview* of existing deep learning approaches that deal with the problem of future frame prediction in videos. Secondly, it presents a neural network architecture that combines modern practices like *batch normalization* with a novel *convolutional LSTM* implementation and *scheduled sampling* to improve training of recurrent models. Thereby, it is able to *cut in half the prediction error* of other state-of-the-art models in the MovingMNIST dataset. Last but not least, all TensorFlow implementations are freely available to the research community, including the scheduled sampling and batch normalization enabled convolutional recurrent cell, as well as several metrics and loss functions to measure perceptual image similarity at training time. Furthermore, a *lightweight, high-level and open source framework for TensorFlow* is contributed that is able to radically reduce boilerplate code of deep learning applications. This is facilitated by providing an abstraction for many recurring or complex tasks that have to be faced while building and training neural network models.

1.4 Organization

The subsequent chapters of this thesis are structured as followed:

Chapter 2 covers the theoretical concepts that are required to understand our final implementation and its consecutive evaluation. It provides a deeper look into neural networks and explains how they are trained. Further, more advanced neural network architectures are explored, namely *convolutional neural networks* (CNN) for spatial learning and *recurrent neural network* (RNN) models for sequential learning. Afterwards, this chapter concludes with the investigation of some modern techniques that are used to improve the overall learning process, as well as metrics for perceptual motivated image similarity assessment.

In **Chapter 3**, a closer look is taken at existing approaches that are suitable for spatio-temporal learning and frame prediction. It briefly discusses their strength and weaknesses, as well as how they have influenced the design decisions regarding the architecture of the final neural network model. Additionally, these models build the baselines in the evaluation.

Afterwards, the implemented neural network model is presented in **Chapter 4**, including its architecture and implementation details. Moreover, a quite recently invented and barely studied variant of recurrent network cell for spatio-temporal learning, namely *convolutional LSTM*, is introduced which builds the central element of the realized model. It also suggests a special learning strategy for RNNs, which speeds up the training process and enables improvement of the overall performance.

Chapter 5 gives a brief overview of the video datasets used within this thesis. All in all, three different sets with increasing complexity have been chosen, namely *Moving MNIST*, *MsPacman* and *UCF-101*.

Next, **Chapter 6** illustrates many experimental results on the previously named datasets. It investigates how changes in the model or hyperparameters do affect the model's performance, as well as compares the results with other existing approaches in detail.

Within **Chapter 7**, a lightweight, high-level framework for *TensorFlow* is presented as a side contribution that has grown out of this project.

In the end, this thesis is concluded in **Chapter 8** by summarizing the overall results, as well as highlight the identified possible improvements for future work.

2 Fundamentals

To get a general understanding of how training a neural network works, this chapter goes through its theoretical concepts first. It starts with the structure of simple feed-forward networks, continue with advanced model architectures that take advantage of the data's spatial or temporal properties. And finally it ends up with recent techniques that are used throughout the final implementation.

2.1 Neural Networks

The main concept of *neural networks* (NN) dates back to the early 1950s, when Warren McCulloch and Walter Pitts tried to build a mathematical model of information processing in our brain. Inspired by this work, Frank Rosenblatt developed the so called *perceptron* about two decades later [Bis06, p. 226].

2.1.1 Basics

The perceptron itself has quite a simple structure. It is usually visualized as a node that has any number of binary inputs x_i, as well as a single output y with $x_i, y \in \{0, 1\}$. In addition, each input is weighted by $w_i \in \mathbb{R}$ to express the importance of each particular input. The output is determined by the simple rule that the weighted sum of all inputs has to reach a specified threshold to make the perceptron fire its output [Nie15]. This threshold is usually called bias $b \in \mathbb{R}$, defined as the negative threshold. All of this can be expressed as follows:

$$y = \begin{cases} 1, & \text{if } \sum_{i=1}^{n} w_i \, x_i + b > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{2.1}$$

Even though its formulation is that simple, it can represent complex decision-making when multiple elements are stacked together, known as a multilayer perceptron (MLP). Such a network forms a *directed acyclic graph* (DAG) and is illustrated in Figure 2.1.

The first and last layer of such a network are referred to as *input layer* and *output layer*. Furthermore, the number of nodes is determined by the given problem to solve. In case

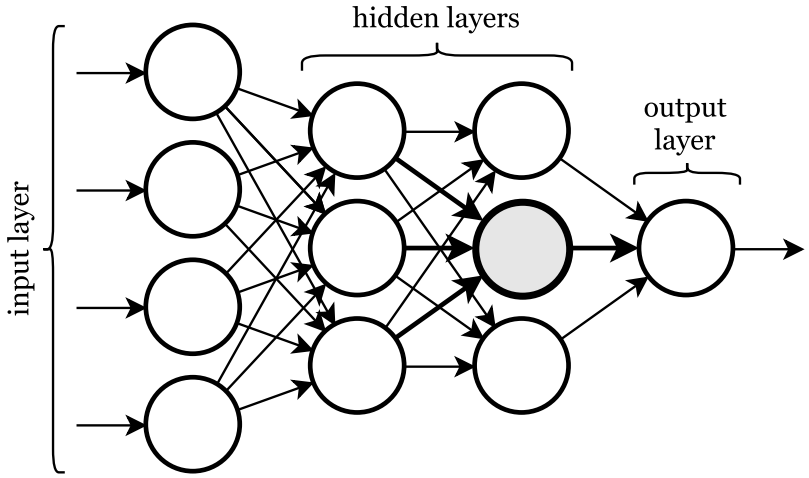


Figure 2.1: Example of a MLP with two hidden layers. A single perceptron is highlighted in bold. (Based on [Nie15])

we want to train a network that identifies human faces in RGB-colored pictures with height and width of 100 pixels, it would require our input layer to have n_{in} = 30,000 perceptrons, as well as a single output node. In contrary, all intermediate layers are known as *hidden layers* and can have any number of elements and depth. When every node from one layer is connected to all nodes of its subsequent layer, it is called *fully connected* (FC).

Afterwards, the input layer can be fed with a data example and apply equation 2.1 in each node to retrieve our binary result. This prediction step is called *inference*. But in order to retrieve meaningful results, the network has to be trained first to have appropriate values for the weights and biases.

2.1.2 Network Training

The final goal of training such a network is to end up with a model that generalizes on any kind of data from the same type [Bis06, p. 2]. Data that is used during this process is called *training set*, the other portion of data that evaluates its generalization capabilities *test set*. Additionally, a third split is preferably used during the training process of the networks to select the best performing approach. It is known as the *validation set*. Since the ground truth outcome of each data example during the training

phase is known, we can quantify the outcomes using a loss function[1], such as *mean absolute error* (MAE)[2]:

$$\mathcal{L}_{\text{mae}}(\mathbf{w}, \mathbf{b}) = \frac{1}{n} \sum_{\mathbf{x}} |y(\mathbf{x}) - t(\mathbf{x})|, \tag{2.2}$$

mean squared error (MSE)[3]:

$$\mathcal{L}_{\text{mse}}(\mathbf{w}, \mathbf{b}) = \frac{1}{n} \sum_{\mathbf{x}} (y(\mathbf{x}) - t(\mathbf{x}))^2, \tag{2.3}$$

or *binary cross-entropy* (BCE) [Shi+15]:

$$\mathcal{L}_{\text{bce}}(\mathbf{w}, \mathbf{b}) = -\frac{1}{n} \sum_{\mathbf{x}} t(\mathbf{x}) \cdot \log\left(y(\mathbf{x})\right) + \left(1 - t(\mathbf{x})\right) \cdot \log\left(1 - y(\mathbf{x})\right), \tag{2.4}$$

where n is the number of examples and $t(\mathbf{x})$ denotes a mapping from an input example $\mathbf{x}$ to its ground truth target. Many other functions exist and some more will be introduced in Section 2.6.3, but the above listed formulas are the main objectives that are used in many other works. During training of the network, the set of weights $\mathbf{w}$ and biases $\mathbf{b}$ have to be found that minimizes the error:

$$\arg\min_{\mathbf{w},\mathbf{b}} \mathcal{L}(\mathbf{w}, \mathbf{b}). \tag{2.5}$$

Parameters beside $\mathbf{w}$ and $\mathbf{b}$ that are not learned during this process are called *hyperparameters*. Examples of such non-trainable parameters are the number of layers or the size of each single hidden layer. More hyperparameters will arise throughout this chapter.

Neurons and Activations

At this point, the fundamental problem of perceptrons is faced. In order find the best set of parameters, small changes in the model's weights $\mathbf{w}$ and biases $\mathbf{b}$ have to be performed to justify the output into the right direction of the desired outcome. But since the perceptron's output is discrete, a small change can cause a sudden flip in the

[1] Often called cost function, objective function or error function as well.

[2] Also known as ℓ_1 when it does not average over all examples, but often used as a synonym. In this work, the averaged variants for all presented functions are always used. Additionally, the final implementation averages across the image pixels when any loss function is applied on images to achieve pixel-wise results that are independent regarding the image dimensions later on in context of frame prediction.

[3] Also referred to as ℓ_2 when no averaging across all examples is performed.

overall output of the model. To overcome this issue, these perceptrons are replaced with *neurons*. The exemplary structure of a neuron is illustrated in Figure 2.2. They are given by:

$$\begin{aligned} z &= \sum_{i=1}^{n} w_i \, x_i + b \\ y &= \phi(z), \end{aligned} \tag{2.6}$$

which allow $x_i, y \in \mathbb{R}$ by wrapping its term with a non-linear *activation function* $\phi(z)$. Frequently used examples are the sigmoid function $\sigma(z)$, hyperbolic tangent $tanh(z)$ and the rectified linear unit (ReLU) $max(0, z)$, illustrated in Figure 2.3.

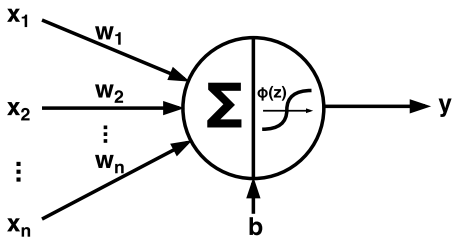


Figure 2.2: Schematic structure of a neuron with its n inputs x_i, weights w_i, bias b and activation function $\phi(z)$.

Note that the sigmoid function's shape is a smoothed out variant of the *step function* [Nie15], which can be used to make a neuron act like a classical perceptron. Additionally, the rectifier differs to both other activation functions in that it is one-sided and partly linear. Even though its shape looks much simpler, it became one of the most favorable activation function for intermediate layers in deep neural networks[4]. The reasons are that it allows faster computation, sparse activation[5], reduces the likelihood of vanishing gradient (see Section 2.3.1) and is more biologically plausible [GBB11b].

Initialization

Before starting the training process, an initial value is assigned to each variable **w** and **b**. This is done by pure randomness, using for example a uniform or Gaussian distribution. But if we start with weights that are too small, the signal could decrease so much that it is too small to be useful. On the other side, when the parameters are initialized with high values, the signal can end up to explode while propagating through the network

[4] List of ReLU's strength and weaknesses: `http://cs231n.github.io/neural-networks-1/`

[5] A sparse activation means that only half of the neurons have an initial non-zero output, when a uniform initialization is used.

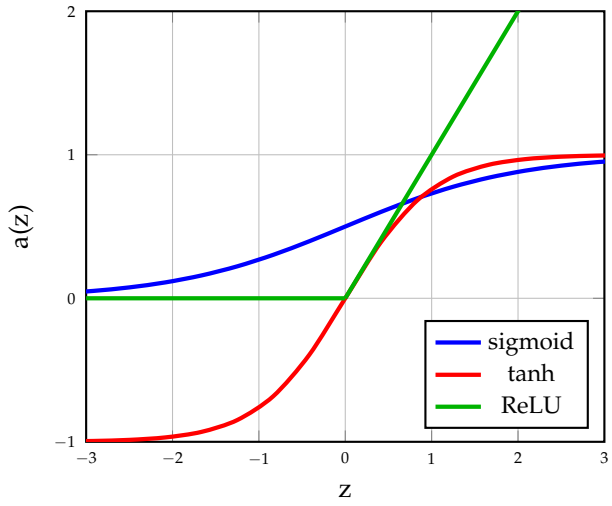


Figure 2.3: Visualization of the most commonly used activation functions in neural networks.

[Jon15]. In consequence, a good initialization can have a radical effect on how fast the network will learn useful patterns.

For this purpose, some best practices have been developed. One famous example used in our final model is *Xavier initialization*[6] (see eq. 2.7). Its formulation is based on the number of input and output neurons and uses sampling from a uniform distribution with zero mean and all biases set to zero [GB10]:

$$\mathbf{w} \sim \mathrm{U}\left[-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right], \tag{2.7}$$

where $\mathbf{w}$ is the weight matrix at any network layer, n_{in} the number of incoming connections and n_{out} the number of outgoing connections to the next layer. This initialization is designed to keep the gradients in all layers within approximately the same scale.

As an alternative to random initialization and performing a training of the entire network from scratch, it is also possible to reuse parts or even all trained parameters from a different model. A famous example that is often used to pre-initialize a network for image processing tasks is *AlexNet* [KSH12]. It is pre-trained for several weeks across multiple graphic cards on the large *ImageNet*[7] dataset that contains 1.2 million images.

[6] Also known as Glorot initialization.
[7] ImageNet dataset: `http://image-net.org/`

Backpropagation Learning Algorithm

To actually train the network by minimizing its error (see eq. 2.5), a learning algorithm called *backpropagation* is applied. This algorithm is based on *gradient descent*, which iteratively tries to find the minima of a function by doing small steps towards the negative gradient. Applying this to the given loss function results in the *update rule* for any trainable weight and bias parameter:

$$\begin{aligned} w_i^{(\tau+1)} &= w_i^{(\tau)} - \eta \cdot \frac{\partial \mathcal{L}}{\partial w_i^{(\tau)}} \\ b_j^{(\tau+1)} &= b_j^{(\tau)} - \eta \cdot \frac{\partial \mathcal{L}}{\partial b_j^{(\tau)}}, \end{aligned} \tag{2.8}$$

where $\eta > 0$ is the *learning rate* that determines the step size that is done along the slope in each iteration [Bis06]. In other words, we proceed backwards through our network in every training iteration and slightly adjust every parameter depending on how much it has contributed to the error. Doing a single step by computing the gradients for the whole training set would require too much time and memory resources. Hence, the gradients are estimated over the whole population by using a smaller sample. This technique is called *stochastic gradient descent* (SGD), whereas the size of the sample is known as *batch size*.

Although this algorithm is really powerful, it comes with some disadvantages that have to be kept in mind. First, the result can converge to any local minimum. In consequence, finding a global minimum is not guaranteed. Secondly, depending on the choice of the learning rate η, the algorithm might converge very slowly or even not at all [Kar12].

Beside SGD, many other advanced gradient descent-based optimization algorithms exist. Detailed explanations and visualizations can be found in [Rud16]. The optimizer that is used in this thesis is called *adaptive moments estimation* (Adam). This algorithm is based on adaptive estimates of lower-order moments and performs a form of step size annealing by using exponential moving averages of the parameters. Additionally, its hyperparameters $\beta_1, \beta_2 \in [0, 1)$ have an intuitive interpretation and control the decay rates of the previous mentioned moving averages. Therefore, it usually requires less tuning of the learning rate or its other hyperparameters, and has shown to work very well in practice [KB15].

Stopping Criteria

The training process could basically run endless. Therefore, a rule should be defined when to stop it. There are many options when to cancel the training. Also, combinations of different *stopping criteria* are possible. These can be for example:

- When the validation loss does not decrease (for a specified number of iterations).
- When the change in loss falls below a defined threshold (for a specified number of iterations).
- When a fixed number of steps or epochs[8] elapses.
- When a defined timeframe exceeds.

2.1.3 Regularization

As already stated, our goal is to find a representation that generalizes well. One common problem that has to be prevented when neural networks are trained is the effect of *overfitting*. This means that even when the training loss decreases further and further, the validation and test error suddenly starts to get worse. One cause might be that the size of the training set is not large enough. But to come up with more data is often not possible. Another reason might be that our *model complexity*[9] is too high. To get an idea about the reason for this, imagine we want to fit a function $g(x)$ using some noisy data points of a ground truth function $f(x)$. When our model exhibits to many parameters, it might come up with a function that perfectly fits to all given data points. Nevertheless, as demonstrated in Figure 2.4, this is a bad estimate of the underlying function $f(x)$.

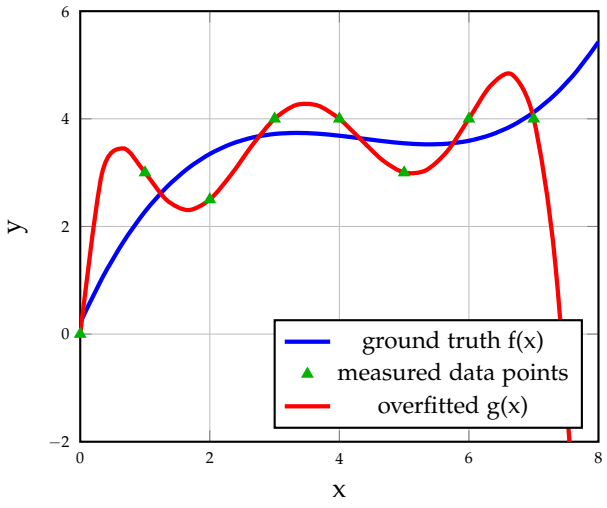


Figure 2.4: Visualization of an overfitted function.

On the other hand, a reduction of model complexity can also be a false conclusion because this limits the potential power of the network. Fortunately, research has

[8] A single epoch is usually defined as the number of steps that is required to iterate over the whole training set.

[9] The complexity of a model is defined by the number of trainable parameters.

originated different methods to master this issue. In the field of machine learning, these methods are referred to as *regularization* techniques.

A well-known technique to delimitate overfitting is to penalize high parameter values which cause the oscillation effect that can be seen in Figure 2.4. Therefore, the loss function is extended with an additional regularization term. This method is called *weight decay*:

$$\mathcal{L}_{total}(\mathbf{w}, \mathbf{b}) = \mathcal{L}(\mathbf{w}, \mathbf{b}) + \frac{\lambda}{n} \sum_{\mathbf{w}} \mathbf{w}^2, \tag{2.9}$$

where the coefficient λ controls the influence of the regularization. The term shown in equation 2.9 uses an ℓ_2 regularizer over all weights, which strongly penalizes a high magnitude of values. Together with the learning rate η, both define two of the usually most significant hyperparameters in any neural network. Finding appropriate values is a major task when fine-tuning a model.

A second regularization approach is known as *dropout*. Instead of modifying the cost function, it manipulates a specific layer of the model by randomly deactivating a neuron with a probability p in every training step. As a result, the networks gets robust against distinct patterns that cause a high activation towards a certain output. Stated differently, the network is forced to not learn any shortcut that could damage generality. It is appropriate to add that no neuron is deactivated during inference. But to compensate the larger amount of active neurons within the layer, all weights of outgoing connections will be multiplied by factor p [Sri+14, p. 1931]

2.2 Convolutional Neural Networks

In the previous section, neural networks have been discovered that exhibit a full connection of neuros from one layer to the next. While this allows to learn complex representations on the one hand, it comes with a couple of downsides on the other hand as well. For example, data such as images would require the layers of the network to become very large, especially in consideration of the input layer. Consequently, the number of connections between these layers would increase exponentially and thus the amount of trainable parameters as well. At the bottom line, this would end up in a network that is either time-consuming to train, or that is not even able to be stored in memory. In addition, it would not take any advantage of local image properties into account.

Therefore, a new network type found attention in recent years, which is known as *convolutional neural network* (CNN). It is inspired by the animals' visual cortex, has already been used in the late 1990s to solve optical character recognition tasks (OCR)

[LeC+98] and received its main attention after beating proven methods in the *ImageNet* competition by a large margin [KSH12]. The structure of a convolutional network, the detailed advantages and its mathematical formulation are described in the following sections.

2.2.1 Structure

A network is called CNN if it consists of at least one convolutional layer. In other words, "*convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.*" [GBC16]. The definition of the convolution operation follows in Section 2.2.2. Simply put, imagine a small window that slides across the whole input space. In every iteration step, it attempts to extract features that are only dependent on a small neighboring region with the size of this window. Moreover, the location of features that it tries to detect is not fixed to any specific spot, as it treats every patch in the same way. In every convolutional layer, this process is repeated several times, resulting in multiple feature maps. Figure 2.5 visualizes the described structure of a simplified convolutional neural network.

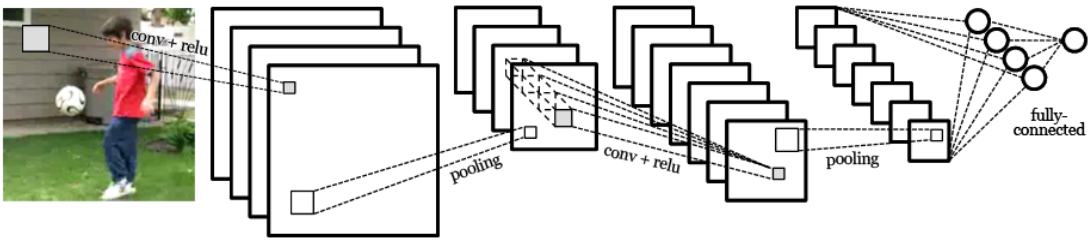


Figure 2.5: Example of a simplified CNN structure with two convolutional layers for image classification. (Based on [LeC+98, p. 2284])

The window mentioned before is called *kernel* and holds the randomly initialized parameters that the network can learn. The kernel acts as a filter that is applied to each location in regular steps. In the two dimensional case, the kernel has a specified width and height, denoted as the *kernel size*. Several kernels are used to extract multiple feature maps in each convolutional layer, but each output feature map is computed with its own kernel. This number of kernels is specified with its *kernel depth*. Furthermore, the range the filter is moved in each dimension per step is called *stride* [DV16].

Each convolutional layer is usually followed by a non-linear activation function, preferably a rectifier. The reason is that the convolution is an affine transformation and is therefore linear. Stacking multiple linear operations could be mathematically reduced to a single one. Optionally, an additional *pooling layer* can be applied that performs a subsampling onto the feature maps. Several pooling variants exist, while *max pooling*

is probably the most frequently used of them. It allows the representation to become roughly invariant to small rotations or translations of the input by only using the maximum value [GBC16, p. 343].

2.2.2 Convolution Operation

Generally speaking, a convolution is a mathematical operation on two functions $f(x)$ and $g(x)$. Its operator is typically denoted with an asterisk [GBC16, p. 332] and is defined as:

$$(f * g)(x) = \int f(\tau) \cdot g(x - \tau)\, d\tau. \tag{2.10}$$

In terminology of convolutional networks, the function f is termed as the *input* and the filter g is referred to as the *kernel*. Moreover, the output of $(f * g)(x)$ is called a *feature map*.

As this thesis is mainly dealing with discrete 2D images, the formulation of equation 2.10 can be discretized and reformulated as:

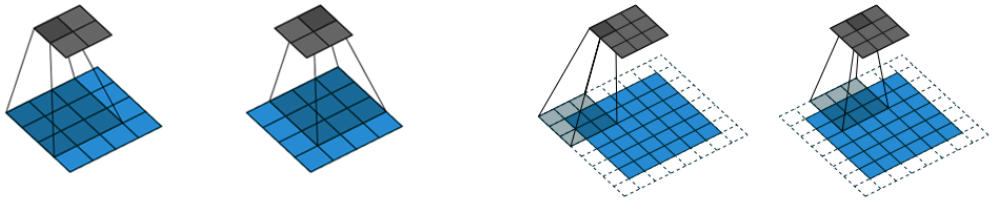
$$(\mathbf{I} * \mathbf{K})(x, y) = \sum_{r=1}^{h} \sum_{c=1}^{w} \mathbf{I}_{c,r} \cdot \mathbf{K}_{x-c,y-r}, \tag{2.11}$$

with an input $\mathbf{I}$ of size $w \times h$ and a two-dimensional kernel $\mathbf{K}$. Depending on the width and height of the kernel with a windows size of $k \times k$ and the chosen stride s, the shape of the convolved output changes. This is why the input is often enriched with zeros in order to have more control regarding the resulting output size. Surrounding the data with zeros is also known as *zero-padding*. The use of no padding ($p = 0$) is also called *valid padding*, as depicted in Figure 2.6a. Also, when a padding of $p = \lfloor k/2 \rfloor$ is used that is half the kernel size, it is referred to as *same padding*, shown in Figure 2.6b. The reasons for its name is caused by the fact that the input and output size stay unchanged when a stride of $s = 1$ is used.

It must be noted that the size of the kernel, padding and stride does not necessarily have to be equal in each dimension. But nevertheless, this is often the case in many practical applications.

2.2.3 Transposed Convolution Operation

The application of the previously presented convolution operation usually transforms the input into lower-dimensional feature maps. However, there are use cases where we would like to go the other way round, while keeping the connectivity pattern



(a) $p = 0$ (valid), $s = 1$ (b) $p = 1$ (same), $s = 2$

Figure 2.6: Visualizations of the convolutional operation with an 3×3 kernel but different settings for padding and stride. The white squares in (b) represent the padded zeros. (From [DV16])

of a convolution. One application example is a convolutional autoencoder which is explained in further detail in Section 2.4.1. This operation is referred to as *transposed convolution*[10], which exchanges the forward and backward passes of a normal convolution. It is also called *fractionally strided convolution*, because it can be emulated with a direct convolution using a zero-spaced input [DV16, p. 19]. Such an implementation is less efficient, but it supports the intuition of how the resulting output shape looks like. Figure 2.7 shows an example of a transposed convolution.

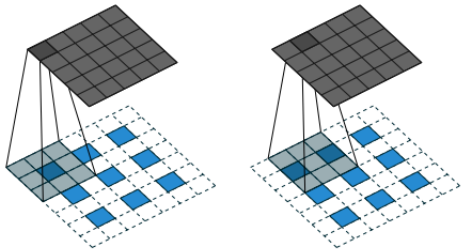


Figure 2.7: Transposition of convolving an 6×6 input using a 3×3 kernel using $p = 1$ and $s = 2$. This is equivalent to performing a convolution using a zero-spaced 3×3 input with $p = 1$ and $s = 1$. (From [DV16])

2.2.4 Advantages

The three central design ideas are emphasized here to sum up the benefits of a convolutional network. It refers to *sparse connections*, *parameter sharing* and *equivariance to*

[10]Mistakenly, the transposed convolution is often called *deconvolution*. But because it is not actually performing the reverse effect of a convolution, which is meant by the mathematical term of a deconvolution, it is strongly discouraged to name it so. An alternative name that is also often used is *upconvolution*.

translation [GBC16, p. 336ff.]. The detailed advantages of these concepts are described in the following sections.

Sparse Connections

The kernel size used in a convolution is smaller than the input. Consequently, less parameters have to be stored, as well as it can take advantage of local relationships present in the data. This also leads to a higher training efficiency and a radical reduction of memory requirements.

Parameter Sharing

To handle all regions of the input data in the same manner, the parameters are *reused* at every location as well. This is implemented by making use of only a single kernel which holds all learnable parameters. Additionally, this *weights sharing* decreases the number of parameters even further. To that end, Figure 2.8 compares the connection pattern and the sharing of model parameters of fully-connected layers against the convolutional case.

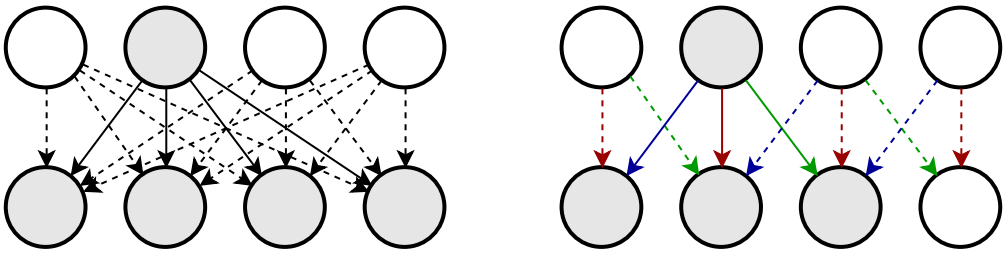


Figure 2.8: Comparison of the connection pattern and usage of parameters between (a) fully-connected layers and (b) convolutional layers. The use of the same color for interconnections denotes the sharing of parameters.

Equivariance to Translation

The sharing of parameters leads to the third advantage. Because the kernel and its parameters are reused at every position, the model learns the same representations at every position [GBC16, p. 339]. For example, if an input image is translated by a fixed number of pixels, the network would handle it in the exact same way.

Unfortunately, convolutions are not tolerant to other transformations from the ground up. But to counteract these, other techniques exist such as a subsequent pooling stage to enable a slight rotation invariance, as already introduced before.

2.2.5 Fully-Convolutional Networks

The complete use of convolutional layers implicates a fourth advantage over fully-connected layers. Since the kernel size in each layer is independent regarding its input, the overall network could be fed with data of different dimension. In contrary, this is not possible anymore as soon as a single FC-layer is used at inference because its fixed-sized weight matrix has to be applied to the entire input, not just to a local region. Nevertheless, this does not imply that no fully-connected layer can be used at all when training the network. Depending on the architecture, FC-layers can be used in those parts of the network that are only used during training, but not when a prediction is performed while inference. This advantage is taken into account for example when a *deep convolutional generative adversarial network* (DCGAN)[11] is trained, whose discriminator network is only used in training mode.

Regarding the problem of frame prediction that has to be solved within this thesis, it has to deal with a huge amount of data in every training iteration. Therefore, the advantages from this *fully-convolutional network* (FCN) approach are taken into account and design the architecture in such a way that it is possible to train the neural network model on small image patches only. Afterwards, it is theoretically able to perform frame prediction on the whole image given a sequence of frames.

2.3 Recurrent Neural Networks

All previously presented network architectures suffer from one missing characteristic: their memory is kind of static and predictions are mostly based on the current inputs only. Consequently, they are hard to be applied on problems where data reveals some sequential or temporal properties. Two examples are handwriting recognition, were the understanding of previous words is required to deduce the current word's context. Also in our case, the knowledge of the past image frames is fundamental to be able to predict the future frames that naturally match to the given previous sequence. A framework that addresses this issue is a *recurrent neural network* (RNN). In this section, an overview is given about their structure and formal description, as well as a succession model is presented that addresses its fundamental problems. It is to add that the whole section is mainly inspired by the great explanations in [Ola15].

[11] Novel network training strategy for generative networks. A generator network G competes against a second discriminator network D in an alternating fashion. Further details in [Goo+14].

2.3.1 Basics

RNNs are a special class of neural networks that allow its models to form a directed cyclic graph. Thereby, they are able to hold a hidden state that represents the sequential dynamics of the past. Given an input sequence $\mathbf{X} = (\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(\tau)})$, the τ^{th} recurrent building block gets $\mathbf{x}^{(\tau)}$ as its input of this sequence, as well as the hidden state $\mathbf{h}^{(\tau-1)}$ of the previous one. These building blocks are typically referred to as a *cell*. Because the first cell has no predecessor, its hidden state input $\mathbf{h}^{(0)}$ is usually manually fed with an zero-initialized state vector. Formally, an RNN can be described as follows:

$$\begin{aligned} \mathbf{h}^{(\tau)} &= \phi(\mathbf{W}_h \mathbf{h}^{(\tau-1)} + \mathbf{W}_x \mathbf{x}^{(\tau)} + \mathbf{b}_1) \\ \mathbf{y}^{(\tau)} &= \mathbf{W}_o \mathbf{h}^{(\tau)} + \mathbf{b}_2 \end{aligned} \tag{2.12}$$

where $\mathbf{W}_h \in \mathbb{R}^{d_h \times d_h}$ are the weights of the hidden-to-hidden transitions, $\mathbf{W}_x \in \mathbb{R}^{d_x \times d_h}$ the weights of the input-to-hidden connections, $\mathbf{W}_o \in \mathbb{R}^{d_h \times d_x}$ the weights of the hidden-to-output transitions and $\mathbf{b}_1, \mathbf{h}^{(0)} \in \mathbb{R}^{d_h}$ as well as $\mathbf{b}_2, \mathbf{x}^{(\tau)}, \mathbf{y}^{(\tau)} \in \mathbb{R}^{d_x}$ the biases, initial state, input and output respectively [Coo+15, p. 2], [GBC16, p. 381]. The activation function $\phi(\mathbf{z})$ is usually chosen to be a hyperbolic tangent.

Like convolutional networks, RNNs take advantage of sharing parameters over different parts of the model [GBC16, p. 374]. But in this case, model parameters are shared over the temporal domain. This allows the network to generalize specific properties across the whole input sequence. Consequently, a model can extract patterns that can occur at any or even multiple positions within the sequence of data.

Structure

For a better understanding of how recurrent networks work, it is helpful to take a closer look on its graphical model that was formally described by equation 2.12. As it can be seen in Figure 2.9a, the hidden state transition can be compactly visualized using a loop. These loops represent the influence of the past values with respect to the current value. However, to have a representation that is analogous to the already shown model, it is possible to unroll the loop in order to convert it back to a valid DAG. The unrolled recurrent network is depicted in Figure 2.9b.

In addition, the framework for recurrent models is very flexible as well. Depending on the implementation, it allows to process either a fixed or even a dynamic number of inputs. This property extends to the number of outputs as well. In contrary, convolutional or artificial neural networks require to define the input and output size at graph construction time. Further, they have to process all data in one chunk and do not allow to handle only single elements of the sequence one after the other. Some example input-output modes of recurrent networks are visualized in Figure 2.10.

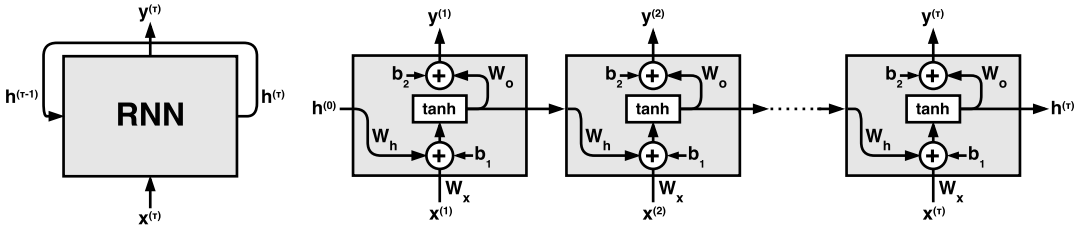


Figure 2.9: Structure of recurrent network cells. The compact cyclic graph model in (a) can be unrolled to receive the model (b) that represents the network in form of an acyclic graph. (Based on [Ola15])

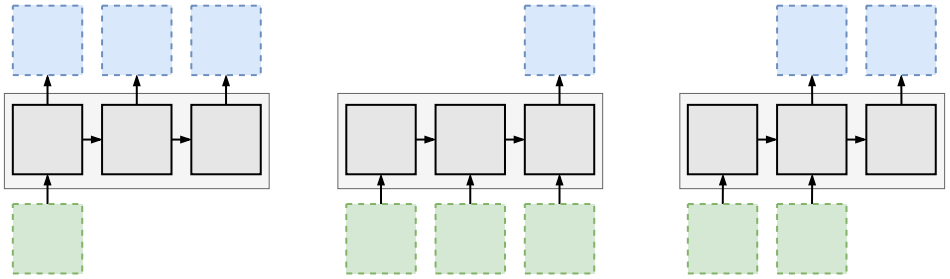
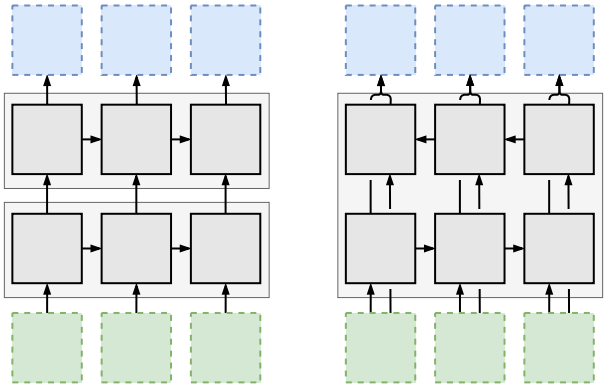


Figure 2.10: Visualization of different recurrent network input-output modes: (a) one-to-many, (b) many-to-one, (c) many-to-many. Green squares denote the inputs, gray squares the recurrent cells and all outputs are colored in blue. Input and output squares can be understood as either further neural networks or the direct input and output. (Based on [Kar15])

Besides the flexibility in the number of inputs and outputs, the recurrent components of the model is very adaptable as well. In common with the use of multiple neural network layers in order to achieve deeper and therefore more powerful models, several recurrent network layers can also be stacked on top of each other to enable a higher temporal depth. Various practical experiments, such as in [Ng+16], [Shi+15] or [SMS15], have shown that *multi-layer RNNs* are able to deliver better results compared to corresponding single-layer variants. Figure 2.11a shows an example of a two-layer recurrent network. As a downside, the increase in the number of recurrent layers usually has a tremendous effect on the memory requirements and the network training time.



(a) (b)

Figure 2.11: Examples of deeper recurrent network architectures. The figure in (a) shows a two-layer RNN, while (b) shows a single-layer BRNN. Both bidirectional outputs per cell of the latter model are combined and could be further processed by subsequent neural networks. (Based on [Kar15])

Additionally, the input sequence could be processed in both directions. This can be enabled by combining an RNN that starts at the beginning of the sequence with a second RNN starting from the very end backwards through time. In consequence, these so called *bidirectional recurrent neural networks* (BRNN) are also able to have access to its future input information [GBC16, p. 394ff.]. It is to add that the recurrent weights are not shared in both directions. An example of such a bidirectional model is depicted in Figure 2.11b. The same idea could also be extended to a sequential processing of data in two dimensions, which is known as a *grid RNN*.

Backpropagation Through Time

Analyzing the RNN structure might raise the question of how this effects the training procedure, because the gradient flows through recurrent cells multiple times while backpropagating the error. It is important to see that recurrent networks are still feed-forward networks with the extension to reuse the same weights expanded in time.

Consequently, the error still has to be propagated backwards starting from the last time step τ like in standard backpropagation. Depending on the length of the sequence and the computational resources, it can proceed until the very beginning, or truncate the view of interest by stopping at a given limit. Additionally, given the shared weights $\mathbf{w}$ of two timesteps τ and $\tau + 1$, the weight constraint $\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau+1)}$ must be met. Therefore, $\nabla\mathbf{w}^{(\tau)} = \nabla\mathbf{w}^{(\tau+1)}$ has to be fulfilled as well. This can be realized by computing the gradients for both time steps independently, but use the average of the following sum:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}^{(\tau)}} + \frac{\partial \mathcal{L}}{\partial \mathbf{w}^{(\tau+1)}} \tag{2.13}$$

when the final model parameter update is performed using the update rule [Hin13]. This principle can be extended to the total sequence length that is considered by the recurrent network and is referred to as *backpropagation through time* (BPTT).

Drawbacks

Keeping the previously explained weight constraints in mind, the recurrent network has to perform a lot of correlated updates of the same model parameters at once. This is actually bad for stochastic gradient descent, as it prefers uncorrelated parameters for the stability of the training. Especially when a sequence gets quite long, this can yield mathematical instability due to many multiplications using the same shared weights. On the one side, the gradients can grow exponentially and become infinite [HS97]. On the other side, the network could not learn anything because the gradients vanish. This issue is known as the *vanishing and exploding gradient problem*. The lack of learning long-term dependencies in recurrent networks has been identified in [Hoc91] and [BSF94]. Fortunately, other RNN variants exist that can deal with long-term dependencies. The currently most prominent version is introduced in the following section.

2.3.2 Long Short-Term Memory

Initially introduced by Hochreiter and Schmidhuber, the *long short-term memory* (LSTM) became kind of the default recurrent network implementation as it is capable to deal with long range dependencies. Over the years, it has been revised by a couple of follow-up studies [GS00] [Gra+06] and is used in many practical applications today.

The central advancement of LSTMs over traditional recurrent networks is the so called *memory cell state* $\mathbf{C}^{(\tau)}$. While a simple RNN cell overrides its state at each time step, the LSTM's memory cell update exhibits only minor linear interaction, so that information could flow through very easily. Moreover, this simplifies the gradient flow backwards through time [Coo+15]. It follows that an LSTM cell inverts the core issue it tries to

solve. Instead of learning to remember things, it is actually trained to learn what can be forgotten. In this way, keeping information over a longer period of time became its default setting.

To regulate the update of the internal memory state, the LSTM introduces the use of an attentive gating mechanism. At each time step, it is controlled by three trainable gates in order to accumulate or remove content from its state. Firstly, the *input gate* determines the flow of information from the current input $\mathbf{x}^{(\tau)}$. Secondly, the *forgot gate* regulates to which extend the information from the past cell state $\mathbf{C}^{(\tau-1)}$ is kept. And lastly, the LSTM output state $\mathbf{h}^{(\tau)}$ is specified by the filtered cell state using the *output gate*. This mechanism allows constant error flow and enables to build a long-term understanding even when applied to long sequences. All this can be formalized as:

$$\begin{aligned}
\begin{pmatrix} \tilde{\mathbf{f}}^{(\tau)} \\ \tilde{\mathbf{i}}^{(\tau)} \\ \tilde{\mathbf{o}}^{(\tau)} \\ \tilde{\mathbf{c}}^{(\tau)} \end{pmatrix} &= \mathbf{W}_h \, \mathbf{h}^{(\tau-1)} + \mathbf{W}_x \, \mathbf{x}^{(\tau)} + \mathbf{b} \\
\hat{\mathbf{c}}^{(\tau)} &= tanh(\tilde{\mathbf{c}}^{(\tau)}) \\
\mathbf{C}^{(\tau)} &= \sigma(\tilde{\mathbf{f}}^{(\tau)}) \odot \mathbf{C}^{(\tau-1)} + \sigma(\tilde{\mathbf{i}}^{(\tau)}) \odot \hat{\mathbf{c}}^{(\tau)} \\
\mathbf{h}^{(\tau)} &= \sigma(\tilde{\mathbf{o}}^{(\tau)}) \odot tanh(\mathbf{C}^{(\tau)})
\end{aligned} \tag{2.14}$$

where $\mathbf{W}_h \in \mathbb{R}^{d_h \times 4d_h}$ are the shared weights for the hidden to hidden transitions at time step τ, $\mathbf{W}_x \in \mathbb{R}^{d_x \times 4d_h}$ the shared weights for the input to hidden connections, $\mathbf{b} \in \mathbb{R}^{4d_h}$ the biases, and $\mathbf{C}^{(0)}, \mathbf{h}^{(0)} \in \mathbb{R}^{4d_h}$ the initial states of the memory cell and the hidden state, respectively [Coo+15]. The last dimension of the weight matrices and biases are multiples of four, because the matrices are concatenations of weights for all three gates plus the new candidate cell state $\hat{\mathbf{c}}^{(\tau)}$ for computational efficiency. The input, forget and output gates are labeled as $\mathbf{i}, \mathbf{f}$ and $\mathbf{o}$. Furthermore, the operator $\odot$ denotes the Hadamard product[12] and a tilde indicates the term before the corresponding activation is applied, so that for example $\mathbf{f}^{(\tau)} = \sigma(\tilde{\mathbf{f}}^{(\tau)})$.

Structure

As depicted in Figure 2.12, the internal structure of an LSTM cell looks way more complex compared to the simple RNN of Figure 2.9b. Nevertheless, the role of each single building block has actually a quite simple interpretation.

As stated before, the fundamental enhancement of LSTMs is the cell state $\mathbf{C}^{(\tau)}$, denoted as a bold line in the graphic. From its previous cell state to the next, there are just

[12] The Hadamard product defines the entry-wise product of two matrices with the same dimension.

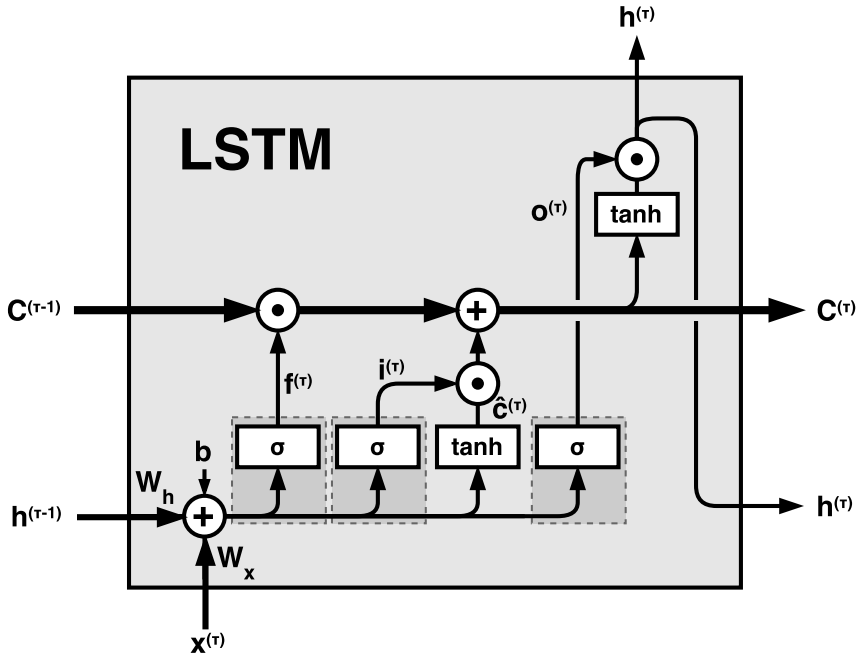


Figure 2.12: Structure of a LSTM cell. The flow of the cell state is highlighted in bold. Gated unites are emphasized with a dotted box. (Based on [Ola15])

two interaction points where its content gets manipulated. First, using a pointwise multiplication with the output of the forget gate $\mathbf{f}^{(\tau)}$, a specific part of the previous state can be partly or fully deleted. Afterwards, the new candidate cell state $\hat{\mathbf{c}}^{(\tau)}$ is added to it, which is only regulated by the input gate $\mathbf{i}^{(\tau)}$. The third gate $\mathbf{o}^{(\tau)}$ has no effect on the cell state itself and governs the hidden output only. The interpretation of how each single gated unit works is the following: each gate itself is a fully-connected neural network layer, that receives the weighted sums of the input $\mathbf{x}^{(\tau)}$ and the previous hidden state $\mathbf{h}^{(\tau-1)}$ as its input. That is the reason why they are also often referred to as *FC-LSTM*. Additionlly, as it can be seen in Figure 2.12, every gate uses a sigmoid as its activation function that is followed by an Hadamard product. Consequently, the output of any gate is in range $[0, 1] \in \mathbb{R}^{d_h \times 4d_h}$, where a value of one means let the whole information flow through, while zero intends to forget everything.

Variants

The long short-term memory belongs to the family of *gated RNNs* [GBC16, p. 411]. Since its invention, a couple of related implementations have been introduced. One example extends the LSTM by having so called *peephole connections*, proposed in [GS00]. These additional connections have the purpose that each gating unit has a direct access to the previous memory cell state $\mathbf{C}^{(\tau-1)}$. The motivation for this is to allow the units

to learn when to open or close their gates based on the cell state [Gre+15], in order to learn more precise timings.

Another example is the *gated recurrent unit* (GRU) [Chu+14], which also regulates the flow of information comparable to an LSTM, but without having a separate memory cell. Additionally, its gates for input and output are merged to a single *update gate*, which leads to a similar performance but with a lower memory footprint [Bal+16]. The forgot gate in context of a GRU is typically called *reset gate*.

A further, novel variant of LSTMs will be presented in Chapter 4, which is inspired by the strengths of convolutional neural networks. Aside from that, it will formalize the use of peephole connections as well.

2.4 Encoder-Decoder Networks

In order to learn useful representations, the generic, informative content has to be extract from the given data. But in many cases, we have to deal with a tradeoff between obtaining valuable properties and preserving as much information as possible regarding the inputs. Additionally, intense overfitting effects have to be handled when these representations are learned in a supervised fashion, because an acceptable amount of training data is often not on-hand for this purpose [GBC16, p. 527]. Therefore, this section focuses on concepts that are able to end up with good representations on the basis of unlabeled data in an *unsupervised learning* process.

2.4.1 Autoencoders

An *autoencoder* is a commonly known neural network model that is able to reconstruct its own input. It consists of two components that are usually arranged in a mirrored style. First, an *encoder* $f(\mathbf{x}) = \mathbf{z}$ that maps any given input $\mathbf{x} \in \mathbb{R}^{d_x}$ to its internal representation $\mathbf{z} \in \mathbb{R}^{d_z}$. Second, a *decoder* function $g(\mathbf{z}) = \mathbf{x}'$ that is able to map the representation back to its input. Due to the analogy to encoding and decoding, the learned representation is also often referred to as *code*. In addition, the input and output layer of such networks have the same number of nodes. A simplified autoencoder model is visualized in Figure 2.13.

The encoder and decoder components can be of any kind. In the simplest case, a neural network with only one single hidden layer could be used. However, it has been shown that deeper networks are able to yield better representations compared to shallow variants [HS06]. Furthermore, the use of convolutional layers is usually preferred for image processing tasks, referred to as *convolutional autoencoders*.

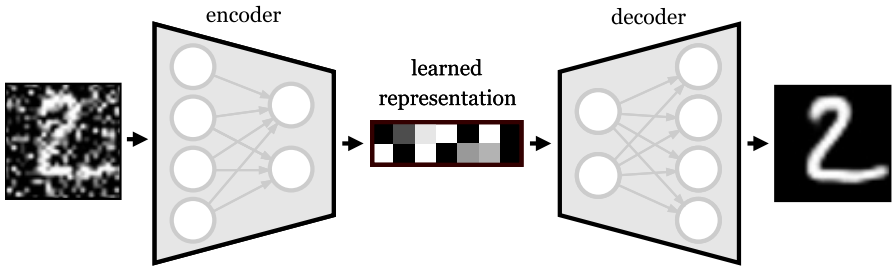


Figure 2.13: Simplified structure of a denoising autoencoder in an undercomplete setting to reconstruct images of handwritten numbers from the MNIST dataset. The encoder network efficiently encodes the input data in order to come up with a learned representation. This representation can then be processed by the decoder network to reconstruct the noise-free image.

Reconstructing the input might sound trivial in first place. And honestly, the model itself that is able to learn the identity mapping $g(f(\mathbf{x})) = \mathbf{x}$ and especially its output is not very interesting, since the network would just perform a copy operation. Instead, we are more interested in some special properties of the latent variable z. These can be provoked by restricting the encoder, decoder or input with some additional constraints so that the model is not able to perform a perfect copy. Moreover, the dimensionality of the internal representation could be varied in order to force the model that it learns to prioritize which fraction of the input is worth to have in contemplation. Hence, it learns valuable properties regarding the data, which might even be useful to cope with other related tasks. Autoencoder models received increasing attention in recent years, especially in relation with *generative models* [GBC16, p. 502].

Undercomplete Autoencoders

On the one side, in the typical *undercomplete* case where the dimensionality of the representation is smaller than the input and output, so that $d_z < d_x$, a model can be trained to reconstruct the original data [GBC16, p. 503]. As an application example, this can yield in an image compression algorithm, where the encoder compresses an input image to another representation of lower size. Afterwards, the decoder network can be used to recover the image data. As a result, depending on how much lower the size of the representation and the dissimilarity of the reconstruction compared to its original image is, the better is our learned representation. It then would have learned to preserve the image information in a more compact way.

Regularized Autoencoders

On the other side, there are also use cases for *overcomplete* models where the representation's dimensionality could exhibit an equal or even higher dimension than the input and output[13], so that $d_z \geq d_x$. But this requires to apply regularization techniques in order to learn useful features. The reason is that such a high model capacity can easily end up in overfitting [GBC16, p. 504f.]. One first possibility is to extend the loss function with an additional sparsity penalty, so that the *sparse autoencoder* network tries to minimize:

$$\mathcal{L}(\mathbf{x}, g(f(\mathbf{x}))) + \lambda \cdot \Omega(\mathbf{z}), \tag{2.15}$$

where λ controls the tradeoff between sparsity and reconstruction and regularizer term $\Omega(\mathbf{z})$ can be any valid loss function, such as ℓ_2. This sparsity constraint reduces the autoencoder's degrees of freedom and therefore prevents overfitting. To put it another way, the model is virtually undercomplete by having a compactly distributed representation instead of a lower dimensionality. Alternatively, sparsity could also be achieved by only using the k most meaningful activations in a *k-sparse autoencoder* by zeroing out weak hidden units manually [MF14] or by using rectifier units in the last encoding layer [GBB11a].

A second regularization strategy is to modify the reconstruction term of the loss function itself by adding random noise to the input. These networks are called *denoising autoencoders* and have to learn how to correct back the added noise for the purpose of reconstructing the original noise-free image [GBC16, p. 507]. Therefore, the network has to learn a deeper understanding of the input data instead of simply copying its content. As a consequence, it has to minimize:

$$\mathcal{L}(\mathbf{x}, g(f(\tilde{\mathbf{x}}))), \tag{2.16}$$

where $\tilde{x}$ is the randomly corrupted version of the ground truth input x.

2.4.2 Recurrent Encoder-Decoder Models

The general idea of the encoder-decoder framework can also be applied to recurrent networks. Thus, it can build a complex representation by incorporating a whole sequence of inputs. To our knowledge, the *recurrent encoder-decoder* model was firstly introduced independently from each other in [Cho+14] and [SVL14]. Further, it was

[13] It is to be added that the presented regularization strategies can also be used for undercomplete autoencoders.

used in [SMS15][14] for the first time in context of image processing. But since then, it is applied in numerous subsequent works [Pen+16], [LZR16]. Figure 2.14 shows an example of such a model in context of a recurrent autoencoder.

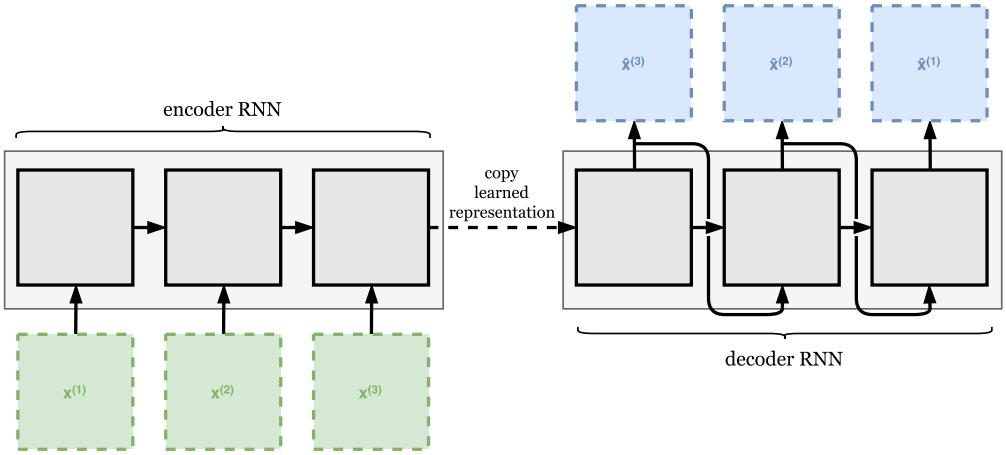


Figure 2.14: Basic structure of a conditional recurrent autoencoder. The inputs (green) are processed by the encoder RNN to learn the representation of the data in sequence. Then, the decoder RNN takes over to infer the reconstructions (blue) of the inputs in reverse order. (Based on [SMS15])

The encoder RNN builds the representation based on all inputs of the sequence and therefore takes advantage of its temporal or ordinal structure. Afterwards, the learned representation is used to initialize the hidden state $\mathbf{h}_{\text{dec}}^{(0)}$ of the decoder, in opposite to the otherwise customary zero initialization. The decoder RNN takes then over and outputs the prediction of the target sequence. Obviously, both the encoder and decoder recurrent networks could be extended to consist of multiple layers or bidirectional connections as well.

There are at least two possibilities of how the decoder can be designed. Firstly, an unconditional setting where each recurrent cell does *not* receive the previous output as its input. Secondly, a conditional setting in which this additional connection from the cell's output to the input of the next element is present. Both have its reasons for and against. On the one side, it can be argued that conditioning on the previous output enables to learn more dynamics in the target sequence distribution. This conditioning might not be required in an autoencoder setting, where there is only exactly one ground truth target sequence. But when this architecture is applied to continue a series of frames in the future, it could be crucial to have access to the previous output. As an example, consider the prediction of a series of frames from a simple video game,

[14] The authors call this recurrent framework the *LSTM autoencoder model* within their publication. But due to the fact that it can be generalized to another recurrent cell implementations, as well as it could be applied in a non-autoencoder fashion, the more general name *recurrent encoder-decoder* model is used here as well. This name is also used in several follow-up works.

where the player could walk either left or right in a specific situation. To properly predict the next outputs afterwards, it is fundamental to know which decision this particular recurrent cell has made. Otherwise, the RNN might end up in averaging over all possible outcomes. On the other side, it could also be argued that conditioning on the previous frame would end up in preventing the network to look deeper inside the encoder network [SMS15].

2.5 Batch Normalization

Training a deep neural network model is said to be very hard. One reason for this is that every layer has not just to learn the overall representation directly from the beginning of the training, but also has to master the continuous change in distribution of its inputs, given all preceding layers. As an example, consider one layer in the middle of a deep network. At training time, its adjustments to the model parameters depend on its inputs, which is equivalent to the output of the previous layer. But the fact that the previous layer learns and modifies its weights and biases, it follows that the input to the following layer can change over time significantly. Especially at the very beginning of the training, due to the common use of random initialization. This ongoing change in the feature's distribution during training is known as the *internal covariance shift* [Coo+15].

One modern practice that tries to overcome this issue is called *batch normalization* [IS15]. It performs a normalization on the inputs to a layer and transforms it to have a mean of zero and a standard deviation of one. Consequently, it compensates the covariate shift between two layers of the network. Its mathematical formulation is:

$$\mathrm{BN}(\mathbf{x}; \gamma, \beta) = \beta + \gamma \frac{\mathbf{x} - \mathbb{E}(\mathbf{x})}{\sqrt{Var(\mathbf{x}) + \epsilon}}, \tag{2.17}$$

where $\mathbf{x} \in \mathbb{R}^d$ is the output of the previous layer to be normalized, $\gamma \in \mathbb{R}^d$ and $\beta \in \mathbb{R}^d$ the learned shift and scale of the distribution, as well as $\epsilon \in \mathbb{R}$ that serves as a regularization parameter to avoid dividing by zero [Coo+15].

When batch normalization is applied, two different modes during training and inference have to be considered. On the one hand at training time, batch normalization takes advantage of two simplifications, caused by the fact that full whitening of the inputs is very expensive. First, each feature dimension is normalized independently. Secondly, the statistics of $\mathbb{E}(\mathbf{x})$ and $Var(\mathbf{x})$ are estimated using the sample mean and sample variance based on each mini-batch of stochastic gradient training. On the other hand, the statistics have to be assessed using the entire training data during inference. One way to do so is to track the moving averaged mean and variance parameters of each

batch-estimate while training the model, and finally use these averaged values when a prediction is performed. This is the way how batch normalization is used throughout this work.

The advantages of batch normalization are versatile:

- Compensation against internal covariate shift to achieve a stable distribution of activation values.
- Separate normalization of individual feature dimensions, so that features are not decorrelated.
- Speed-up training, because it allows to use higher learning rates.
- Practice has shown that even a higher accuracy can be achieved compared to non-batch-normalized versions of a network model [IS15, p. 7].

2.6 Image Similarity Assessment

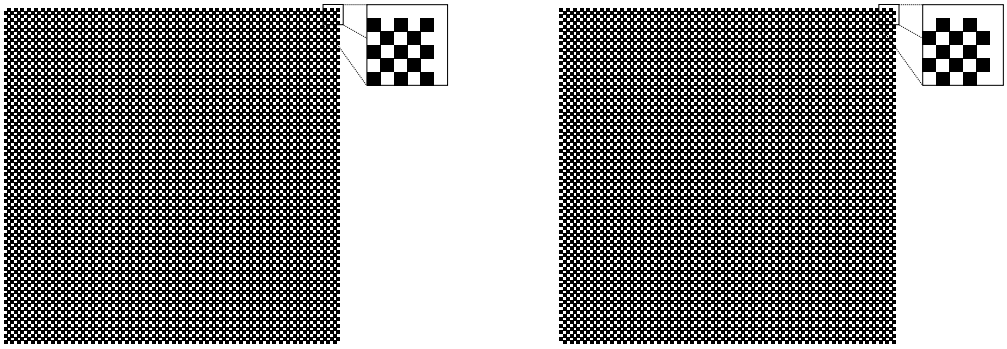
The practical use of deep learning in context of image generation tasks received a tremendous increase in recent years. Examples include denoising autoencoders (see Section 2.4.1) for image reconstruction, semantic image completion [Yeh+16], the determination of optical flow [Fis+15], [WGH15] or future frame prediction of a given image sequence. The last mentioned field of application is in scope of this thesis.

A neural networks main driver in respect to learning a good representation is the loss layer. Unfortunately, it found only little attention in most existing studies when applied in image processing tasks, at least until this year [Zha+16]. Therefore, this section provides an overview of the main challenges of image similarity assessment and introduces some approved methods, as well as novel approaches when training neural networks in context of pictures.

2.6.1 Naive Approach

The de facto standard loss function to compare a generated image with its ground truth target has been the *pixel-wise squared error* (or ℓ_2) that was already mentioned in equation 2.3. It is easy to apply and usually provided by any neural network framework out-of-the-box. But as a downside, it suffers from some well-known limitations such as poor correlation with the human sense of perceptual quality [Zha+16]. The reason for this are twofold. First, ℓ_2 makes the assumption of a Gaussian noise model, which has its downsides when being applied to multimodal distributions. Second, it strongly penalizes outliers independently from local characteristics of the image, such as contrast

or luminance. In contrary, the visual system of humans perceives noise in a different way.



(a) (b)

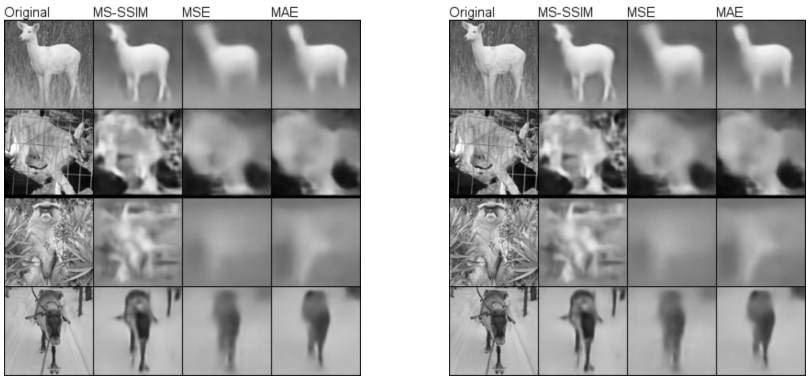
Figure 2.15: Example to demonstrate the weakness of ℓ_2 and MSE error in context of perceptual image similarity.

As an extreme but simple example, consider two images containing a chess field of black and white pixels as shown in Figure 2.15. From an aerial perspective, both images look completely identical, because it is perceived as a gray surface. Additionally, even when we look at it from a closer distance, we would still evaluate them as quite similar. This is for the very reason that both images provide the same colors, structure, sharpness or contrast. Unfortunately, assessing both images using the given squared error function results in the maximum possible difference. The reason for this is because it considers the squared differences of related pixels only. Even though this specific weakness is not solved by using the ℓ_1 loss, several studies have shown that the usage of the absolute error function slightly reduces the blur effect on edges in image generation processes of natural images [Zha+16] [MCL16]. Some example image reconstructions comparing the use of different loss functions are show in Figure 2.16.

2.6.2 Perceptual Quality Metrics

The example in the previous section has shown that the consideration of the human visual perception is very important when assessing image similarity. With this in mind, a couple of metrics were developed to evaluate image quality differences between two images. These metrics can be used either for a quantitative evaluation of generated results, or even as a loss function by doing some minor modifications to fulfill the required properties. Beside neural network training, these metrics were originally invented to measure the quality of image compression codecs such as JPEG[15].

[15]JPEG: stands for *Joint Photographic Experts Group* and is the name of the committee that created this compression standard for digital pictures.



(a) (b)

Figure 2.16: Comparison of reconstructions with different loss functions using images from the STL-10 dataset: (a) where results using a perceptual motivated loss function were ranked first by human judgement, (b) where MAE or MSE were ranked best. (From [Rid+16])

In the following sections, $\mathbf{x}$ will denote the ground truth image and $\mathbf{y}$ its generated reconstruction to compare with.

Peak Signal-to-Noise Ratio

A first metric to assess the similarity of generated images is the *peak signal-to-noise ratio* (PSNR). It describes the ratio between the maximum possible image intensity and the corrupting noise that affects precision of the reconstruction. Its value is expressed in a logarithmic decibel scale, where a higher value indicates better quality. In terms of human perception, it is a rough approximation to evaluate reconstruction quality, because its denominator is still based on MSE. It is computed as follows:

$$\text{PSNR}(\mathbf{x}, \mathbf{y}) = 10 \cdot \log_{10} \left(\frac{\mathbf{y}_{max}^2}{\frac{1}{w\,h} \sum_{c=1}^{w} \sum_{r=1}^{h} (\mathbf{x}_{c,r} - \mathbf{y}_{c,r})^2} \right), \tag{2.18}$$

where $\mathbf{y}_{max}$ is the maximum *possible* intensity of any given image with size $w \times h$.

Structual Similarity

For predicting the perceived image quality, the *structural similarity* (SSIM) index invented in [Wan+04] can be used as another assessment criteria. This full reference metric[16] is an improvement over PSNR as it is based on several assumptions of the

[16] Full reference metric (FR) is a quality term which means that the evaluation is based on every pixel of the entire ground truth image as a reference.

human vision system. Therefore, it assesses both images based on luminance $l(\mathbf{x}, \mathbf{y})$, contrast $c(\mathbf{x}, \mathbf{y})$ and structural similarity $s(\mathbf{x}, \mathbf{y})$ [WSB03]. These components are defined as follows:

$$\begin{aligned} l(\mathbf{x}, \mathbf{y}) &= \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \\ c(\mathbf{x}, \mathbf{y}) &= \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2} \\ s(\mathbf{x}, \mathbf{y}) &= \frac{2\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}, \end{aligned} \tag{2.19}$$

where μ_x, σ_x and σ_{xy} is the mean, standard deviation and covariance of x and y, respectively. Further, $C_1 = (K_1 \cdot L)^2$, $C_2 = (K_2 \cdot L)^2$ and $C_3 = \frac{C_2}{2}$ are small constants for numerical stability, $K_1 = 0.01$ and $K_2 = 0.03$ by default and $L = 255$ is the typical dynamic range of pixel-values for 8-bit *gray-scale* images. These terms can be combined to define the SSIM index given by:

$$\text{SSIM}(\mathbf{x}, \mathbf{y}) = [l(\mathbf{x}, \mathbf{y})]^\alpha \cdot [c(\mathbf{x}, \mathbf{y})]^\beta \cdot [s(\mathbf{x}, \mathbf{y})]^\gamma, \tag{2.20}$$

where α, β and γ parameterize the relative importance of all three components, typically $\alpha = \beta = \gamma = 1$. The terms for contrast and structure can be further simplified to $cs(\mathbf{x}, \mathbf{y})$ [Zha+16, p. 5], resulting in:

$$\begin{aligned} \text{SSIM}(\mathbf{x}, \mathbf{y}) &= \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \cdot \frac{2\sigma_{xy} + C_2}{\sigma_x^2 + \sigma_y^2 + C_2} \\ &= l(\mathbf{x}, \mathbf{y}) \cdot cs(\mathbf{x}, \mathbf{y}). \end{aligned} \tag{2.21}$$

The SSIM index can be computed using a sliding window approach [WB02]. Therefore, a square kernel[17] of size 11×11 and *valid* padding is used that moves over the whole image, pixel by pixel. The index is then calculated in every local region and finally averaged to receive the overall image quality index for evaluation. The metric value is in range $SSIM(\mathbf{x}, \mathbf{y}) \in [0, 1]$, where a higher value indicates more similarity.

Multi-Scale Structural Similarity

A further study has shown that the viewing conditions can have a tremendous influence on the perceived image similarity [WSB03]. Therefore, the SSIM index is extended

[17] The initial paper states to use an 8×8 window. But many open source libraries, and even the MATLAB implementation of the paper's author itself use a kernel size of 11×11. In addition, some works list their evaluation results using several different kernel sizes.

to incorporate the image on M different scales, where $M = 1$ indicates the full-size image that is iteratively downsampled by a factor of two. This metric is known as the *multi-scale structural similarity* (MS-SSIM) index for images. It is given by:

$$\text{MS-SSIM}(\mathbf{x}, \mathbf{y}) = [l_M(\mathbf{x}, \mathbf{y})]^{\alpha_M} \cdot \prod_{j=1}^{M} [c_j(\mathbf{x}, \mathbf{y})]^{\beta_j} \cdot [s_j(\mathbf{x}, \mathbf{y})]^{\gamma_j}, \tag{2.22}$$

where the exponents α_M, β_j and γ_j parameterize the relative importance of each component. The luminance difference $l_M(\mathbf{x}, \mathbf{y})$ is only computed for the smallest image size at scale M. As a simple standard selection for the exponents, one can use $\alpha_M = \beta_j = \gamma_j$ and $\sum_{j=1}^{M} \gamma_j = 1$. But by performing an empirical study in [WSB03], the authors propose to use $\beta_1 = \gamma_1 = 0.0448$, $\beta_2 = \gamma_2 = 0.2856$, $\beta_3 = \gamma_3 = 0.3001$, $\beta_4 = \gamma_4 = 0.2363$ and $\alpha_5 = \beta_5 = \gamma_5 = 0.1333$ by incorporating $M = 5$ scales. As a downside, evaluating multiple scales can be computational expensive. Additionally, the selected window size has to be smaller than the image at scale M. Also, the image has to have an appropriate minimum size due to the iterative downsampling of the algorithm.

Sharpness Difference

To quantitatively evaluate the difference in sharpness between two images, the *sharpness difference* metric proposed in [MCL16] can be used. It is based on the formulation of PSNR (see eq. 2.18) with a modified denominator. Instead of using the squared error to quantify the pixel-wise intensity differences, it measures the difference of gradients between the ground truth and its reconstruction:

$$\text{SharpDiff}(\mathbf{x}, \mathbf{y}) = 10 \cdot \log_{10} \left(\frac{\mathbf{y}_{max}^2}{\frac{1}{w\,h} \sum_{c=2}^{w} \sum_{r=2}^{h} |(\nabla_{left}\,\mathbf{x} + \nabla_{top}\,\mathbf{x}) - (\nabla_{left}\,\mathbf{y} + \nabla_{top}\,\mathbf{y})|} \right), \tag{2.23}$$

where $\nabla_{left}\,\mathbf{x} = |\mathbf{x}_{c,r} - \mathbf{x}_{c-1,r}|$ and $\nabla_{top}\,\mathbf{x} = |\mathbf{x}_{c,r} - \mathbf{x}_{c,r-1}|$ are the gradient differences to the left and top pixel.

2.6.3 Perceptual Motivated Loss Functions

Due to the lacking consideration of human perceptional qualities like sharpness, contrast or structure of standard error functions, new forms of losses should be considered when a neural network is trained to solve image processing tasks.

Structural Loss

The differentiability of the SSIM index makes it a well suited candidate to be used in neural network training. However, due to the fact that $\text{SSIM}(\mathbf{x}, \mathbf{x}) = 1$, it does not fulfill all required properties of a loss function. Fortunately, this can be rectified by exchanging the minimum and maximum value of the metric:

$$\mathcal{L}_{\text{ssim}}(\mathbf{x}, \mathbf{y}) = 1 - \text{SSIM}(\mathbf{x}, \mathbf{y}). \tag{2.24}$$

Of course, the same principle can be applied to end up with the multi-scale structural loss function:

$$\mathcal{L}_{\text{ms-ssim}}(\mathbf{x}, \mathbf{y}) = 1 - \text{MS-SSIM}(\mathbf{x}, \mathbf{y}). \tag{2.25}$$

Both error functions were evaluated in context of image generation super-resolution or JPEG artifact removal in [Rid+16] and [Zha+16]. The latter suggests to combine each of them with ℓ_1 to get the best of both worlds.

Gradient Difference Loss

Using the same criteria of the sharpness difference metric (see eq. 2.23), a strategy to further sharpen the image is to penalize the gradient differences in image space. This loss function is referred to as *gradient difference loss* (GDL) and was proposed in [MCL16]. Combined with another error function, it can serve as an additional bias to deliver sharper results. To be more specific, the authors suggest to combine it with an ℓ_1 loss function. The per-pixel GDL function to assess the ground truth image with its corresponding reconstruction is defined as follows:

$$\mathcal{L}_{\text{gdl}}(\mathbf{x}, \mathbf{y}) = \frac{1}{w\,h} \sum_{c=2}^{w} \sum_{r=2}^{h} \Big(|\nabla_{left}\,\mathbf{y} - \nabla_{left}\,\mathbf{x}|^{\alpha} + |\nabla_{top}\,\mathbf{x} - \nabla_{top}\,\mathbf{y}|^{\alpha} \Big), \tag{2.26}$$

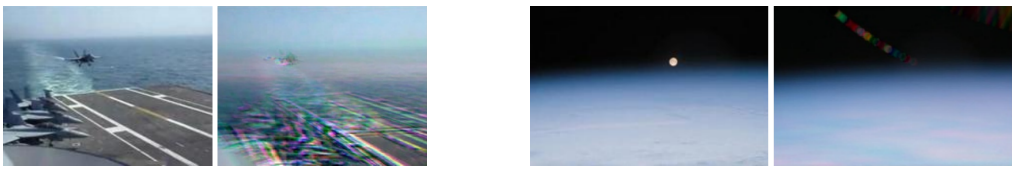
where $\alpha \in \mathbb{N}^{+}$ is a parmeter to adjust the exponent. Typically, $\alpha = 1$ is chosen when combined with ℓ_1 loss and $\alpha = 2$ in combination with ℓ_2. With training efficiency in mind, the function describes the simplest image gradient by only considering the intensity difference of the direct neighbors.

3 Related work

This chapter presents existing deep learning approaches that have addressed the issue of future frame prediction. These are grouped into three sections depending on the model implementation, namely neural networks, recurrent networks and adversarial networks. The strengths, weaknesses and design decisions of these models are briefly discussed together with a short analysis of the achieved outcomes. Further, it is highlighted how these approaches have influenced the architecture of our final model that is used throughout the evaluation in Chapter 6. Aside from that, their results form the baseline in the assessment of our model.

3.1 Neural Network Approach

First approaches to predict future frames in an image sequence were made in [Kar12] and its follow-up publication [Ver12]. Probably due to the lower computational power and the weak development of CNNs at that time, they tried to perform single frame prediction based on an artificial neural network. They performed several preprocessing steps in order to train such a model using image data. Firstly, the image data was partitioned by the R, G and B color channels into three parts. Next, the dimension of data was reduced from the order of 10^4 to 100 in each part using techniques like *principle component analysis* (PCA)[1]. The final training and inference was then accomplished on three separate neural networks of the same architecture for each color channel. After each prediction, the PCA process was inverted to obtain the initial dimensionality, as well as all three outputs were combined to obtain the final image.



(a) Fighter dataset (b) NASA dataset

Figure 3.1: Single frame predictions using an ANN model with two hidden layers. Left: ground truth target frame. Right: generated prediction. (From [Kar12])

[1]PCA: Technique to reduce the data dimensionality by mapping it into its eigenspace.

The loss during the training process is measured in terms of the MS-SSIM index in order to optimize the network to preserve the luminance, contrast and structure of the image. Since the data of the used Fighter and NASA datasets have a high image size, applying the multi-scale version of SSIM is a reasonable choice.

When we take a look at the presented prediction results in Figure 3.1, it can be seen that the network more or less averages over the input sequence. This effect is clearly visible in Figure 3.1b, where the movement of the moon from the top left towards the earth's horizon is predicted like being composed of previous moon positions. As a result, this simple architecture does not sufficiently capture the temporal correlations of the input data.

3.2 Recurrent Network Approaches

In order to leverage the sequential structure and temporal correlations of video data, several works performed frame prediction based on recurrent network models. The following models have inspired our final model architecture the most.

3.2.1 LSTM Encoder-Decoder-Predictor Model

A huge step forward was made when the recurrent encoder-decoder framework, earlier presented in Section 2.4.2, was applied in [SMS15] to perform unsupervised learning of video representations. The fact that the same operation should be applied at each time step to produce the next state was their key idea to use this framework in that context.

All in all, they preseted a LSTM autoencoder model that is trained to reconstruct an entire input sequence of about ten image frames, similar to Figure 2.14. This model was then slightly modified in a second step to predict the future sequence of frames. In a last step, both models were combined to a single model that contains only one encoder to learn the dynamics of the video, but two separate encoder networks. Initialized by a copy of the learned representation, one decoder tries to reconstruct the inputs backward in time, while the other decoder predicts the future frames forward in time. Consequently, the decoder has to come up with a representation that can be handled by both decoders. In this way, they tried to compensate the shortcomings of each model, such as the potential tendency of the reconstruction decoder to learn the trivial function, or to counteract that the future predictor considers the last frames of the input sequence only. This combined model delivers the best results and is shown in Figure 3.2.

Within their work, they also explored if the decoder should condition on the previously generated output or not, as earlier discussed in Section 2.4.2. The final choice has fallen on the conditioned variant because it delivered slightly sharper frame prediction results

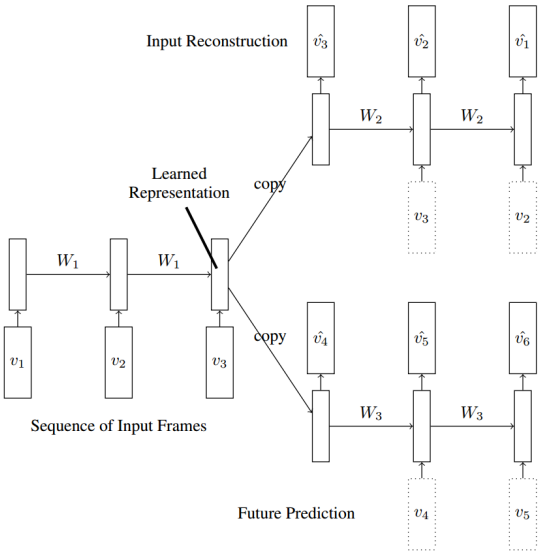


Figure 3.2: The composite LSTM autoencoder model. The top branch reconstructs the input sequence backwards in time, while the bottom branch performs frame predictions forward in time. (From [SMS15])

in the qualitative evaluation. Besides that, they also varied the number of recurrent layers with the clear result that deeper LSTMs yields best performance.

Another contribution of this work was the introduction of a simple dataset that can be generated on-the-fly in order to explore the architecture of the model, as well as the effects of hyperparameter changes. It uses handwritten numbers that bounce around in a short sequence of images. Since this dataset is used in several other subsequent works as well, it offers the ability to be used as a basic benchmark to compare the performance of various models. The dataset is presented later in Section 5.1. Sequences from this and another dataset were then used as input to the LSTM encoder to train the model. It must be highlight that they utilize the full image patch for this purpose. They have also mentioned to use convolutional percepts of the image sequence as inputs, but actually used this approach in the second part of their paper only, where the pre-trained encoder was transferred to improve the performance of supervised human action classification in videos.

The authors also pointed out that the choice of the loss function is fundamental with respect to the quality of results. Nevertheless, they decided to rely on standard error functions and kept the use of more advanced objective functions for further research. To be more precise, they trained their network using binary cross-entropy when being applied to Moving MNIST, and squared error for real world tests on UCF-101. Details about the latter dataset can be found in Section 5.3.

All in all, the strength of this model regarding future frame prediction is that it is able to infer a variable number of frames by taking the temporal correlations of the entire input sequence into account. But as a downside, the use of FC-LSTM cells with such high dimensional inputs implies a huge model complexity in the order of 10^8 in case of a two-layer LSTM with 2048 hidden units each. Consequently, such a model takes a very long time to learn useful patterns. Furthermore, it does not consider spatial properties of each single input due to the use of fully-connected state transitions.

3.2.2 Convolutional LSTM Encoding-Forecasting Model

The previously described model was extended in [Shi+15] with the general goal to develop a deep learning approach for precipitation nowcasting[2]. But in order to moderate the tremendous redundancy of spatial correlations in standard FC-LSTM cells, they invented a modified version of LSTM that features convolutional structure for both input-to-state and state-to-state transitions. More details regarding its formulation and internal structure follows in Section 4.1.1. To put it in a nutshell, they exchanged each matrix multiplication by a convolution operation whereby the internal states become three dimensional tensors[3] and preserve spatial information. These *convolutional LSTM* (ConvLSTM) cells are then used in the same decoder-encoder framework as before, like illustrated in Figure 3.3. At the bottom line, these cells are able to capture spatio-temporal properties of the data much better than FC-LSTM cells and have shown to outperform them with even containing way less model parameters.

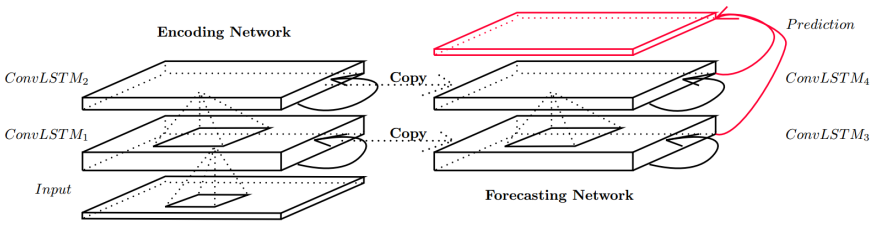


Figure 3.3: The ConvLSTM encoding-forecasting model that was used in the paper in context of frame prediction and precipitation nowcasting. (From [Shi+15])

Among other datasets, the authors trained this model on Moving MNIST in the course of their work and therefore it is another good candidate to compare our model with. The model was thereby fed with reshaped tensors of size $16 \times 16 \times 16$ by splitting the original frames using a 4×4 grid [Shi+15, p. 6]. The reason for this reshaping was not argued in the paper, despite the fact that this unnecessarily increases the spatial redundancies. But since it divides the size of the image by a factor of 16, one simple

[2]Precipitation nowcasting: short-term forecasting of rainfall.
[3]Tensor: multidimensional data array that flows through the computation graph. It's shape can change when passing any neural network layer.

reason might be to reduce the computational complexity because the depth is increased by the ConvLSTM's convolutions nevertheless. To generate the final prediction, the state of each ConvLSTM layer per time step τ is concatenated and fed into a 1×1 convolutional layer for the purpose of reducing their depth to match with the ground truth target [Shi+15, p. 4]. Also at this point, it was not explained why the concatenated hidden state of all layers is used to generate the prediction, instead of the more intuitive choice of only relying on the the final layer'w output.

To condense the three most important findings of their evaluations, it was shown that the kernel size of the state-to-state transitions has to be at least bigger than 1×1 to capture spatio-temporal motion patterns. The windows size of this kernel can be interpreted as the maximum motion that the model is able to detect from one time step to the next. The second outcome is that deeper models can produce better results even when each layer contains fewer parameters. And last but not least, as already stated earlier in this section, the use of convolutional LSTM cells instead of FC-LSTM cells enables to reach better performance with less training examples, requires less iterations to converge and is less likely to overfit.

3.2.3 Spatio-Temporal Video Autoencoder Model

A second work using ConvLSTM cells was published in [PHC16]. It is based on the fundamental idea that a video autoencoder should differ from spatial autoencoders by being able to encode the significant differences instead of memorizing the entire video sequence. With such an encoding at hand, neural networks should be able to learn the generation process of the future for any given frame. Compared to both previous models, this one does not rely on the standard recurrent decoder-encoder framework. Instead, it uses convolutional layers followed by a ConvLSTM encoder to produce a dense transformation map as its learned representation. Afterwards, a semi-detached optical flow module serves as a temporal decoder on the basis of the learned transformation map. The estimated optical flow is then applied to the current image in order to generate the next frame using a convolutional decoder. This architecture is demonstrated in Figure 3.4.

The vanilla version[4] of the described model uses a single convolutional layer in the spatial encoder or decoder with 32 filters and a kernel of size 7×7 each. The temporal encoder comes with the same kernel size, but 45 filters from one time step to the next. This model exhibits a total number of only 703,651 model parameters.

For the assessment of the model, they used the Moving MNIST dataset as well, but unfortunately in context of single frame prediction only. The binary cross-entropy cost

[4]Vanilla version: an often used term in deep learning that refers to the standard configuration of something.

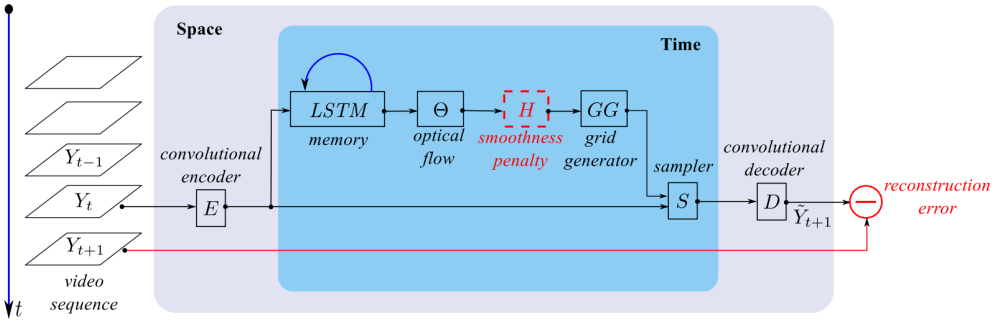


Figure 3.4: The spatio-temporal video autoencoder model with its components that are separated by the space and time domain. (From [PHC16])

function in combination with binary MNIST data was used in the first as well, but also other loss functions for floating-point MNIST data like ℓ_2, a variant of gradient difference loss, as well as the sum of both. The latter combined loss has thereby achieved the best qualitative and quantitative results in the final analysis. Some sample results from their paper are depicted in Figure 3.5, including comparisons to other architectures of previously presented models. Besides that, various depth combinations of each encoder or decoder component were evaluated as well. They ended up with the final result that deeper models clearly achieve better performance, regardless of whether it is about the space or time domain.

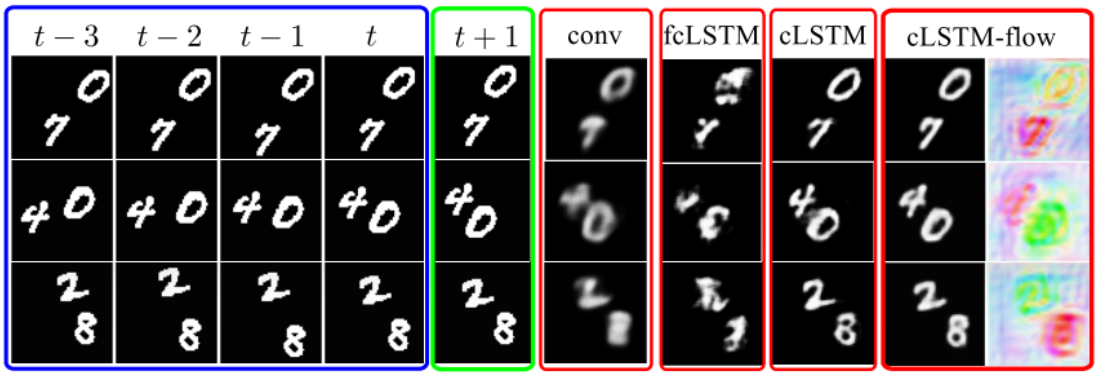


Figure 3.5: Qualitative comparison of results on MovingMNIST using different model architectures. Blue: the last 4 out of 10 frames fed into the network. Green: the next frame's ground truth. Red: Prediction results on various models in comparison, where *conv* is a simple convolutional autoencoder that handles each frame as a separate input channel, and *cLSTM-flow* is the model presented in this section. (From [PHC16])

A possible downside of this model is the fact that is was trained to predict one single frame only. Even though the predicted results in Figure 3.5 look very similar to its ground truth future, it was not shown how the performance continues over the course

of the following predictions in the future. One may assume that because each single predicted frame is slightly blurry compared to images from the input sequence, it might happen that the prediction quality of further frames could decrease rapidly. As a matter of fact, the model has never seen any blurry image example during the whole training process. At least such a fast decrease in quality was experiencedd during the development of the final architecture, after starting with a simple convolutional encoder-decoder model that produced a single frame only, together with a sliding window approach to produce the following future frames.

On the opposite side, this work offers several interesting insights to learn from as well. For one thing, it emphasizes the power of ConvLSTM cells regarding spatio-temporal learning once more, for another thing, it has shown that the use of convolutional percepts as input to the recurrent encoder offers another possibility to improve the results. The latter also reduces the dimensionality that has to be processed by condensing the input data. The final model architecture, that is presented in Chapter 4, takes advantage from both of them.

3.3 Adversarial Network Approach

In the final stage of this thesis, we came across a novel approach to train neural networks to perform frame prediction without using recurrent cells. The authors of [MCL16] used a very simple, overcomplete convolutional generator network $G(X)$ in order to generate a single or multiple frames from an input sequence X. This simple network is displayed in Figure 3.6 and consists of only convolutional layers, with a constant height and width but variable number of feature maps. Training a model of such a simple architecture has several weaknesses, such as that it could only capture short-range dependencies across the entire input sequence with the size of the kernel, due to the fixed size feature maps. Also, default loss functions such as ℓ_1 or ℓ_2 during the training experientially lead to blurry results, as previous studies have already shown.

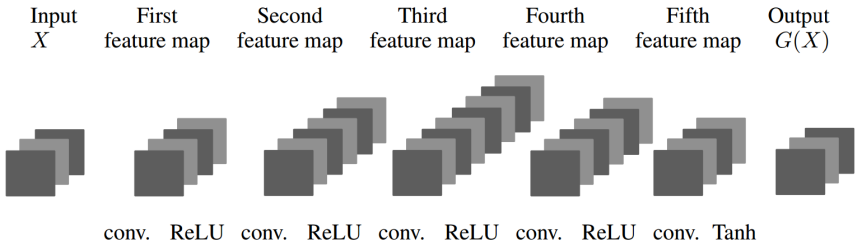


Figure 3.6: A very basic convolutional network that maps a fixed number of input frames X to predict one or multiple future frames $\hat{Y} = G(X)$. The feature maps exhibit the same height and width at each layer, but different depths. (From [MCL16])

To overcome these issues, they proposed three different but complementary learning strategies. Firstly, they used a multi-scale approach where multiple generator networks are iteratively trained on different scales of the input patch, starting from the lowest scale. The prediction of the next larger scale then used the upscaled prediction of the previous scale as a starting point. This technique enabled the network to consider motion patterns of longer range[5]. Secondly, they extended their loss function with an additional GDL term in order to penalize blurry outputs in image space. This gradient-based loss function has already been introduced in Section 2.6.3. And lastly, they plugged this simple convolutional network into the adversarial training framework. The described network therefore defines a generator network G to predict the next frame, while a second discriminator network D is consulted in order to assess whether the output of the generator network is the real target frame of the future or just a generated fake. Using an alternating training procedure, both networks learn to perfect the system. In other words, the discriminator network of this adversarial training process can be seen as an adaptive loss layer that assesses the generated output in feature space. Also other works highlight the usefulness of considering the error in feature space in addition to the image space error, such as in [DB16]. Summarizing the last two proposed learning strategies, a triplet loss is used where a standard loss function in image space in combined with a perceptional motivated loss function to preserve sharpness, as well as an adversarial error function which quantifies the realness of the generated frames.

Figure 3.7: Comparison of different loss function combinations using a simple CNN to predict one frames given four inputs. The second future frame is predicted recursively. (From [MCL16])

The performance of different loss function combinations was also compared in a qualitative evaluation. Several output samples are shown in Figure 3.7. As can be seen, the combination of multiple loss functions with different objectives enables to end up with predictions of higher quality and realism. Besides that, a detailed comparison to

[5]It is to add that networks using ConvLSTM cells do not suffer from the issue of being limited to short-range dependencies in such an extend, because the used state-to-state kernel size determines the maximum motion from one time step to the next. In contrast, the kernel size of this convolutional autoencoder defines the maximum motion across the entire input sequence. Thus, a multi-scale approach is not necessarily required for networks using ConvLSTMs for this purpose.

other LSTM based models on the UCF-101 video dataset was given. Thus, it allows us to compare our outcomes to all these results as well.

But even when this network using all the proposed learning strategies is able to produce outstanding frame prediction results, it comes with some weaknesses nevertheless. For instance, the temporal correlations of the input sequence are not explicitly modelled in the generator network. Hence, it has to explore the sequential structure of the data by its own from scratch. Furthermore, adversarial networks are said to be hard to train, because the oscillating loss values of the generator and discriminator networks are tough to interpret. Also, experience is of advantage, since the learning rates of both networks have to be kept in balance.

4 A Fully-Convolutional LSTM Encoder-Predictor Model

This chapter presents the final model that is used in this thesis for future frame prediction. It combines the insights and strengths of previous works that have been analyzed in the preceding chapter. Each of its components is investigated in detail and then assembled to end up with the overall network architecture. But beforehand, two cruicial techniques are presented which improve the learning of spatio-temporal features and the recurrent training performance.

4.1 Techniques

In this section, two important methods are presented that enable recurrent network based models to reach better performance in a shorter time of training. The next section then uses these techniques as building blocks while describing the network architecture.

4.1.1 Convolutional LSTM

As discussed in earlier chapters, the standard LSTM cell has the drawback of lacking spatial correlations in the input-to-hidden and hidden-to-hidden transitions, because it operates over sequences of vectors with only one dimension. The LSTM activations are computed based on linear transformations using fully-connected layers and subsequent non-linearities. The authors of [Shi+15] propose a variant of the LSTM cell with the core idea to handle all inputs, hidden states, cell or gate outputs as 3D tensors to preserve the spatial data properties. This is realized by exchanging the internal matrix products by convolution operations. The resulting recurrent cell type is called *convolutional LSTM*, as mentioned earlier in Section 3.2.2.

In contrast to the realization in [PHC16], peephole connections are included in our implementation of ConvLSTM. These allow the gated units to have direct access to the previous memory cell state. In addition, it can be further extended to use optional batch normalization layers in both input-to-hidden and hidden-to-hidden transitions to allow faster learning and to take advantage of its other benefits which have been described

in Section 2.5. The latter modification was originally proposed for the standard LSTM cells in [Coo+15]. All this together can be formulated as:

$$
\begin{aligned}
\begin{pmatrix} \tilde{\mathbf{f}}^{(\tau)} \\ \tilde{\mathbf{i}}^{(\tau)} \\ \tilde{\mathbf{o}}^{(\tau)} \end{pmatrix} &= \mathrm{BN}(\mathbf{W}_h * \mathbf{h}^{(\tau-1)}; \gamma_h, \beta_h) + \mathrm{BN}(\mathbf{W}_x * \mathbf{x}^{(\tau)}; \gamma_x, \beta_x) + \mathbf{W}_{peep} \odot \mathbf{C}^{(\tau-1)} + \mathbf{b} \\
\hat{\mathbf{c}}^{(\tau)} &= tanh(\mathrm{BN}(\mathbf{W}_{hc} * \mathbf{h}^{(\tau-1)}; \gamma_h, \beta_h) + \mathrm{BN}(\mathbf{W}_{xc} * \mathbf{x}^{(\tau)}; \gamma_x, \beta_x) + \mathbf{b}_c) \\
\mathbf{C}^{(\tau)} &= \sigma(\tilde{\mathbf{f}}^{(\tau)}) \odot \mathbf{C}^{(\tau-1)} + \sigma(\tilde{\mathbf{i}}^{(\tau)}) \odot \hat{\mathbf{c}}^{(\tau)} \\
\mathbf{h}^{(\tau)} &= \sigma(\tilde{\mathbf{o}}^{(\tau)}) \odot tanh(\mathrm{BN}(\mathbf{C}^{(\tau)}; \gamma_c, \beta_c))
\end{aligned}
\tag{4.1}
$$

where $\mathbf{W}_h \in \mathbb{R}^{d_h \times 3d_h}$ and $\mathbf{W}_{hc} \in \mathbb{R}^{d_h \times d_h}$ are the shared weights for the hidden-to-hidden transitions at time step τ, $\mathbf{W}_x \in \mathbb{R}^{d_x \times 3d_h}$ and $\mathbf{W}_{xc} \in \mathbb{R}^{d_x \times d_h}$ are the shared weights for the input-to-hidden connections, as well as $\mathbf{W}_{peep} \in \mathbb{R}^{d_x \times 3d_h}$ are the shared weights for the peephole connections. Next, $\mathbf{b} \in \mathbb{R}^{3d_h}$ and $\mathbf{b}_c \in \mathbb{R}^{d_h}$ are the biases, as well as $\mathbf{C}^{(0)}, \mathbf{h}^{(0)} \in \mathbb{R}^{4d_h}$ are the initial states of the memory cell and the hidden state, respectively. Furthermore, $*$ denotes the convolution operation and $\mathrm{BN}(\mathbf{x}; \gamma, \beta)$ a batch normalization layer with its learned shift γ and scale β. As in [Coo+15], β_h and β_x are set to zero by default to avoid the unnecessary redundancy with the existing bias terms $\mathbf{b}$ and $\mathbf{b}_c$. The structure of such a cell is visualized in Figure 4.1.

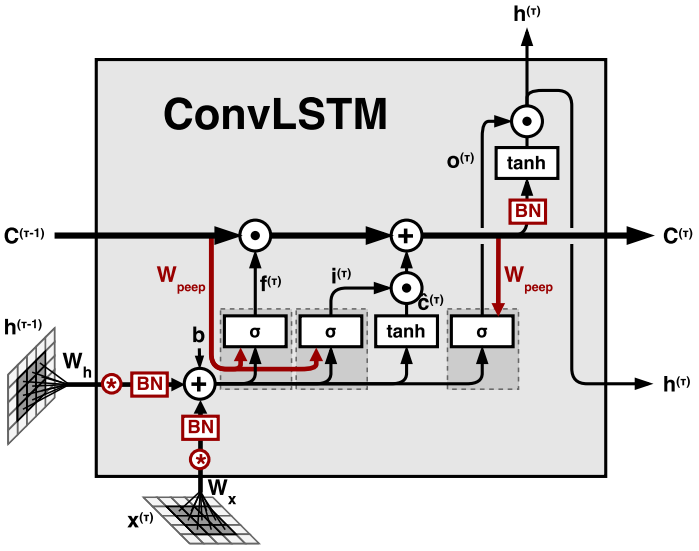


Figure 4.1: The simplified structure of the batch-normalized ConvLSTM cell including peephole connections. The inputs and previous hidden states are convolved to produce 3D tensors that flow through each cell. Changes to standard FC-LSTM are highlighted in red.

4.1.2 Scheduled Sampling

When the first recurrent network based models were tested in the course of this thesis, a discrepancy between training and inference was discovered. At training time, each cell is usually fed with the ground truth $\mathbf{x}^{(\tau-1)}$ of the previous time step in order to generate the current prediction $\hat{\mathbf{x}}^{(\tau)}$. In contrast, the generated frame $\hat{\mathbf{x}}^{(\tau-1)}$ of the previous cell is used instead as cell input during inference, because no ground truth is available in that case. Thus, mistakes made earlier in the sequence flow through all following cells and can quickly amplify since the network has never dealt with such errors at training time. In context of future frame generation, this effect was identified when the prediction quality has dropped tremendously after the first generated frame. The fact that predicted frames usually look slightly blurry is the clear reason for this, because a network that was fed purely with ground truth input images has never seen pictures with smooth edges.

As a first step, the recurrent cell's input during the training process was therefore adapted to behave in the same way as in inference mode. Therefore, a recurrent cell that is trained to generate a future frame has to condition on the previously generated frame instead of the ground truth[1]. This strategy can be seen as an automatic form of data augmentation or a natural regularizer and helps the network to learn a robustness against imperfect inputs. However, this comes with the downside that the RNN model converges much slower during the training, because it has to predict the correct output given poor or even wrong input data.

To combine the best of both worlds, an experiment with a specific training strategy was performed where each recurrent cell started to condition on the ground truth frame and have slowly changed to the inference mode conditioning using a probability variable p with linear or exponential decay. In this process, a random number $r \in [0, 1]$ is generated at each training step and the RNN cells condition on the generated frame when $r \geq p$. First experiments have shown positive results, because a better performance could be reached compared to both other strategies mentioned earlier.

Shortly afterwards, we came across [Ben+15] where the authors proposed a similar but even more radical strategy and demonstrate insightful evaluation results. Their so-called *scheduled sampling* (SS) training strategy for recurrent networks is based on the same core idea, but instead of generating only a single random number r for the whole input sequence, they proposed to do this for every single time step τ. In addition,

[1] Always conditioning on the generated outputs while training is called *always sampling* (AS) according to [Ben+15].

they suggested to use an inverse sigmoid decay function in order to provide a smooth transition from training mode to inference mode:

$$\sigma_{inv}(i;\alpha) = \frac{\alpha}{\alpha + exp(i/\alpha)}, \tag{4.2}$$

where i identifies the current training step and $\alpha \geq 1$ controls the expected speed of convergence. This function and a diagram of the scheduled sampling approach is depicted in Figure 4.2.

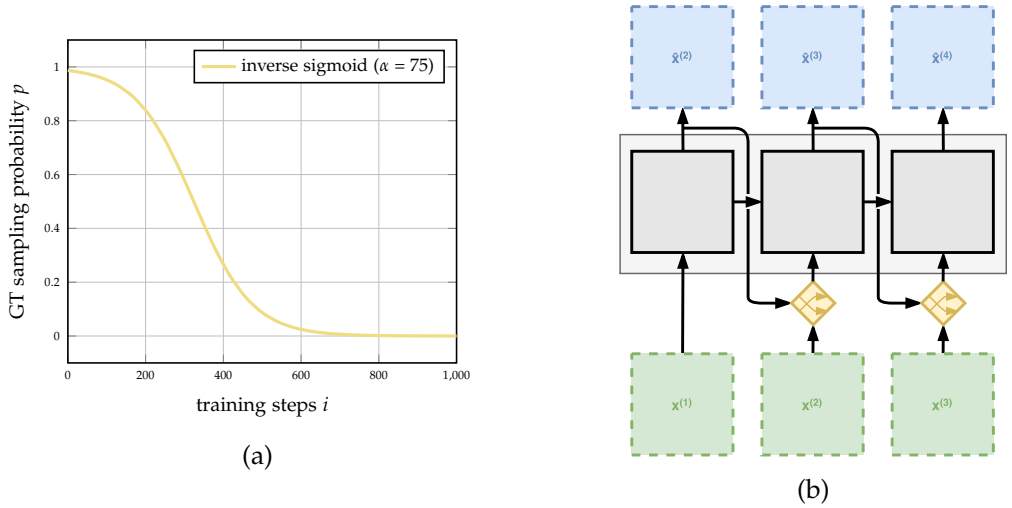


Figure 4.2: Illustration of scheduled sampling, where (a) shows the inverse sigmoid decay function and (b) the structure of a RNN that uses this approach. The yellow blocks symbolize the scheduled sampling components, which decide whether a cell conditions on the ground truth or the previous output by flipping an unbiased coin.

4.2 Network Architecture

The final model architecture is based on the RNN encoder-decoder framework introduced in Section 2.4.2 and both previously presented advancements to improve the learning of spatio-temporal features in video data. This architecture is chosen because it explicitly models the temporal correlations and is flexible regarding the input and output sequence. Furthermore, previous works demonstrated in Chapter 3 have shown promising results using this recurrent framework. The architecture that is presented in this section describes the default version of the network that is used for frame prediction of generated image sequences of animated, handwritten numbers. The following evaluation in Chapter 6 assesses this model with different settings, such as variations in the network's depth.

4.2.1 Components

Before the entire network model is demonstrated, we take a detailed look at its main components first. These components for encoding or decoding are ordered according to the flow of data while propagating forward through the network. Afterwards, the loss layer is presented which combines standard error functions with perceptual motivated bias terms.

Spatial Encoder

Instead of feeding the recurrent encoder directly with raw image data like in [SMS15] or [Shi+15], each frame flows through a multilayer CNN first. Therefore, it is projected to an image percept in feature space of lower size, but higher depth. This single convolutional network is shared across the entire time domain of the spatio-temporal encoder component, which is described in the next section. The motivation to use this spatial encoder is based on [PHC16] who suggest that a deeper encoding of the input image can yield better results. Furthermore, decreasing the height and width of the image has a positive impact on the overall runtime and memory efficiency. Although the number of feature maps and hence the dimensionality of the convolved frame percepts is higher compared to the original image, it does not harm these two advantages with respect to the total efficiency of the model. This is due the fact that the ConvLSTM cells and their convolutional state-to-state transitions within the following recurrent encoder increase the feature space representation's depth either way. The spatial encoder component is illustrated in Figure 4.3.

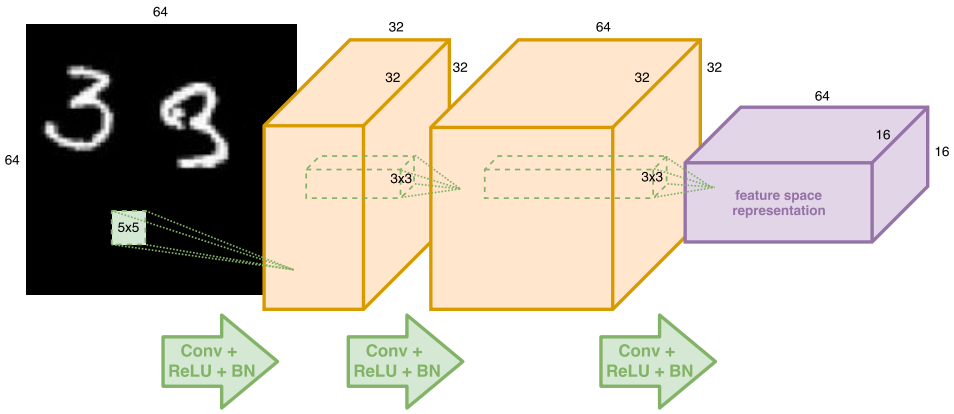


Figure 4.3: Spatial encoding network that maps each input image to its feature space representation (purple) using convolutional layers with ReLU activations and subsequent batch normalizations (green). Intermediate tensors (yellow) are denoted with their height, width and depth.

No max pooling layers are used because rotation tolerance would be counter-productive in context of frame prediction. This kind of downsampling could remove important details that might be required to successfully reconstruct the image. Instead, each frame is downsampled by using a stride of $s = 2$ in specific convolutional layers with the motivation that the network itself should learn how to perform such a downsampling operation. Additionally, each convolutional layer is activated using ReLUs and followed by a batch normalization layer to compensate the internal covariate shift of such a deep network model. The resulting frame percepts exhibit a shape of $16 \times 16 \times 64$.

Spatio-Temporal Encoder

After each percept tensor was produced by the spatial encoder CNN, it flows unchanged into a recurrent network to preserve sequential correlations and to learn temporal dynamics. To be more precise, ConvLSTM cells are used to retain the spatial structure of the three-dimensional input data. The cell state $C^{(0)}$ and hidden state $h^{(0)}$ of the spatio-temporal encoder network are initialized with zero by default. Moreover, all convolutional layers within the ConvLSTM cells adapt their number of produced features maps according to the input's depth in order to keep all shape sizes constant. After the whole input sequence of about ten frames has been processed, the resulting cell and hidden states of the last unit then encode the learned motion of the sequence. This representation is then transferred to the following prediction component. It should be noted that the hidden outputs of the encoder's ConvLSTM cells are discarded and not used in the subsequent decoding network. The spatio-temporal encoder component is shown in Figure 4.4.

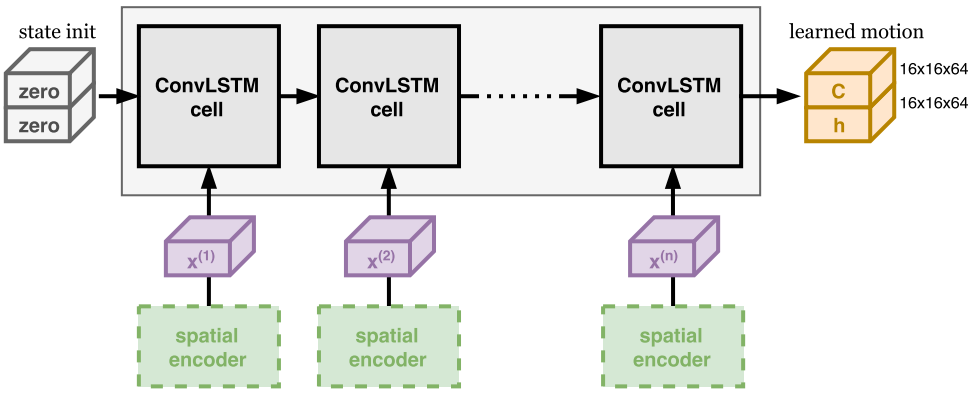


Figure 4.4: Spatio-temporal encoder network that encodes a motion representation (orange) based on a frame sequence in feature space (purple). The feature space representations are mapped from each single frame in image space and are produced by the previously described spatial encoder (green).

Spatio-Temporal Predictor

Initialized by the last cell and hidden state of the previous ConvLSTM encoder, the recurrent predictor component takes over to map each frame in feature space at time step τ to its future representation at time step $\tau + 1$. The spatio-temporal decoder takes advantage of the ConvLSTM cells once more, but it handles the inputs and outputs of each cell in a different way.

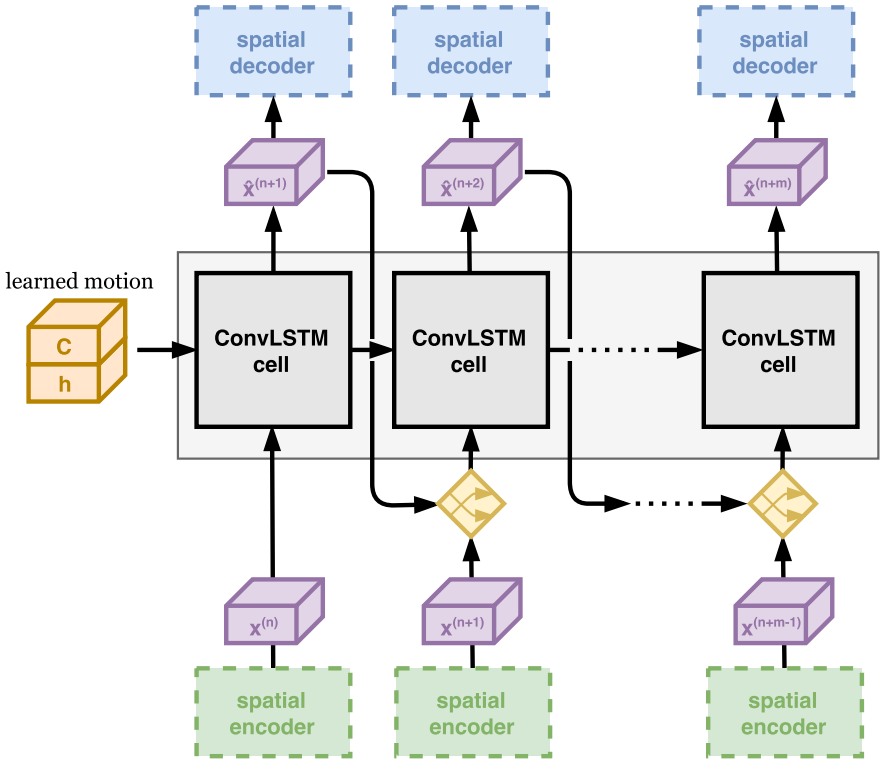


Figure 4.5: Spatio-temporal decoding network that maps a frame in future space to its future representation. The ConvLSTM is initialized with the learned motion representation (orange) produced by the recurrent encoder. Each ConvLSTM cell receives either the ground truth representation or the output of the previous cell as its input while training. This decision is made by scheduled sampling components (yellow) at each time step.

As depicted in Figure 4.5, the prediction component utilizes the scheduled sampling technique, earlier presented in Section 4.1.2, to improve the training performance. By default, the network begins to condition on ground truth frame representations in feature space that were produced by the spatial encoder component. By decaying the probability of conditioning on the ground truth using an inverse sigmoid function with $\alpha = 1000$, the predictor RNN slowly starts to condition on previously generated frames like in inference mode. With this setting, it takes about 10,000 training steps until this

component predicts future frame representations entirely based on generated outputs. After the entire output sequence has been generated, each single tensor still exhibits a shape of size $16 \times 16 \times 64$ and altogether represent the video's future in feature space.

Spatial Decoder

To map each frame representation back to the image space, a second CNN is modelled that performs the same transformation steps of the spatial encoder but in reverse order. It therefore uses transposed convolutional layers with rectifier units and batch normalization layers in between. However, the activation function of the output layer is either a sigmoid or a hyperbolic tangent function to ensure that the generated frames exhibit a valid scale of values. The described spatial decoder is illustrated in Figure 4.6.

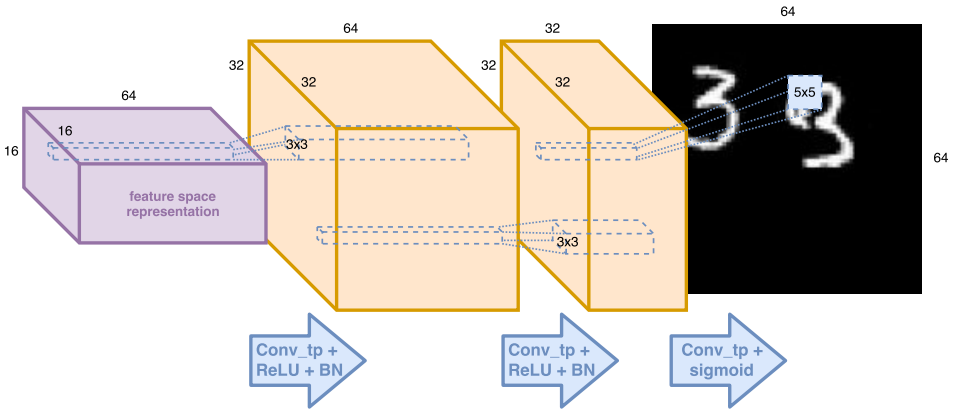


Figure 4.6: Spatial decoding network, which learns to reverse the mapping of the spatial decoder using transposed convolutional layers (blue), to map a feature space representation back to original image space.

Loss Layer

The last component which has to be mentioned is the loss layer used while training the model. Since previous works have shown the problems of using standard loss functions like ℓ_2 in image processing tasks, as earlier described in Section 2.6.3, the main objective $\mathcal{L}_{\text{main}}$ used in this work is extended by two perceptual motivated bias terms that respect the human notion of visual similarity. The main loss function itself is a placeholder for any commonly used loss function. Depending on the data characteristics, $\mathcal{L}_{\text{bce}}$, $\mathcal{L}_{\text{mse}}$ or $\mathcal{L}_{\text{mae}}$ is therefore used in this context. To counteract the lack of sharpness in predicted frames, a GDL objective function is used as a first extension in order to quantify the sharpness of the image. As a second extension, the SSIM-based loss function is used to assess the image's luminance, contrast and structure. The SSIM-kernel size is chosen

to be $k = 5$, reasoned by the fact that only small image patches are used during the training process, as well as to improve computational efficiency. The same reasons have also led to the decision to not use the MS-SSIM index. Moreover, RGB image patches are temporarily converted to grayscale which is required to calculate the structural similarity index. Combining these three loss terms together with different weights results in the *triplet loss function* that is used within our experiments:

$$\mathcal{L}_{\text{triplet}}(\mathbf{X}, \hat{\mathbf{X}}) = \lambda_{\text{main}}\mathcal{L}_{\text{main}}(\mathbf{X}, \hat{\mathbf{X}}) + \lambda_{\text{gdl}}\mathcal{L}_{\text{gdl}}(\mathbf{X}, \hat{\mathbf{X}}) + \lambda_{\text{ssim}}\mathcal{L}_{\text{ssim}}(\mathbf{X}, \hat{\mathbf{X}}), \tag{4.3}$$

where $\mathbf{X}$ denotes the ground truth target sequence and $\hat{\mathbf{X}}$ the network's predicted future sequence. The weight and bias parameters of each loss function are neglected for the benefit of simplicity. The coefficients λ_{main}, λ_{gdl} and λ_{ssim} control the relative importance of each component in the triplet loss function. A simple default setting is to weight them equally by setting $\lambda_{\text{main}} = \lambda_{\text{gdl}} = \lambda_{\text{ssim}} = 1$.

Additionally, experiments similar to [DB16] were performed where a further loss was injected to quantify the similarity of feature space representations. Unfortunately, the results could not be improved with such an additional term, because it is not trivial to find an appropriate coefficient λ_{feat} to weight this feature space loss relative to other image space losses. On the one hand, it has no visible effect when the coefficient is chosen too low. On the other hand, the accuracy of the predictions decreases tremendously when the coefficient is chosen too high. Due to the difficulties in finding a good balance, as well in the interest of time, such a feature space loss is not used in the final model.

4.2.2 Model

For a better overview regarding the resulting model complexity, Table 4.1 lists the number of trainable parameters for each network component described previously.

Component	Trainable parameters
Spatial encoder (CNN)	56,416
Spatio-temporal encoder (ConvLSTM)	864,512
Spatio-temporal decoder (ConvLSTM)	864,512
Spatial decoder (CNN)	56,353
Total	1,841,793

Table 4.1: Overview of the network parameters per component in the described vanilla setting using 1-layer ConvLSTMs.

In order to provide an overview, each component is put together in Figure 4.7 to demonstrate the entire model. It has to be noted that not every depicted building block

or connection is used when a future sequence is predicted within an inference step. For example, only half as many frames have to be processed by the spatial encoder, because the feature representations of the ground truth target sequence are obviously not available when an actual future prediction is performed. The same is also true for the scheduled sampling components and the loss layer that are only required to train the model. Additionally, the data tensors of each single frame is processed by a separate CNN. Therefore, the trainable parameters in each convolutional layer of the spatial encoder and decoder has to be shared across the whole input sequence.

Since the overall architecture follows the concept of an encoder-decoder network, it should be argued why this model is likely to learn useful features. According to the argumentation in [SMS15, p. 3f.], it is unlikely to learn the trivial function for the following two reasons. First, an entire sequence of variable size as well as the temporal dynamics have to be encoded and decoded using a fixed-sized representation. In order to accurately predict future frames multiple time steps ahead, the decoder network really has to come up with a learned representation that can distinguish several foreground objects from static background, as well as understand the motion of object and the environmental constraints within a given video scene. Second, the learned motion pattern has to generalize well so that it can be applied to any time step of the sequence. Performing a simple copy of the last state might be almost sufficient when a single frame is predicted only, but not when it has to look further into the future.

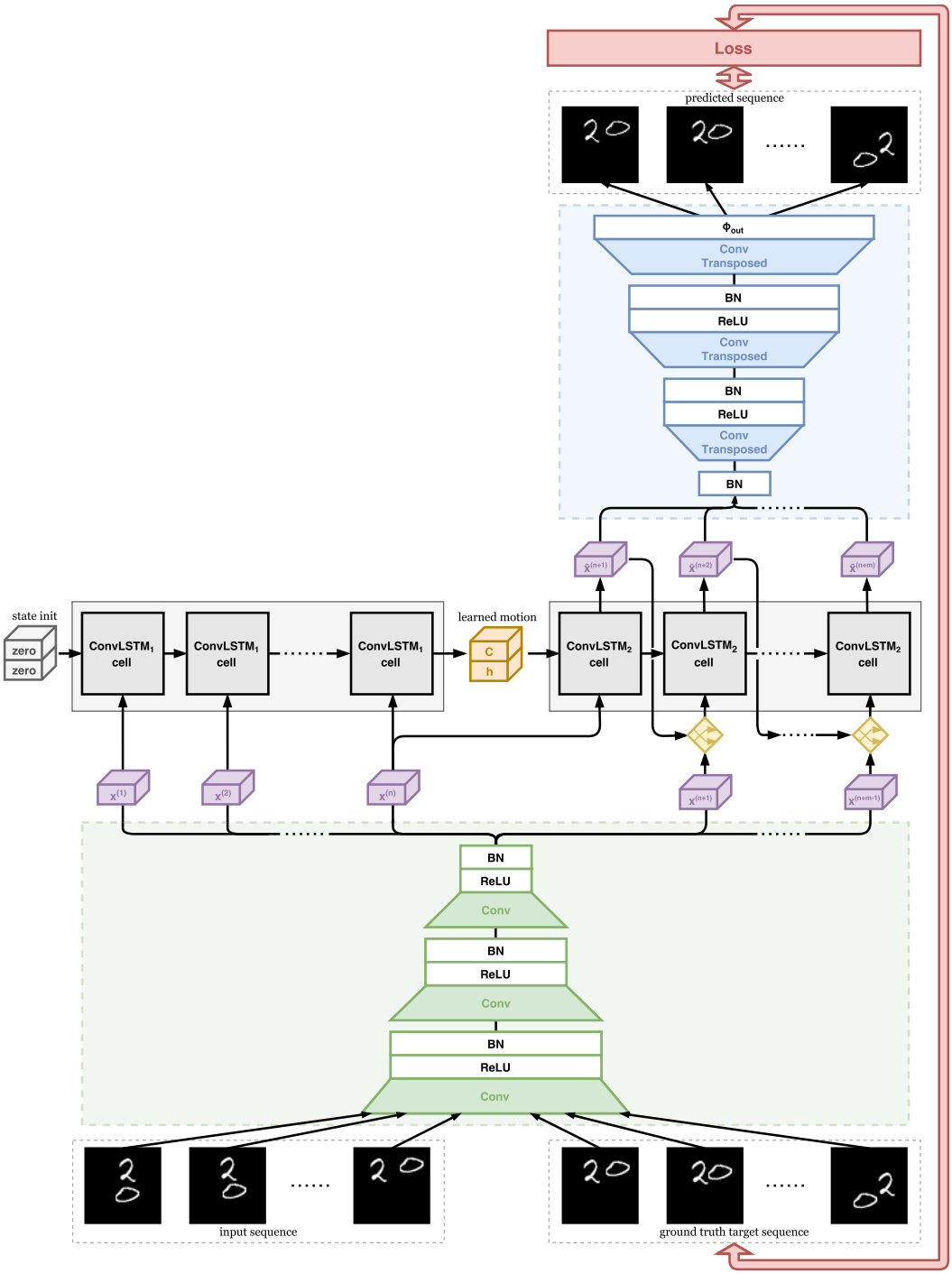


Figure 4.7: The ConvLSTM Encoder-Predictor Model. Weights of the convolutional encoder (green) and decoder (blue) are shared layer-wise across the whole sequence. Spatial encodings of the ground truth frames, scheduled sampling components (yellow) and the loss layer (red) are only used while training.

5 Datasets

This chapter presents all datasets that are going to be used in the following evaluation. Three different video datasets were chosen that are used in related works in order to be able to compare the results and analyze the strengths and weaknesses of different network models. The selected datasets will be introduced one after another, ordered by the content complexity with respect to the possible variations in color, motion and physical environment. Additionally, random samples from each dataset are shown to get a better idea of how the data looks like that is fed to the network.

5.1 Moving MNIST

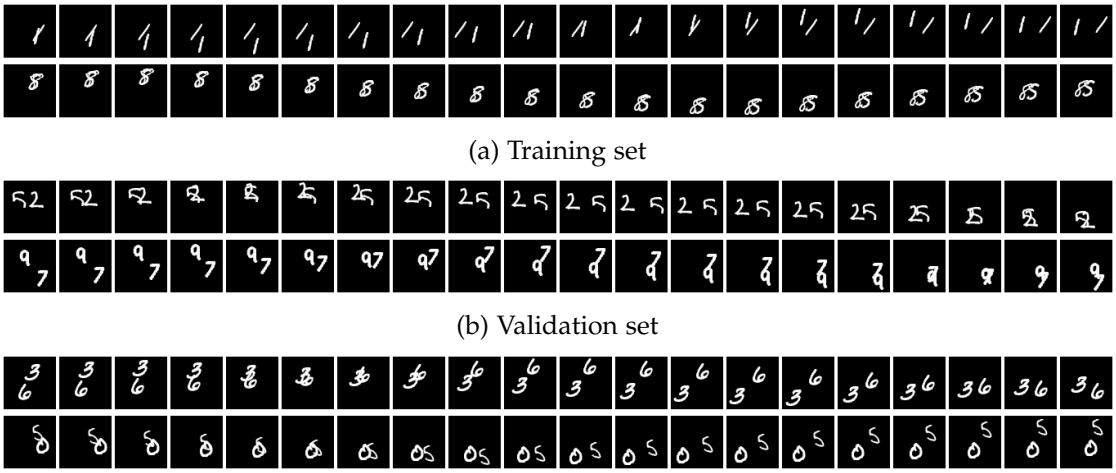
The model is trained on a synthetic dataset of black and white images with flying handwritten digits. The *Moving MNIST* has been introduced in [SMS15] and applied in context of video frame prediction. Since then, it was used several times in different follow-up works like [PHC16] or [Shi+15].

5.1.1 Characteristics and Data Generation

In the proposed setting, each sequence consists of 20 image frames at a size of 64×64 with two random moving digits from the MNIST[1] dataset in it. One major advantage of this simple dataset is that it exhibits a nearly unlimited size, because it can be generated on the fly. When training a model, two random digits are therefore randomly chosen from the first 55,000 digits of the training set and place them on any location of the first image patch. For the generation of subsequent frames, a velocity is assigned to each digit, whose direction is chosen uniformly from a unit circle. Further, the simple physical rule is applied that the angle of incidence is equal to the angle of reflection when any digit at a size of 28×28 touches the wall. This enables other interesting properties of the dataset, such as basic dynamics due to having to predict the right trajectory after bouncing off a wall, as well as multiple occlusion effects of overlapping digits. Consequently, even though that the generation process of the dataset is that simple, it is hard for a model to generate accurate predictions in the test set without

[1] MNIST dataset of handwritten digits: `http://yann.lecun.com/exdb/mnist/`

learning a representation that encodes the internal motion of the system [Shi+15, p. 6]. Last but not least, having a simpler dataset at hand allows us to gain a better understanding of the model's behavior in respect to its hyperparameters. Especially in consideration of the very long training time when more complex or even natural videos are used.



(a) Training set

(b) Validation set

(c) Test set

Figure 5.1: Randomly chosen samples of generated image sequences of size 64×64 from the Moving MNIST dataset.

The generation procedure of the validation set is equal to the previously described process to create the training data, but with the difference that the last 5000 digits of the original MNIST training split is used. On the contrary, the test set is not generated by using the MNIST test split. Instead, the pre-generated test set of [PHC16][2], that contains exactly 20 frames long sequences with 10,000 examples has been used. In this way, more comparable results can be achieved to at least one competing model. A random sample from each of these splits is presented in Figure 5.1.

Even though some other works have used only a fixed number of pre-generated frame sequences, the on-the-fly generation process of the initial paper was kept for at least three reasons. First, it limits the amount of data and therefore increases the chance of overfitting. Seconds, loading the pre-generated frames from disk takes more time than generating them on the fly; hence it could have a slightly negative impact on the overall training time. And third, it reduces the total memory requirements in case the whole data would otherwise be pre-loaded into memory in order to eliminate the second mentioned issue.

[2]Pre-generated Moving MNIST test set with 10,000 sequences: `http://mi.eng.cam.ac.uk/~vp344/`

5.1.2 Data Preprocessing

Since the original pixel values of the MNIST dataset are in a range of $[0, 255]$, simple rescaling to $[0, 1]$ is performed in order to normalize the data. Further, subtracting the mean pixel-values has been considered as well because grayscale images have the stationarity property[3]. But due to the fact that this is not often done in practice when the MNIST dataset is used [Ng+13] and no noticeable improvements could be seen when applying it, caused that mean subtraction is not applied while preprocessing the data. Moreover, one can believe that since most parts of any image is black (and therefore zero), further processing of the data is not necessarily required.

Additionally, instead of feeding the model with continuous floating values in the normalized range, binary pixel values are used only. This decision results from the use of binary cross-entropy as the main loss function for this dataset, which has shown to be the favorable choice for image generating models with MNIST [SMS15], [Shi+15]. Therefore, every pixel p is set to zero if $p < 0.5$, and a value of one is assigned to all other pixels. Moreover, the sigmoid activation function is used in the output layer in order to support the saturation of all pixels into either zero or one.

5.2 MsPacman

In order to accommodate the request of assessing our model on more complex data compared to the previously presented dataset, but which still has negligible dynamics compared to natural clips, a second data collection is used that consists of video game recordings. The *MsPacman*[4] dataset has been used in an independent TensorFlow implementation of [MCL16][5] for adversarial video generation and future frame prediction. It contains several interesting dynamics and properties that make it a reasonable choice to be considered in the evaluation of our model. These will be discussed in more detail in the course of this section.

5.2.1 Characteristics

This video game dataset contains about half a million single images at 160×210. These images can be grouped into 517 game recordings for the training set and 51 recordings for the test set. Each recording has a variable length and ranges from about 500-1500 frames per sequence. The last 51 sequences of the training data are taken for the

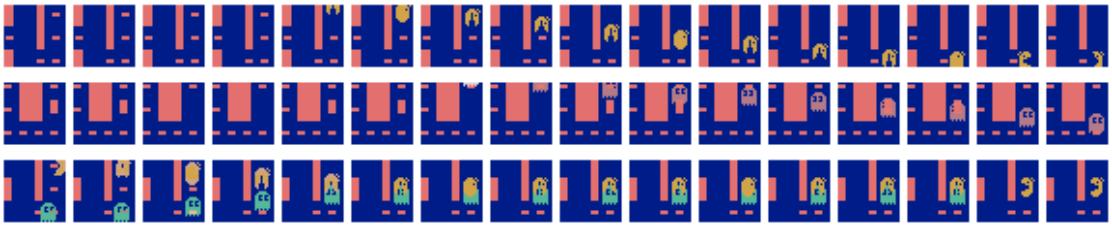
[3] For example when the statistics for each data dimension follows the same distribution, the data is said to be stationary.

[4] MsPacman dataset: generated by *Jun Ki Lee* in a student project at Brown University by taking recordings of the classical video game.

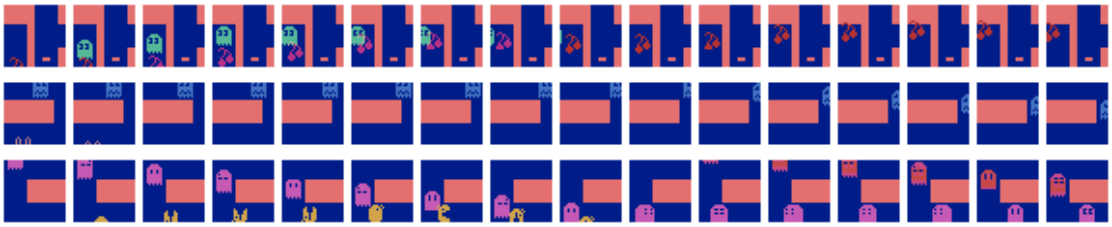
[5] Adversarial video generation in TensorFlow: [Coo16]

validation set to have it roughly the same size as the testing set. Thus, it ends up with a total number of 418,287 frames for training, 45,007 images for validation and 46,380 images for testing.

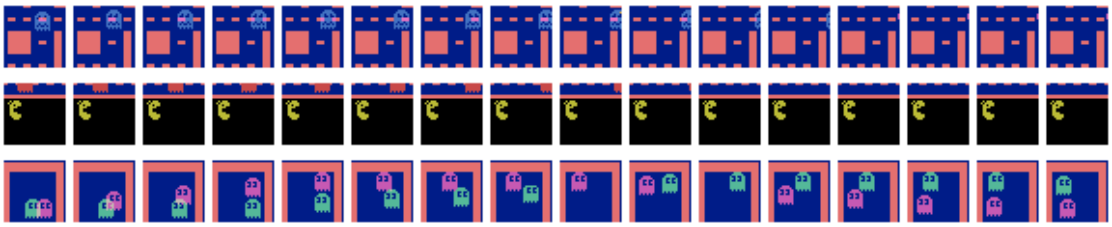
The action inside each recording acts in a closed world with various game rules that the network has to understand. Just to name some example, all objects follow the path between the walls of the game world, with the two exceptions that ghosts in the center might exit the cave through the top barrier, as well as that Pacman[6] can teleport himself by leaving the world using its left or right exit. Furthermore, the game's main character opens and closes its mouth, as well as can eat the distributed dots, fruits or blue ghosts. Some recordings from the different dataset splits are depicted in Figure A.2.



(a) Training set



(b) Validation set



(c) Test set

Figure 5.2: Example sequences from different splits of the used MsPacman dataset. These randomly selected frames have been cropped to 32×32 and filtered to ensure enough movement while training the model.

[6]Pacman: name of the main character of the video game in this context.

5.2.2 Data Preprocessing

The images of every sequence are rescaled to be in range of [-1, 1]. To ensure that the predicted outputs exhibit the same value scale, a *tanh* activation function is used after the last transposed convolutional layer. Since each single image is quite big, advantage from the fully-convolutional approach is taken; hence the the network is only trained on random crops at a size of 32×32. Some random samples of these cropped images are shown in Figure 5.2. Unfortunately, the usage of small image crops follows that most of the used training sequences show no motion at all.

After experiencing a strong preference of predicting the worlds background over the small moving objects only, a technique similar to [MCL16] is used in order to ensure that each temporal sequence shows enough movement. Therefore, each time when a random part of the sequence is selected from the training set, the ℓ_2 difference between each consecutive frame pair is calculated. Afterwards, this randomly chosen cropped sequence is rejected until the overall movement of the input sequence is higher than a given threshold of $25 \cdot n_{frames}$, or a specified repetition limit is reached in order to prevent an endless loop. Additionally, it is checked that there is also movement at the end of the input sequence to ensure that the whole movement is not just taking place at the beginning of the whole sequence only. Further, it reduces the chance that the network has to guess the movement of objects that enter the patch after the prediction-decoder has taken over, which it obviously cannot predict. Last but not least, the input and output sequence length has been slightly shortened compared to the other two datasets to 8 frames each. The reason for this is that this performed motion detection is almost inefficient when being applied on too long sequences, because of the high chance that all these fast moving objects within the selected crop have already left the scene too quickly. However, even though this filtering of the final training examples sounds quite radical, it is to highlight that a high fraction of the input space is still static content. This can be attributed to the use of convolutional layer with small window sizes that slide over the input space.

5.2.3 Data Augmentation

In terms of data augmentation, the brightness or contrast of the image examples is *not* randomly modified, because it does not make sense in the context of this video game which uses only a fixed set of colors. But since the game world is mirrored horizontally, random horizontal flipping is performed in case the chosen crop is not showing any parts of the status display at the bottom. Furthermore, it iterates over all sequences 256 times per epoch, reasoned by the fact that the dataset consists of only a few but very long sequences. A second reason is that a very short clip and a small random crop from these frame sequences is used only.

5.3 UCF-101

Finally, a third dataset is used to examine if the model can also deal with complex, natural videos. Therefore, the *UCF-101* dataset [SZS12][7] is used. With its 13,320 clips and about 27 hours of video data, it belongs to the largest labeled dataset for human action recognition. The dataset is made of user-uploaded videos that contains both overloaded background and camera movement. All 101 categories can be divided into 25 main groups, where each video from the same group shares similar features, such as an roughly equal viewpoint. The action categories can also be separated into five types, namely human-object interaction, body-motion only, human-human interaction, playing musical instruments, and sports. Even though this dataset was originated for human action recognition, it can be used in context of frame prediction as well as by only using the raw video data.

Before taking a deeper look into the characteristics and applied preprocessing steps, it should also be briefly mentioned that there exists an even larger video dataset for the purpose of representation learning. This huge dataset is called *Sports-1M* and contains over 1.1 million YouTube video links of 478 classes that have been annotated in an automated process [Kar+14]. But due to infrastructural issues with such a huge dataset, as well as a tremendous time exposure regarding data preprocessing, it is not use in this thesis.

5.3.1 Characteristics

The average video length of the whole dataset is about 6.2 seconds, while each single can range from about 1 second to a maximum of 71 seconds. Even though the initial paper states that each video has a fixed resolution and frame rate of 320×240 and 25 FPS respectively, a handful of videos exhibit a slightly different resolution nevertheless. Consequently, these frames are padded with zeros or cropped in the center to end up in an equal size for all videos. Such a padded video, as well as other example clips, can be seen in Figure A.1b.

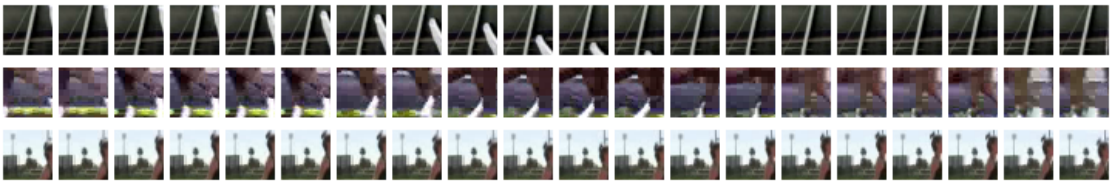
The dataset provides three standards train/test splits intended to be used for either action recognition or action detection. The third standard split for action recognition is used in this work, because it consists of the most videos for training, and allowes the simplest divisibility of the test split. Since a huge training set can expected to be fundamental for the network in order to deeply explore the inner dynamics, the validation data is taken from the test split instead of the training data as otherwise customary. For the avoidance of doubt, other previous works using UCF-101 for frame prediction either have only used 10% of the test set for actual testing [MCL16, p. 12], or

[7] UCF-101 dataset and further information: `http://crcv.ucf.edu/data/UCF101.php`

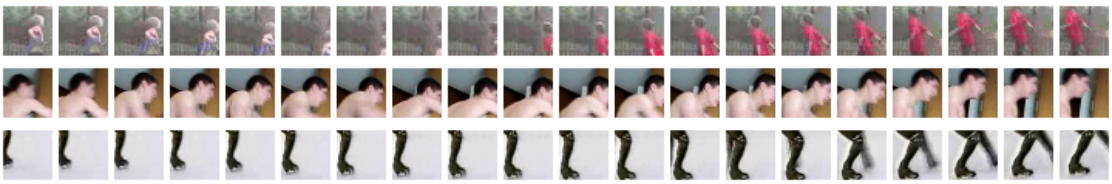
did not make any comment about which data partitions they have chosen for validation and testing. Finally, it therefore ends up with 9624, 1232 and 2464 videos for the training, validation and test set, respectively.

5.3.2 Data Preprocessing

The data preprocessing that is performed in UCF-101 is very similar to the procedure described in Section 5.2.2, but with the following differences. First, the width and height of all images are cut in half, resulting in downscaled videos at 160×120 using linear interpolation. This is done in order to compensate the noisy, pixelated artefacts in the videos. Also, it increases the chance of finding a random crop at a size of 32×32 that actually has some true motion in it, instead of just flickering caused by noise. Second, the constraint regarding motion filtering is slightly weakened, when selecting the crop region of the randomly selected clip. While a sequence, that has a very low motion in the input frames, is still rejected, it does not dismiss a sequence that has no motion at the end. Figure 5.3 shows several cropped sequence examples from the different dataset splits that are fed into our model.



(a) Training set



(b) Validation set

(c) Test set

Figure 5.3: Sequence examples from UCF-101. These frames have been randomly selected from the different splits, cropped to 32×32 and filtered to ensure they contain at least a small proportion of motion.

Furthermore, because it is wasteful to load the whole video file into memory, especially in regard that only a very small portion of the data is actually used to generate a single example for the next batch, an additional offline preprocessing of the data is performed before the actual training process starts. For that reason, it iterates over all raw video files and generates non-overlapping binary video sequences with a length of 30 frames each. Fortunately, the generated files can be reused after doing this process once. Finally, it ends up with 55,150 clips for the training set, 7183 clips for the validation set and 14,451 clips for the test set.

5.3.3 Data Augmentation

Regarding data augmentation, the contrast and brightness of the overall image sequence is randomly modified by a delta of $\pm 20\%$. Random horizontal flipping is also performed for the training data. While the contrast or brightness in the validation or test data is not randomly changed, their size is doubled by using both the normal and the flipped instances. Four crops from every video are used in each evaluation iteration in order to have a balance between more consistent evaluations and an acceptable processing time.These augmentation steps are performed on-the-fly. To facilitate this, an advanced double-buffered input queue is used. The first *filename queue* is randomly filled with references to the binary sequence files generated before. Afterwards, 16 CPU threads dequeue a reference from this queue, load the sequence into memory and then perform all preprocessing steps in parallel. This is for the purpose of generating a single training example. Finally, this example is then pushed to the *shuffled batch queue*, from which the model loads its batches in every iteration. Consequently, there is no waiting time in between each training step.

6 Evaluation

This chapter presents the evaluation results of our model on all datasets of chapter 5. Since the model exhibits a vast number of hyper parameters and each training can take several days, performing an extensive grid search is not possible. That is why the model's substantial hyperparameters are explored independently from each other by using the simplest dataset. Afterwards, the acquired knowledge about the model's behavior can then be transferred to the other datasets in order to perform a basic grid search. Beside the quantitative assessment of the model, it is also qualitatively compared to experimental results of related projects at the end of each section.

All experiments are computed on a single Nvidia GeForce GTX Titan X using the TensorFlow open source software library for machine intelligence in version r0.10 with CUDA 7.5.18 and cuDNN 4. This quite new deep learning framework is developed by Google as a second-generation system based on their experiences with DistBelief [Dea+12]. A computation in TensorFlow is expressed as a stateful dataflow graph, where each node represents a mathematical operation and the graph edges a data tensor that can flow from one node to the next [Aba+15]. It has gained increasing attention since its first public release in November 2015. The training and evaluation code is available open-source [1].

If not stated differently, the model configuration described in Chapter 4 is used, but with a two-layer ConvLSTM using 3×3 input-to-hidden and 5×5 hidden-to-hidden kernel size in the internal convolutions. Further, the optional BN-layers within these cells are deactivated for two reasons. Firstly, first tests have not shown any improvements using them. Secondly, the fact that its TensorFlow implementation back then had the issue[2] of performing batch statistic updates multiple times when being shared across time resulted in an enormous drop in performance. The use of these batch normalizations within the spatio-temporal encoder and decoder components is therefore kept for future work. Additionally, all weights are initialized using the Xavier method of equation 2.7 and the default weight decay regularization coefficient is set to $\lambda = 1e^{-5}$. A batch size of 32 is used during the training process together with Adam optimizer ($\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e^{-8}$) and a default initial learning rate of $\eta = 0.001$. In addition to the learning rate annealing mechanism of this optimizer, the learning rate is manually

[1] Repository of the project: `https://github.com/bsautermeister/imseq`
[2] Further details: `https://github.com/tensorflow/tensorflow/issues/4361`

decreased once per epoch using a stair-case exponential decay function by a factor of $\alpha = 0.9$.

Lastly, it has to be mentioned that diagrams, which show results of any kind of objective function, are always shown with its y-axis in logarithmic scale so that fine differences become better recognizable. Unless otherwise specified, all presented losses can be understood as the average per-pixel error across the whole generated future sequence.

6.1 Experiments on Moving MNIST

The synthetic Moving MNIST dataset is used as a starting point for the evaluation of our model. Given an input sequence of ten frames, the network has to understand and encode the motion of this input in order to predict the next ten frames of the future. In this section, the results are qualitatively and quantitatively compared with other network models, because this dataset was used in several previous works as well. The loss layer used in this section sets $\lambda_{\texttt{ssim}} = 0$ by default.

6.1.1 Model Exploration

The model is incrementally explored in this section in order to understand its behavior regarding changes in the hyperparameters, as well as to see the effects of the modern techniques used in the training process.

Scheduled Sampling

The consequences of the used scheduled sampling technique are examined first. Figure 6.1 visualizes the training and validation binary cross-entropy using our standard model with either *scheduled sampling* (SS) or *always sampling* (AS). The latter method means the approach to always sample from the previously generated frame, which is comparable to SS with a constant sampling probability of $p = 0$. It can be seen that there is a huge discrepancy between the training and validation error in the starting phase, where the SS-approach mainly trained on input samples taken from the ground truth. This can be explained that sampling on the ground truth in every time step of the recurrent network tremendously speeds up the convergence of the training loss, because each cell does not have to correct errors of the previous cells. In contrary, the validation loss is very bad in this phase, since the results represent the average out of ten predicted frames and the spatio-temporal decoder suddenly receives its own feature space predictions as input to predict the next frame, which is in contrast to the procedure while training at that point.

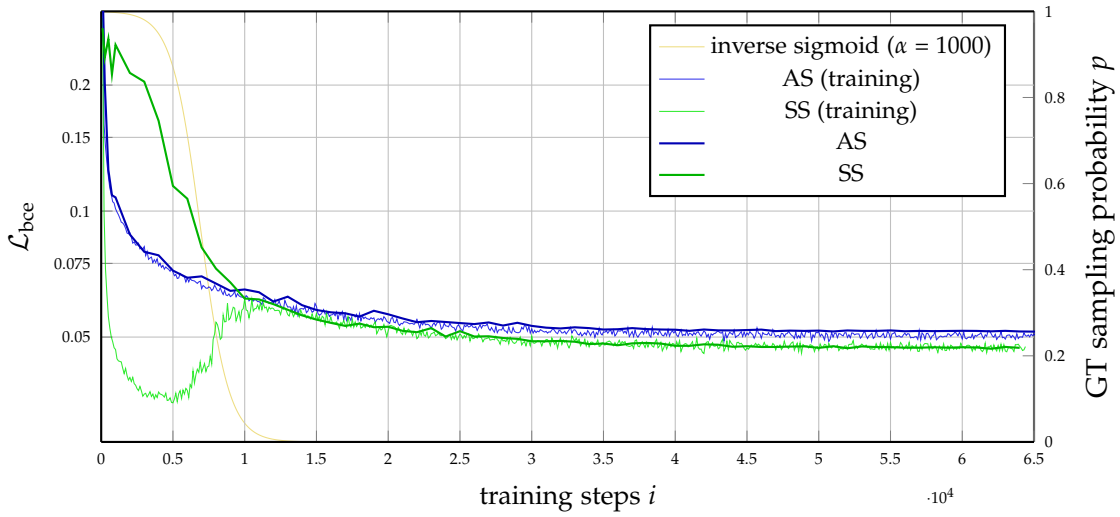
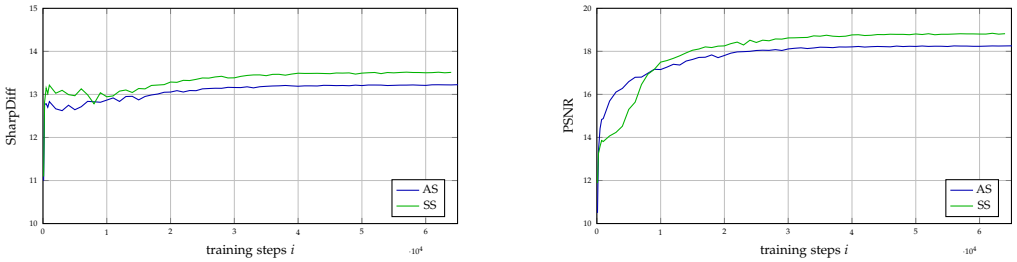


Figure 6.1: Influences of scheduled sampling regarding the training and validation error in context of recurrent networks and future frames prediction on Moving MNIST.

But the interesting insight here is that as soon as the scheduled sampling component completely changed the input behavior to inference mode, the achieved prediction performance is continuously better compared to the approach of always sampling from previous generated frames directly at the very beginning. This behavior could be explained that the SS approach introduces a form of pre-training phase, where the network can learn to predict the next frame when a perfect current frame is given. Hence, it has not to deal with errors made in the previous time step. By slowly changing this behavior to the mode as it is used during inference, it then also starts to learn a robustness against imperfect input frames. A similar behavior can be seen when comparing the results of the PSNR metric between AA and SS in Figure 6.2b. However, when we look at the sharpness difference metric in Figure 6.2a, it can be seen that the sharpness of the predictions is continuously better when scheduled sampling is used.

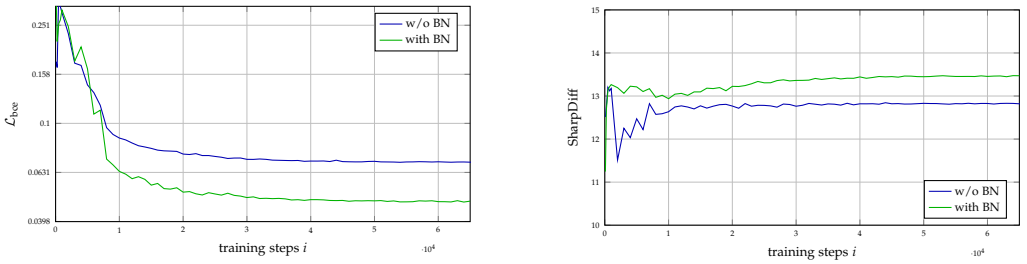


(a) (b)

Figure 6.2: Comparison of validation results based on a model with either using the scheduled sampling or the always sampling training technique.

Batch Normalization

The use of batch normalization in the spatial encoder and decoder components is investigated next. Figure 6.3 shows a clear advantage of using BN-layers. It can be argued that our model is able to profit strongly from batch normalization, because it does not only support the convolutional encoder and decoder to learn faster by compensating the internal covariate shift, but also both recurrent networks in between benefit from it. This can be reasoned by the fact that frame representations in feature space, which are produced by the spatial encoders, tend to have a more stable distribution. Consequently, the spatio-temporal encoder has fewer problems to extract useful patterns from these representations while consuming them in the course of the training process.



(a) (b)

Figure 6.3: Comparison of the validation results from two model instances where one uses batch normalization layers in the spatial encoder and decoder components.

Learning Rate Decay

The network's learning behavior using two different exponential decay rates is illustrated in Figure 6.4. In this example, the learning rate is slightly decayed after each epoch by a specified factor. Because the Moving MNIST dataset is generated on-the-fly, its total size is specified to be virtually 32,000. In this way, the learning rate is effectively decayed after each validation step[3], since a batch size of 32 is used. The network denoted with blue has a clear disadvantage, because its learning rate at the end becomes so small that it almost stops learning. However, throughout the experiments it is determined that a very low learning rate at the end of the training is beneficial regarding the image quality and finer details of the generated images.

[3] In the course of this evaluation, the model is assessed for a full epoch on the validation set every 1000$^{\text{th}}$ training step.

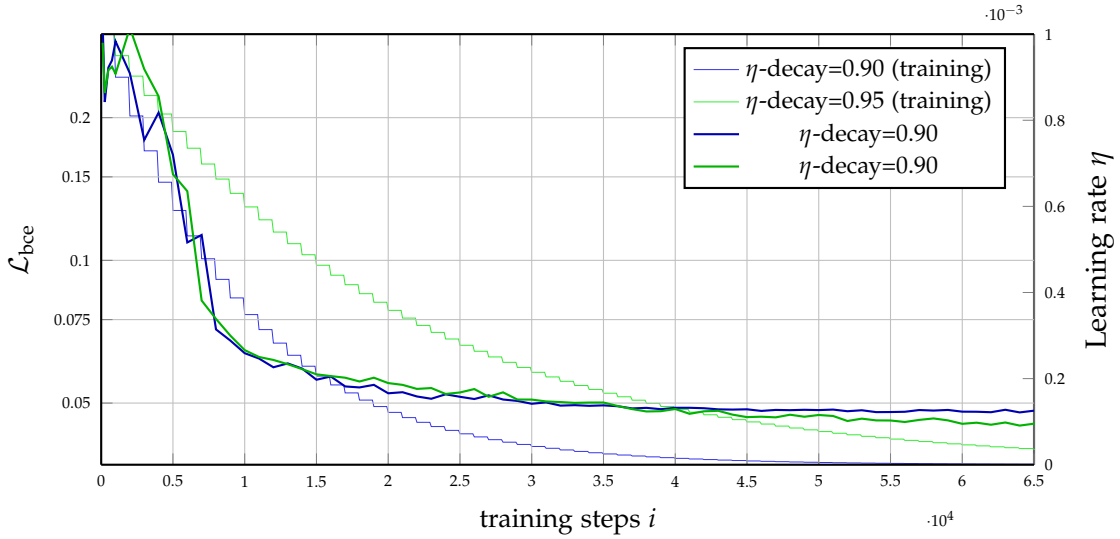
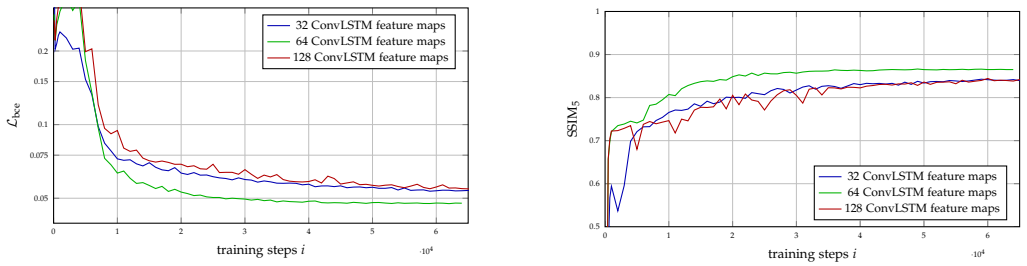


Figure 6.4: Validation losses of two identical network models using different learning rate decay rates in combination with Adam optimizer.

ConvLSTM Feature Maps

Next, the network's behavior with respect to the hidden-to-hidden feature maps in the spatio-temporal decoder and encoder components is investigated. Therefore, the number of feature maps produced by the convolutional layers in the spatial encoder is adjusted. While a simpler variant is used that produces (16, 32, 32) feature maps in each layer at first, another more complex variant is used as well which convolves (64, 128, 128) feature maps. The number of feature maps produced by the last layer is then used constantly within the ConvLSTM cells and its temporal transitions. Consequently, the performance of these two model configurations is compared against the standard model with 64 temporal feature maps in Figure 6.5.



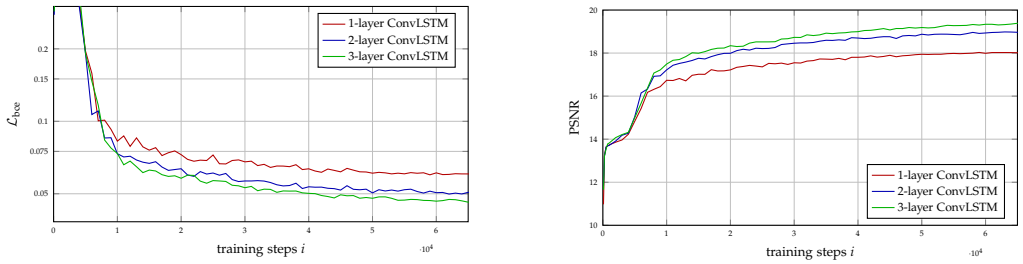
(a) (b)

Figure 6.5: Impact of the number of feature maps within the ConvLSTM encoder and decoder networks regarding overall frame prediction performance.

As expected, the network with only 32 feature maps performs worse compared to the standard setting. Nevertheless, it is very surprising that the model configuration with 128 feature maps achieves even worse results. One possible reason could be an unfavorable, random initialization of the initial model parameters. However, due to the fact that such a network configuration requires roughly the entire video memory of the GPU, as well as takes twice as long to train, the decision was taken to continue the experiments only on 64 feature maps.

ConvLSTM Layers

The number of recurrent layers has a very positive effect in order to generate future frames of better quality. The same behavior was also mentioned in one practical experiment of [SMS15, p. 6], in which they have qualitatively shown that a deeper LSTM is able to generate sharper images. Figure 6.6 shows that the improvement of adding a third layer are apparently smaller compared to the difference when a second layer is added. Unfortunately, a 4-layer network could not be tested due to memory limitations of the graphics card. Additionally, it is worthwhile noting that the number of recurrent layers has a huge impact in respect to model complexity, training time and memory requirements.

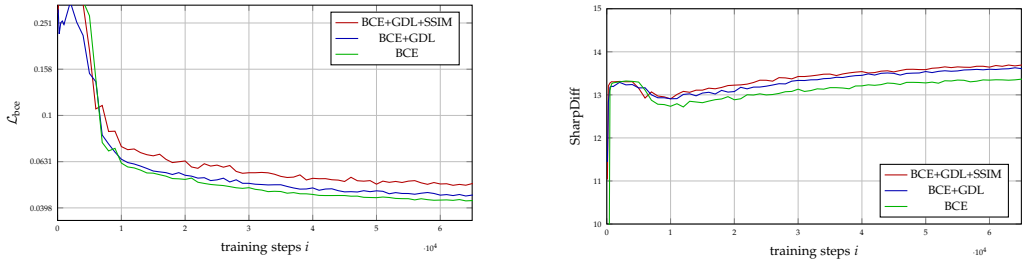


(a) (b)

Figure 6.6: Validation performance of network models with a varying number of recurrent layers. All three models are trained using the triplet loss function.

Loss Layer

As a last experiment of the test series, the impact of different loss terms is examined. Three networks with the identical standard configurations, but different objective functions have therefore been trained. Starting from a network using binary cross-entropy only, perceptual motivated loss terms like GDL and SSIM are added one after another until it ends up in the triplet loss described in equation 4.3. The validation results are depicted in Figure 6.7.



(a) (b)

Figure 6.7: Comparison of the validation loss using different objective functions.

While the network purely trained on the BCE loss function performs best regarding the binary cross-entropy on the validation set, it nevertheless performs worst according to the sharpness difference metric shown in Figure 6.7b. These results are not surprising, because network using the triplet loss function for example is putting more emphasis on other properties, in contrast to specifically optimizing it regarding to the binary cross-entropy. However, after the qualitative analysis of many generated prediction samples in the course of the experiments, it seemed that networks using the simple BCE loss produced the best results. Especially in samples where one handwritten number crosses the other number, predictions of the farer future kept slightly more stable. All in all, perceptual motivated loss terms seem to be useless or even marginally counter-productive when applied on binary Moving MNIST image frames.

6.1.2 Test Results

For the final evaluation of our model with Moving MNIST, the qualitative and quantitative performance of the best configuration is assessed on the test set. Inspired by the findings of the model exploration, as well as a rudimentary grid-search, the model for this final test ends up to use the default settings as described earlier, but with an exponential learning rate decay of $\alpha = 0.95$ and a loss layer that utilizes no perceptual motivated loss term. Three models of this network with different numbers of recurrent layers are trained for 100,000 steps.

Quantitative Results

A comparison of the best performing model with experimental results of other works is given in Table 6.1. It can be seen that the proposed model exhibits way less model parameters, especially compared to FC-LSTM approaches. Nevertheless, our ConvLSTM model is able to cut the average pixel-wise test error of the best performing competitive model in halves. The listed results are taken from several published papers. However,

the results of FC-LSTM-Combo published in [SMS15] could not be reproduced using their open-source code. After training the model for about 1,000,000 training steps, which takes almost one week, the results are not nearly as good as in the paper. Despite the fact that about 20,000 training iterations are already sufficient for the proposed model in this thesis to generate even better results.

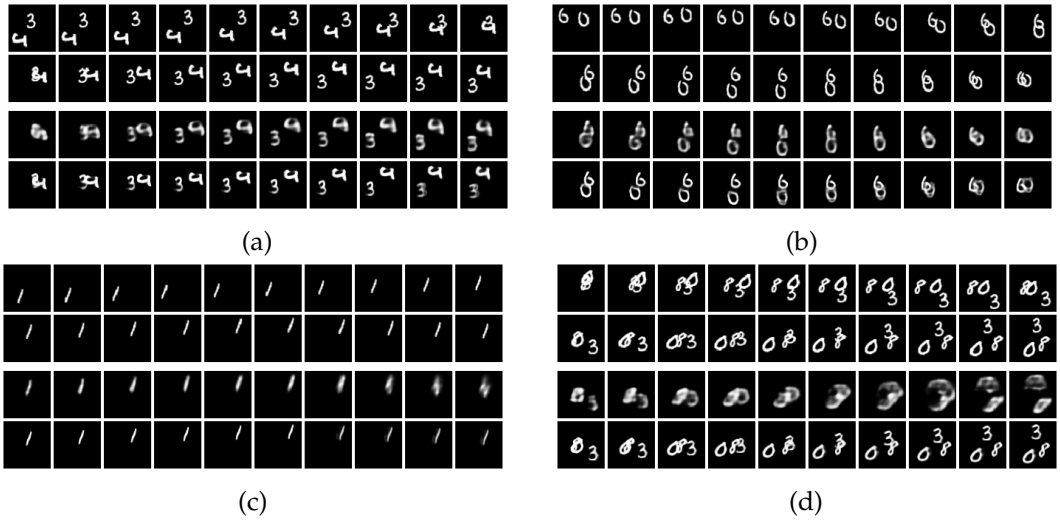
Model	**Trainable parameters**	**BCE test error**
FC-LSTM-Combo(2048-2048) [SMS15]	214,001,664	0.0855
FC-LSTM(2048-2048) [Shi+15]	142,667,776	0.1180
ConvLSTM(5/128-64-64) [Shi+15]	7,585,296	0.0896
2enc-ConvLSTM(7/45-45)[4] [PHC16]	6,132,203	0.0835
3enc-ConvLSTM-SS-3dec(5/64)	1,841,793	0.0524
3enc-ConvLSTM-SS-3dec(5/64-64)	3,570,817	0.0414
3enc-ConvLSTM-SS-3dec(5/64-64-64)	5,299,841	**0.0407**

Table 6.1: Comparison with other networks on Moving MNIST. The numbers in brackets identify the *amount of hidden units* per layer in case of FC-LSTMs, and for ConvLSTMs the *hidden-to-hidden kernel size* followed by the *feature maps per layer* are listed. Further, *3enc* denotes tree convolutional layers in the spatial encoder. THe model of this thesis is called *3enc-ConvLSTM-SS-3dec*.

Qualitative Results

Next, several future frame prediction samples are qualitatively compared with results of other models presented earlier in chapter 3. To that end, all Moving MNIST examples that are published in other works are taken and the network is applied on exactly the same input data. The contrast against an FC-LSTM approach can be seen in Figures 6.8a or 6.8a, as well as the comparison with another ConvLSTM implementation in Figure 6.9a. The outcomes of our two-layer model are slightly sharper and much brighter in all three examples. However, the digits 3 and 0 in Figure 6.8 show some visible artifacts at the end of the predicted future sequence.

Further predictions of our model, which have been randomly chosen from the Moving MNIST test set, can be found in Figure A.3 in the appendix. While most generated results look quite similar to the ground truth future, the sample in Figure A.3b clearly shows that the model still has its issues to preserve details when both digits overlap each other over a longer period.

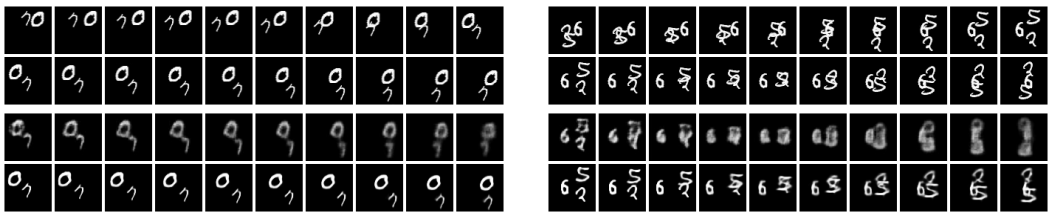


(a) (b) (c) (d)

Figure 6.8: Qualitative comparison with FC-LSTM-Combo(2048-2048) [SMS15]. Experiments in (c) and (d) are using a different number of digits than the network was trained for. From top to bottom: inputs sequence; ground truth target sequence; predictions of FC-LSTM-Combo; predictions of our 2-layer model.

Out-of-Domain Results

Additionally, the model's is verified regarding its ability to deal with data that is outside the domain it has been trained for. Therefore, several Moving MNIST sequences that use one or three digits instead of two are taken and compared the forecasted results with predictions generated by other competitive models. At this point, it is to highlight that no cherry-picking is performed in this experiment, because the exact same sequences are used that were selected in related works. In a manner of speaking, this test scenario uses specific input sequences which probably worked better in other approaches.



(a) (b)

Figure 6.9: Qualitative comparison with ConvLSTM(5/128-64-64 [Shi+15]. The experiment in (b) uses more digits than the network was trained for. From top to bottom: inputs sequence; ground truth target sequence; prediction results of the competitive model; predictions of our 2-layer ConvLSTM.

A remarkable difference is visible when the outcomes of our model are qualitatively compared with predictions of existing approaches. In case of the samples using three digits in Figures 6.8d and 6.9b, the structure of each digit stays more stable and does not get washed-out, even after a digit has been overlapped by both others. Moreover, it does not try to merge two numbers in order to end up with the same number of digits like the networks have been trained for. A similar behavior can be seen in Figure 6.8c, in which only a single digit is used. In that scenario, the other network clearly tries to hallucinate a second flying number at the end of the future sequence. This misbehavior is not visible in the outputs of the proposed model.

Further out-of-domain results can be found in Figure A.4 in the appendix. It shows the scenario where the model tries to predict a much longer sequence than it has been trained for. Such a scenario is simple to realize with this model due to its recurrent encoder-decoder architecture. Therefore, the spatio-temporal decoder component is unrolled for 30 time steps in order to verify the long-term stability of predicted frames. It can be seen that the network is able to continuously produce acceptable results, with the exception of the last five frames in Figure A.4a. From about the 30^{th} predicted future frame, the model then starts to produce very noisy outcomes in almost every case.

6.2 Experiments on MsPacman

The network model is applied on the MsPacman dataset in the second experiment to see how it behaves with data of increased complexity. The effects of perceptual motivated loss terms are investigated more deeply in this dataset, because we strongly belief that their weaknesses in Section 6.1 are caused by the artificiality of binary image data.

6.2.1 Hyperparameter Tuning

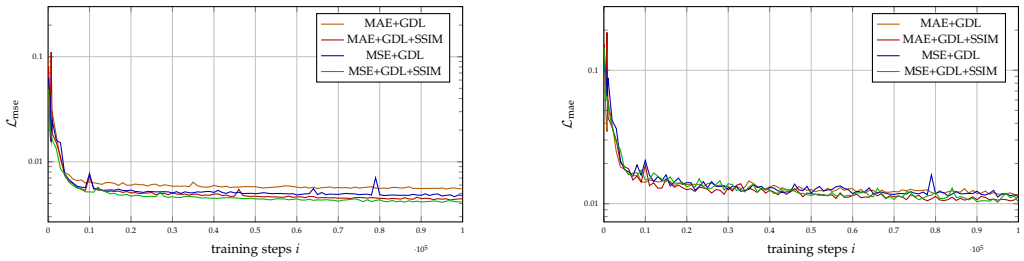
In order to find good hyperparameters for the network model, the best performing model of the previous experiments on Moving MNIST is used as a starting point, but with the four following modifications. First, the three layers of the spatial encoder and decoder components use strides of $s_i = (1, 2, 1)$ instead $s_i = (2, 1, 2)$, where i denotes the index of the convolutional layer, in order to end up with a feature space representation of the same shape as before. This change is caused by the fact that the model is trained on smaller 32×32 patches. Second, a *tanh* activation function is used in the output layer so that each generated frame is within the valid scale. Third, a learning rate decay of $\alpha = 0.95$ per training epoch is used by default. And lastly, the triplet loss function of Section 4.2.1 with MAE as the main loss function is used as a starting point in this

evaluation, because previous works have shown its advantages in image processing tasks compared to squared error.

Several networks are trained with different numbers of recurrent layers, varying initial learning rates $\eta \in \{0.0005, 0.001, 0.005\}$, as well as different weight decays $\lambda \in \{1e^{-6}, 1e^{-5}, 1e^{-4}\}$. While a clear improvement between the single layer and two-layer ConvLSTM model can not be observed, there is no clear benefit from using a third layer. The differences in results when performing variations in learning rate and weight decay are also marginal. Eventually, it ended up with an initial learning rate of $\eta = 0.0005$ and the default weight decay of $\lambda = 1e^{-5}$ is continued to be used. Further, the two-layer model is used for the main reason to save training time.

Loss Layer

Even though network's performance can only marginally be improved by performing variations in several hyperparameters, modifications in the objective function are indeed able to cause more significant differences in both quantitative and qualitative results. As a consequence, further investigation regarding different loss term combinations are performed next. Several validation results are therefore illustrated in Figure 6.10.

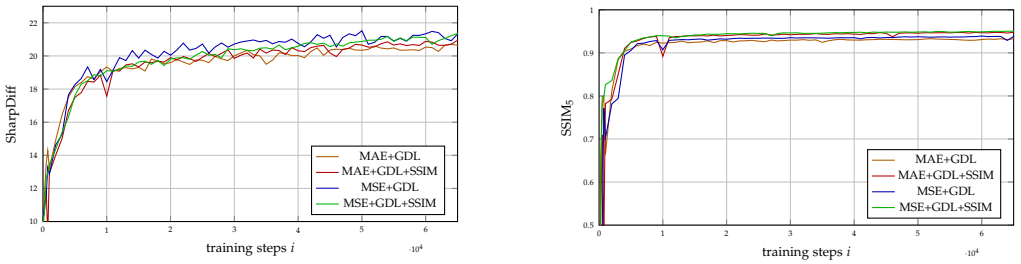


(a) (b)

Figure 6.10: Comparison of validation results on MsPacman using a 2-layer network with varying loss function combinations. All models have been trained given an input sequence of 8 patches to predict the next 8 frames. The loss values that are shown in the graphs represent the per-pixel error.

Both diagrams show that the use of the triplet loss function leads to better results regarding squared or absolute error. The same is true for the SSIM index, as it can be seen in Figure 6.11b. But overall, the MSE loss function seams to result in a slightly better network performance. Special emphasis should be placed on the fact that for example a network that is trained using the triplet loss MSE+GDL+SSIM performs even better than a model trained with MAE+GDL regarding the MAE loss on the validation set, even though the latter network is explicitly optimized for this objective. Additionally, according to the sharpness metric demonstrated in Figure 6.11a, the

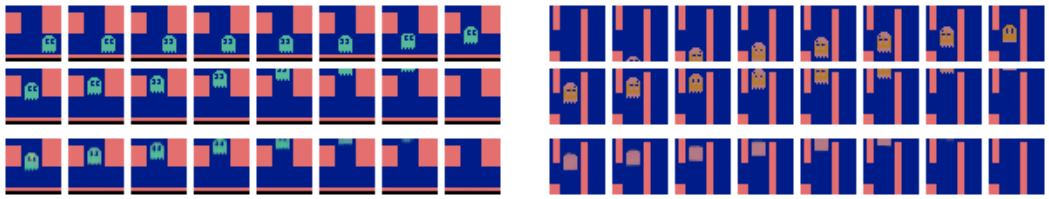
predicted frames of both networks that utilize the mean squared error as its main loss function feature sharper edges.



(a) (b)

Figure 6.11: Image metrics computed after each training epoch on the MsPacman dataset. All four networks use a 2-layer ConvLSTM and the same hyperparameters, but a different objective function.

As a final qualitative comparison regarding the differences of these objective functions, two samples of the best and worst performing models are presented in Figure 6.12. While the network that has generated Figure 6.12a is trained using the MSE-based triplet loss function, so is the other model instance, which has predicted the frames of the third column in Figure 6.12b, trained on a combined MAE+GDL function. By comparing both predictions of this similar scene, the ghost of the latter network does visibly lose its finer details. More precisely, it is predicted as a simple blob without its eyes.



(a) (b)

Figure 6.12: Two predictions showing a similar scene of a moving ghost. The sequence in (a) is predicted using a network trained with the MSE-based triplet loss function, whereas the network in (b) is trained on MAE+GDL loss.

6.2.2 Test Results

Next, the test results of the best performing two-layer model are presented. This model is configured as described earlier in this section. In the qualitative assessments, the propsed network takes advantage from the triplet loss function using MSE as its main loss term. It produces the most promising results while the behavior of the model was

explored on this dataset. It is important to note that the test set is not filtering out random crops that show just few or even no movement at all. That is why it is to expect that the average test results can be much better compared to the validation result, since it is quite easy for the network to predict scenes that only contain the maze's static background with no movement at all.

Quantitative Results

The image metric results of our network model using the discussed loss function combinations is shown in Table 6.2. Unfortunately, there is no related project that has published any future frame prediction results on this dataset yet. However, this dataset is indeed used in an independent implementation of [MCL16]. The result of some experiments with it are added for the sake of completion. However, these results are unfortunately hard to compare with ours, because the metrics are calculated on that parts within a frame where motion is actually happening. Moreover, it predicts only one frame given a sequence of four.

As it is the case for the validation set, the MSE-based triplet loss function produces the best results in all similarity metrics, regardless of whether it is about the first predicted frame, or the average of the whole future sequence. However, by looking at the sharpness results, the particular model instance which utilizes no SSIM term in its objective function is able to produce sharper results according to the metric. This is not surprising, because the GDL term is weighted more in that network, which is the central objective of this metric.

Loss function	**1st frame**			**mean of 8 frames**		
	Similarity		Sharpness	Similarity		Sharpness
	PSNR	$SSIM_5$	SharpDiff	PSNR	$SSIM_5$	SharpDiff
Adv+GDL [Coo16]	25.3168	-	17.2725	-	-	-
MAE+GDL	47.5462	0.9854	27.0217	48.6902	0.9817	27.8182
MAE+GDL+SSIM	44.8974	0.9917	26.9131	44.6555	0.9856	27.9516
MSE+GDL	50.5197	0.9893	**30.0961**	50.1164	0.9833	**29.9851**
MSE+GDL+SSIM	**50.4655**	**0.9925**	28.5023	**50.2746**	**0.9862**	28.7008

Table 6.2: Metric results of produced patch sequences using a 2-layer model on MsPacman's test split. The achieved similarity and sharpness of the same models using different loss layers are given for both the first frame only and the average of the whole generated future sequence. The results of the other approach in the first row are not comparable to ours due to a different evaluation procedure.

Another interesting fact is that the differences between the image metric results of the first predicted frame and the average of all predicted frames are very marginal. This might be due to the reason that the bigger part of each patch is just static, noise-free

content. Hence, the actual interesting parts of each frame that include motion have only little effect on the metric results. Furthermore, it is surprising that the result of the sharpness metric for the first predicted frame is slightly worse compared to the metric's mean over the whole predicted sequence.

On the other side, the pixel-wise absolute and squared error results in Table 6.3 appear more plausible. Here, the differences between the first frame and the average of the sequence are significantly larger. But one fact remains unchanged: the networks trained on the MSE-based triplet loss function are able to produce the best prediction results. Furthermore, it is interesting to see that both networks that utilize a squared error loss term as their main objective are able to outperform the other model instances regarding the MAE error. Despite the fact that the other two networks are explicitly optimized for this objective. This superiority could be reasoned by the characteristics of the MsPacman dataset.

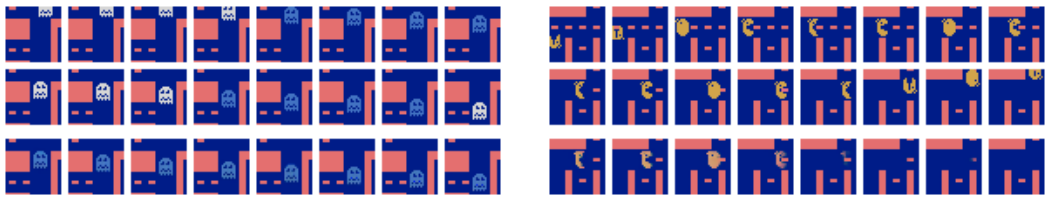
Loss function	**1st frame**		**mean of 8 frames**	
	MAE	MSE	MAE	MSE
MAE+GDL	0.0033	0.0009	0.0041	0.0018
MAE+GDL+SSIM	0.0031	**0.0005**	0.0047	0.0015
MSE+GDL	0.0025	0.0006	0.0040	0.0016
MSE+GDL+SSIM	**0.0023**	**0.0005**	**0.0037**	**0.0014**

Table 6.3: Absolute and squared error test results on MsPacman dataset using varying loss layers in our 2-layer network.

Qualitative Results

For a qualitative evaluation of the network, a detailed look at some generated future sequences is taken. Starting the clip in Figure 6.13a, the motion of the ghost is predicted very precisely. The fine details of the ghost's eyes and zigzag mouth are preserved even until the eighth frame. However, the dynamics of the blue and white blinking effect is not continued in the predicted sequence.

As a second example, the prediction result shown in Figure 6.13b is considered. On the one side, pacman's dynamics are correctly continued in first four generated frames. It correctly moves forward, performs chewing and eats the orange dot, but with some delay. On the other side, the game character suddenly disappears when it arrives at the second crossing. Such a behavior is actually quite often observed in many other samples, too. As a result, this indicates that the network has its issues to decide what direction the moving objects will take within the maze. This assumption can be further confirmed by some other observations, such as that objects reaching a crossing are either affected by a loss of opacity, split up into multiple directions, disappear complete as shown in the demonstrated example, or in best case it chooses any random direction.



(a) (b)

Figure 6.13: Two random prediction samples on MsPacman that contain at least some motion. From top to bottom: ground truth input and output sequence; predicted sequence of the 2-layer network.

Such disappearance effects take rarely place in corners where there is beside turning around only one other option.

More predictions on small patches can be found in Figure A.5 in the appendix. It is worth noting that still-standing motion is particularly well captured and continued by the network. As an example, pacman's chewing animation in Figure A.5a is kept up in all predicted frames, so that it looks almost identical at the first glance. The only noticeable difference here is that pacman temporarily changes its direction in the fifth frame for just a single time step in the ground truth future.

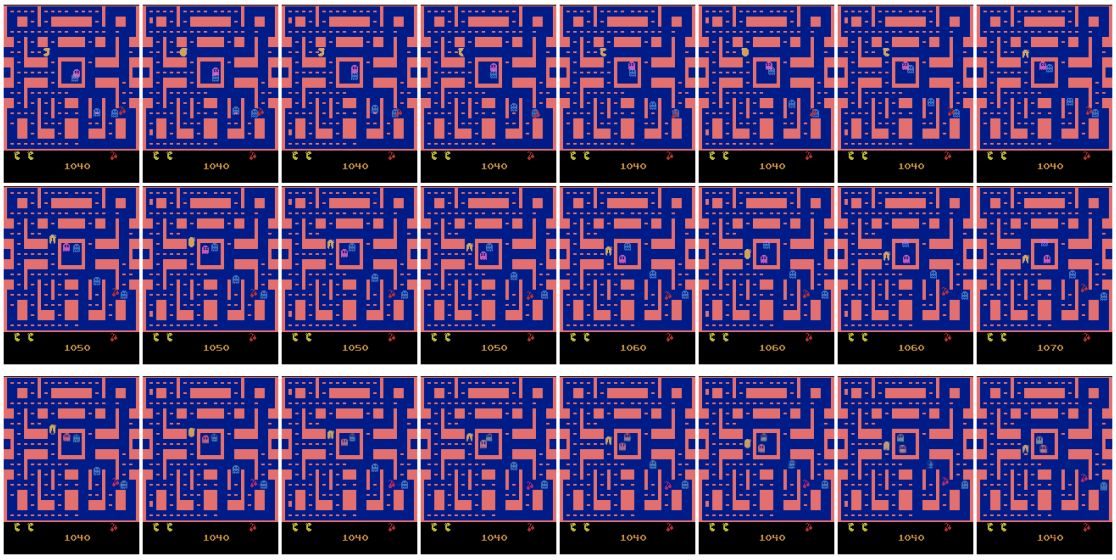


Figure 6.14: Prediction sample of a full game scene using our 2-layer network, which is trained on small 32×32 patches only. This is possible due to our FCN approach. From top to bottom: ground truth input and output sequence; predicted game sequence.

As explained in section 2.2.5, the fully-convolutional approach of our model comes with the advantage that it can be applied on input images of larger size. Therefore, a prediction sample of a sequence showing a full game scene is presented in Figure 6.14.

It shows that future frames can indeed be forecasted for the full images with acceptable results as well. For instance, the blinking effect of the large dots in the corners of the maze and the trajectory of the red cherry is predicted almost perfectly. Furthermore, the motion of the game characters is continued quite well, with the two exceptions that one ghost disappears and both ghosts in the center slightly change their color. A second result containing the full maze can be found in Figure A.6. In that example, attention should be paid to the blue ghost in the center, who is correctly predicted to leave the cave instead of going for another spin.

Finally, the stability of the generated future frames is investigated by performing predictions on longer sequences. As it can be seen in Figure A.7, some artifacts become visible starting from the 14^{th} forecasted frame. While all moving game characters gradually disappear, the red cherry object is predicted very precisely until the end of the lengthened sequence.

6.3 Experiments on UCF-101

The last experiment investigates the performance of the proposed neural network model on natural videos. More precisely speaking, various network instances are trained on 32×32 patches from the UCF-101 dataset, which is further described in Section 5.3. Due to the fact that clips from this dataset show natural scenes with a lot of movement and noise, it can be expected that the inclusion of perceptual motivated bias terms in the loss function is finally able to unfold its full effect. For that reason, the examination of different loss function combinations from the second experiment on MsPacman is repeated for this dataset in the hope that it reveals more significant improvements.

6.3.1 Hyperparameter Tuning

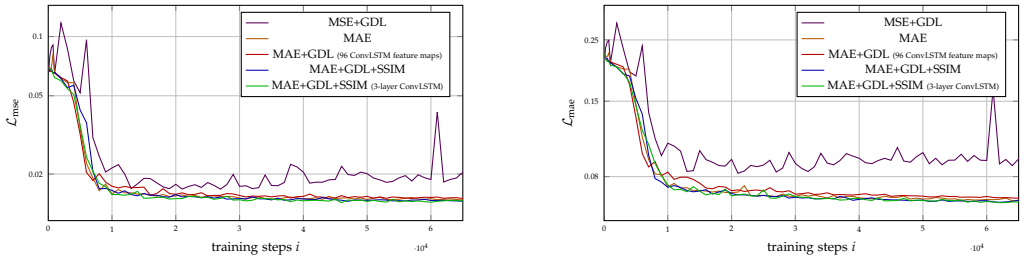
The search for good hyperparameters and the resulting settings is very similar to the procedure of Section 6.2.1. Several model instances are trained with varying learning rates, but limit the regularization coefficients to a smaller search space in order to save time, in particular $\lambda = \{1e^{-6}, 1e^{-5}\}$. Further, no single-layer ConvLSTM configuration is tested, argued by the insights of both previous experiments. In detail, most trials are performed on two-layer ConvLSTM set-ups, and finally test the same configuration using a three-layer ConvLSTM. Even though MSE as the main objective term of the triplet loss function has shown to deliver the best results on MsPacman, the absolute error as the main loss term is used nevertheless. This is due to its positive results in context of natural images, as already mentioned in Section 2.6.1 and Chapter 3.

The grid-search ends up with the same model configuration as in the last experiment. But this time, the differences between using different initial learning rates become

more noticeable, with lower learning rates in the advantage. Hence, the following investigations are based on an exponential decaying learning rate starting from $\eta = 0.0005$ with a weight decay of $\lambda = 1e^{-5}$.

Loss Layer

The absolute and squared pixel-wise prediction errors on the validation set using various network configurations are illustrated in Figure 6.15. It can be clearly seen that the use of MSE produces less accurate predictions compared to all other loss function combinations. While the results of all other configurations are very close together, the error of the MAE-based triplet loss function is slightly in advantage. Even though the diagram presents its results of the trained three-layer network instance, the same is true for the two-layer variants.



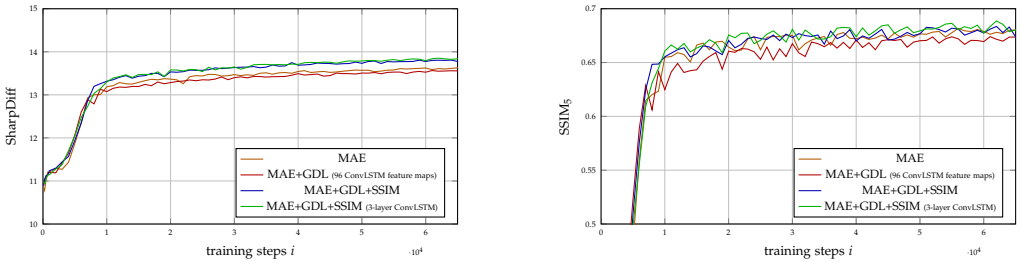
(a) (b)

Figure 6.15: Comparison of validation results on UCF-101 using similar network configurations with varying loss functions. All models have been trained given an input sequence of 10 patches of size 32×32 to predict the next 10 frames. The showen loss values represent the per-pixel error. All models except one are using the vanilla configuration with 64 filters within the ConvLSTM cells.

The advantages of a network that is trained using the triplet loss function become more visible when the differences in sharpness are considered. At least the sharpness metric in Figure 6.16a quantifies such a hypothesis.

6.3.2 Test Results

In the following, the test results of different network configurations are presented in a quantitative and qualitative analysis. In analogy to the previous experiments on MsPacman, it is worth mentioning that the video patches from the test set are not filtered in case they show very less movement.



(a) (b)

Figure 6.16: Computed image metrics computed on the UCF-101 dataset. All four networks use the same hyperparameter settings, but differ in the number of recurrent layers, as well as the used objective function.

Quantitative Results

Starting with the qualitative test results, Table 6.4 compares several configurations of our model with outcomes from related works on various image similarity or sharpness metrics. It demonstrates test results of the first predicted frame, as well as the average metric values of the forecasted 10 frames long sequence. However, the results of our model are not really comparable with [MCL16], since they calculate the metrics only on specific regions of the image, where the optical flow exceeds a specified threshold.

Loss function	**1st frame**			**mean of 10 frames**		
	Similarity		Sharpness	Similarity		Sharpness
	PSNR	SSIM		PSNR	SSIM	
single-scale ℓ_2 [MCL16]	26.5	0.84	24.7	-	-	-
multi-scale ℓ_1 [MCL16]	28.7	0.88	24.8	-	-	-
multi-scale GDL+ℓ_1 [MCL16]	29.4	0.90	25.0	-	-	-
multi-scale Adv+GDL [MCL16]	31.5	0.91	25.4	-	-	-
MSE+GDL	15.9652	0.7112	13.6947	22.5218	0.7016	14.5219
MAE	30.8241	0.9043	17.0282	25.9873	0.7653	15.4952
$MAE+GDL_{96}$	29.4092	**0.9047**	16.7509	25.8577	0.7686	15.4089
MAE+GDL+SSIM	**31.3144**	0.7668	**17.3785**	**26.6385**	0.7668	15.7363
MAE+GDL+SSIM 3layers	29.4000	0.9031	17.2316	26.2130	0.7679	**15.7283**

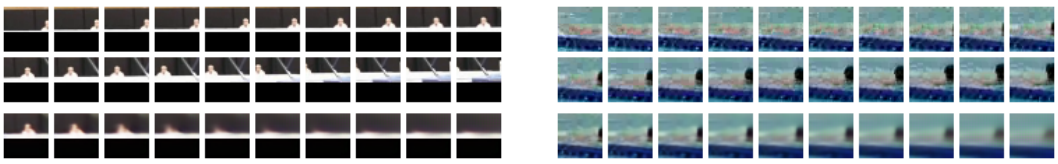
Table 6.4: Metric results of predicted patch sequences using our approach on the UCF-101 test split. The achieved similarity and sharpness results by using different loss layers are given for both the first frame only and the mean of the generated sequence. All our models are trained for the same amount of time, in detail the 2-layer networks for 100,000 and the 2-layer network for 68,000 iterations. The outcomes of other approaches are not comparable to the propsed model in rows 5-9, because the mertic results of the given other soltion takes only moving areas of the images into account.

Analyzing the achieved test results listed in Table 6.4, it can be seen that the use of absolute error as the main objective function produces much better results in all cases in comparison to squared error. This has already been observed during the search for the optimal hyperparameters of the model. However, it might be doubted that the metric results of the first predicted frame using the MSE+GDL loss are actually worse compared to the mean of the entire forecasted sequence. A quick look at Figure 6.18 (row 15, columns 1) indicates that this network indeed has its issues with the brightness of the first generated image.

While the use of the third recurrent layer does not seem to be advantageous, it has to be noted that this network is trained for fewer iterations. Thus, the results represent different networks that are trained for roughly the same amount of time. Although the two-layer network using the triplet loss function is slightly outperforming the other models, its SSIM index result is far beyond all three other MAE-based networks.

Qualitative Results

For a qualitative evaluation of the model, many prediction results are analyzed regarding the perceptual similarity to the ground truth future. Two selected sequences are therefore depicted in Figure 6.17. Both predictions are visibly affected by a blur effect the further the network has to look into the future. Nevertheless, a slight notion of movement is noticeable at the beginning of all presented clips. Especially the swimmer in Figure 6.17b is continued quite precisely, although he enters the scene just at the very end of the input sequence. Further examples of predicted patches can be found in Figure A.8 in the appendix.



(a) (b)

Figure 6.17: Two prediction examples using our 2-layer ConvLSTM model. From top to bottom: Input and ground truth frames; prediction results.

Fortunately, the UCF-101 dataset i applied in many related studies as well. It is therefore possible to do a more extensive comparison regarding outcomes of different approaches. Results from an FC-LSTM encoder-decoder network, different convolutional with or without adversarial training and various combinations of loss functions are collected in Figure 6.18. When comparing the predicted sequences of non-adversarial networks with the results of the proposed model in the last group at the bottom of the graphic, it can be seen that their outcomes are generally blurrier. However, a clear motion is

also not discernible in the forecasted frames of our model. Although the predictions of the adversarial networks do not look close to the ground truth on the long term, they indeed remain very realistic.

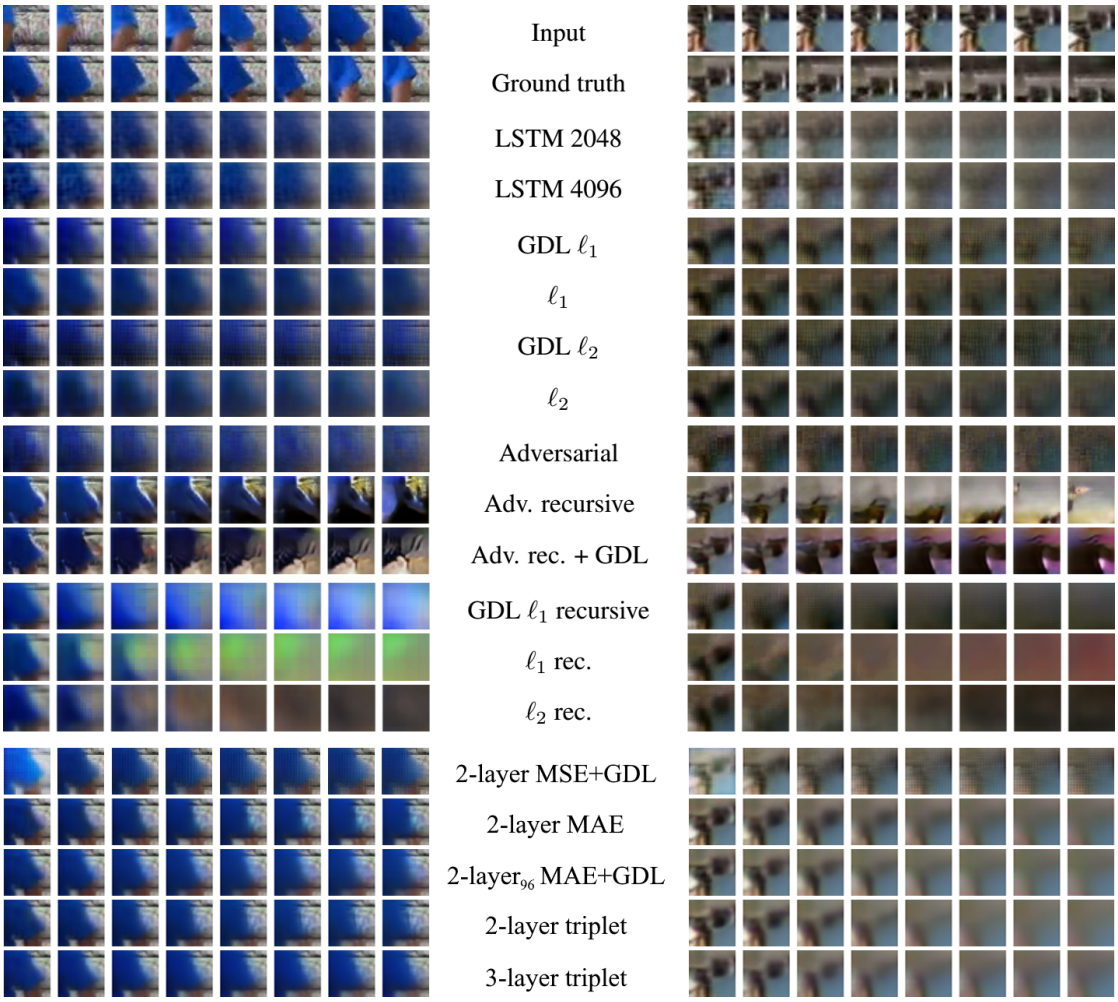


Figure 6.18: Qualitative comparison to different approaches, including various objective functions. From top to bottom: input and ground truth target frames; FC-LSTM results; simple encoder-decoder CNN that predicts 8 future frames given 8 input patches; the same CNN but with adversarial training; simple encoder-decoder CNN that recursively predicts a single frame give 4 patches; our model in its vanilla configuration, except 2-layer$_{96}$ where 96 feature maps are used in each ConvLSTM cell. (Based on [MCL16, p. 13])

Additional long-term predictions are performed in order to assess the networks stability regarding its generated future frames. One example sequence is demonstrated in Figure 6.19, in which the network predicts twice as many frames as it has been trained for. When the whole sequence from the first until the very last frame is considered, a tendency of the head to move slowly downwards can indeed be observed. However,

the predictions are getting more and more burry as expected. For a second long-term example of a soccer player, see Figure A.9 in the appendix. It reveals that the network has its issues to predict the accurate future of small or thin details of the content that move quite fast. In this case, the player's legs tend to vanish until the end of the sequence. The swift disappearance of fast moving objects can be observed in many other examples as well. As a consequence, this might indicate that the used hidden-to-hidden kernel size of 5×5 within the ConvLSTM cells, and/or the kernels of the other convolutional layers, are too small to capture such motion.

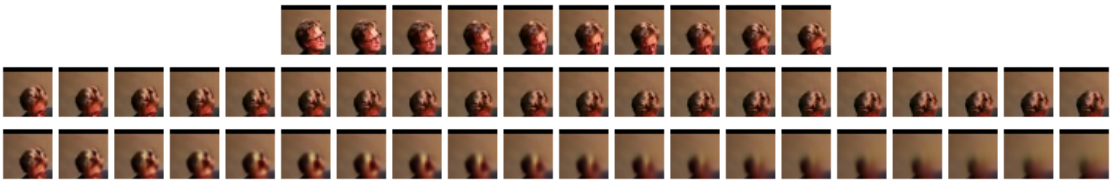
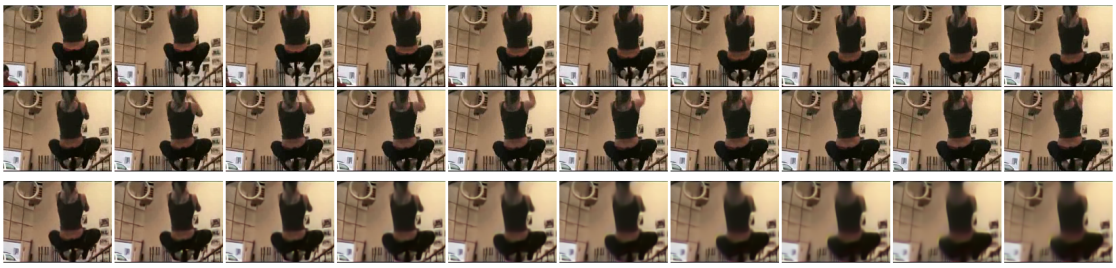
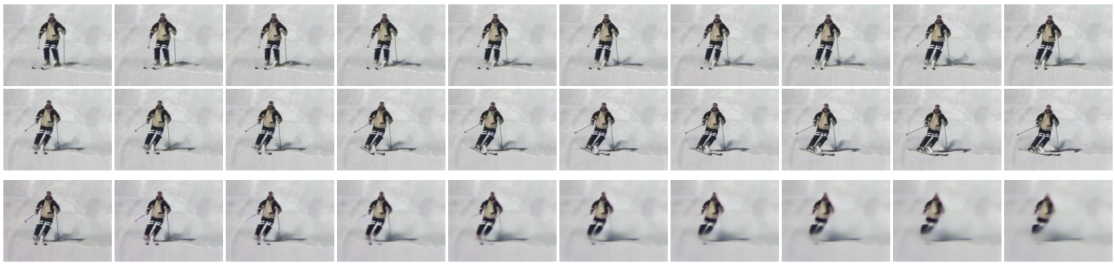


Figure 6.19: Two out-of-domain prediction examples where the network predicted 20 patches, while being trained to predict only 10 frames. From top to bottom: Input and ground truth frames; prediction results from our 2-layer ConvLSTM model.

Finally, the probably most interesting test scenario is investigated: future frame predictions using the full-size video frames. Two examples concerning this matter are depicted in Figure 6.20, but more examples can be found in the appendix in Figure A.10. We can see that the quality of results is similar to the previous scenarios. In almost every sample, static background is predicted very precisely. Even fine detail of the background can be preserved until the end of the predicted sequence, such as the wall's tile pattern in Figure 6.20a. Furthermore, some few examples are able to capture more complex dynamics of the scene. For instance, forecasting the skier's loping position in Figure 6.20b.



(a)



(b)

Figure 6.20: Prediction samples of full video frames, enabled due to our fully-convolutional approach. They are generated using a 2-layer network that is trained on smaller 32×32 patches. From top to bottom: ground truth input and output sequence; predicted future of the video.

7 Contribution

Before concluding this thesis, we would like to present a side contribution that arised in parallel to this thesis. When the implementation of this final project has started, the TensorFlow library for machine intelligence had just published its second public release with version 0.7. Thus, there has not been that much experience and best practices around with TensorFlow, as well as its API is very low-level for several use cases even today. As a result, there has been the desire to create a reusable library to reduce boilerplate code of TensorFlow based projects, as well as to retain best practices of existing examples and also the lessons learned from this thesis. A second idea has been that future theses or other deep learning projects of the *Computer Vision Group*[1] at TUM might benefit from such a library. However, this project has grown larger and larger over time and ended up in a powerful high-level framework, that has been developed independently from other high-level APIs for TensorFlow like *TF-Slim*[2] or *Keras*[3]. Ultimately, about 99% of the overall code of this thesis has been transferred into this framework, consistently with having abstraction and reusability in mind.

7.1 A High-Level Framework for TensorFlow

In this section, the design goals and key features of the TensorLight[4] framework for TensorFlow based projects are being presented. Additionally, the main principal architecture is visualized and its usage is demonstrated in a short example. We make the code of the project available to the research community under the MIT license.

7.1.1 Guiding Principles

The TensorLight framework is developed under its four core principles, namely *simplicity*, *compactness*, *standardization* and *superiority*. These goals are briefly described in the following list:

[1] Computer Vision Group at Technische Universität München: `https://vision.in.tum.de/`
[2] TFLearn - Deep learning library featuring a higher-level API for TensorFlow: `http://tflearn.org/`
[3] Keras - Deep learning library for Theano and TensorFlow: `https://keras.io/`
[4] TensorLight - A lightweight, high-level framework for TensorFlow: `https://github.com/bsautermeister/tensorlight`

- **Simplicity:** Straight-forward to use for anybody who has already worked with TensorFlow. Especially, no further learning is required regarding how to define a model's graph definition.
- **Compactness:** Reduce boilerplate code, while keeping the transparency and flexibility of TensorFlow.
- **Standardization:** Provide a standard way in respect to the implementation of models and datasets in order to save time. Further, it automates the whole training and validation process, but also provides hooks to maintain customizability.
- **Superiority:** Enable advanced features that are not included in the TensorFlow API, as well as retain its full functionality.

7.1.2 Key Features

To highlight the advanced features of TensorLight, an incomplete list of the ten main functionalities is provided that are not shipped with TensorFlow out-of-the-box. Some of them might even be missing in other high-level APIs. These include:

- Transparent lifecycle management of the session and graph definition.
- Abstraction of models and datasets to provide a reusable plug-and-play support.
- Effortless support to train a model symmetrically on multiple GPUs, as well as prevent TensorFlow to allocate memory on other GPU devices of the cluster.
- Train or evaluate a model with a single line of code.
- Abstracted, runtime-exchangeable input pipelines which either use the simple feeding mechanism with NumPy arrays or even multi-threaded input queues.
- Automatic saving and loading of hyperparameters as JSON to simplify the evaluation management of numerous trainings.
- Ready-to-use loss functions and metrics, even with latest advances for perceptual motivated image similarity assessment.
- Extended recurrent functions to enable scheduled sampling, as well as an implementation of a ConvLSTM cell.
- Automatic creation of periodic checkpoints and TensorBoard summaries.
- Ability to work with other higher-level libraries hand in hand, such as *tf.contrib* or *TF-slim*.

7.1.3 Architecture

From an architectural perspective, the framework can be split into three main components. First, a collection of *utility functions* that are unrelated to machine learning. Examples are functions to download and extract datasets, to process images and videos, or to generate animated GIFs and videos from a data array, to name just a few. Second, the *high-level library* which builds on top of TensorFlow. It includes several modules that either provide a simple access to functionality that it repeatedly required when developing deep learning applications, or features that are not included in TensorFlow yet. For instance, it handles the creation of weight and bias variables internally, offers a bunch of ready-to-use loss and initialization functions, or comes with some advanced visualization features to display feature maps or output images directly in an IPython Notebook. Third, an *abstraction layer* to simplify the overall lifecycle, to generalize the definition of a model graphs, as well as to enable a reusable and consistent access to datasets. Figure 7.1 illustrates the overall architecture of TensorLight.

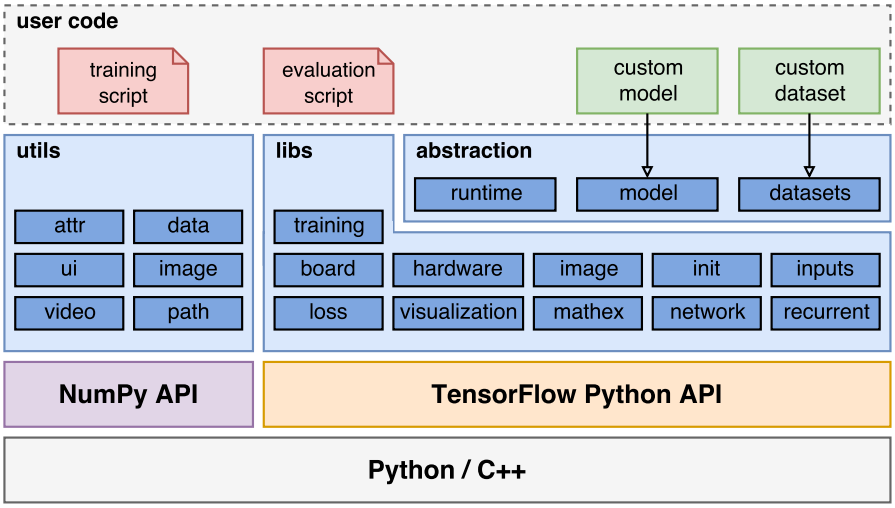


Figure 7.1: Architecture diagram of the TensorLight framework and its modules. The user program can take advantage of the provided abstraction layer, but also use the library and utility functions standalone.

The user program can either exploit the high-level library and the provided utility functions for his existing projects, or take advantage from TensorLight's abstraction layes while creating new deep learning applications. The latter enables to radically reduce the amount of code that has to be written for training or evaluating the model. This is realized by encapsulating the lifecycle of TensorFlow's session, graph, summary-writer or checkpoint-saver, as well as the entire training or evaluation loop within a *runtime module*.

7.1.4 Example

This short example of a convolutional autoencoder, which reconstructs the entire input sequence of Moving MNIST, emphasizes the simplicity of deep learning applications implemented with TensorLight. At first, the model has to be defined. This is realized by implementing a class that derives from `tl.model.AbstractModel`. This class provides a standard interface to define the graph structure, the loss layer, as well as how this model has to be evaluated by the framework. The example's model implementation can be found in Figure A.14 in the appendix. In this case, the model is trained using binary cross-entropy. Moreover, image similarity metrics like PSNR, sharpness difference and SSIM are going be calculated and visualized in TensorBoard for every validation iteration in addition to the error value of the objective function.

Training

The program code of an entire training process is depicted in Figure 7.2. Within the context of a `tt.core.AbstractRuntime` class instance, the model, optimizer and datasets are registered with its parameters. Each runtime and dataset instance has a path-parameter in order to specify the root directory of the training outputs and meta files, as well as to reuse downloaded and preprocessed files across multiple processes. Afterwards, the computation graph is built and the training loop is started for 100 epochs. Since the example tries to reconstruct the inputs, one can set `autoencoder=True` in order to redirect the target pipeline of the dataset to the input data. Additionally, each batch of size 256 is distributed across two GPUs to speed up training.

```
import [...]
with MultiGpuRuntime("/tmp/train", gpu_devices=[0, 1]) as rt:
  rt.register_model(ConvAutoencoderModel(5e-4))
  rt.register_optimizer(Optimizer("adam", initial_lr=0.001,
                                  step_interval=1000, rate=0.95))
  rt.register_datasets(MovingMNISTTrainDataset("/tmp/data", as_binary=True,
                                               input_shape=[10, 64, 64, 1]),
                       MovingMNISTValidDataset("/tmp/data/", as_binary=True,
                                               input_shape=[10, 64, 64, 1]))
  rt.build(is_autoencoder=True)
  rt.train(batch_size=256, epochs=100,
           valid_batch_size=200, validation_steps=1000)
```

Figure 7.2: Example code for training a model with TensorLight. A convolutional autoencoder is trained on */gpu:0* and */gpu:1* for 100 epochs in order reconstruct Moving MNIST image sequences. It uses Adam optimizer with a low exponential learning rate decay.

When the training is complete, the runtime is ready to do inference or testing on the model. In case the model's performance is still unsatisfactory, it is possible to continue the previous training for a few steps more. In addition, the framework allows to register a validation callback in order to perform specific functionality after every validation process. In this case, a GIF animation is generated that compares the ground truth with its reconstruction of a randomly generated sequence from the validation set. This is illustrated in Figure 7.3.

```
13  # continue training
14  def on_valid(rt, global_step):
15    inputs, _ = rt.datasets.valid.get_batch(1)
16    pred = rt.predict(inputs)
17    tl.utils.video.write_multi_gif(os.path.join(rt.train_dir, "{}.gif".format(global_step)),
18                                   [inputs[0], pred[0]], fps=5)
19  rt.train(batch_size=256, steps=10000, valid_batch_size=200,
20           validation_steps=1000, on_validate=on_valid)
```

Figure 7.3: Code snippet that continues the previous training for another 10,000 steps. Additionally, a validation hook is registered to write a GIF animation after every validation.

Evaluation

After the whole training process is complete, the model is ready to be evaluated on the test set. Therefore, model and test dataset instances are created and registered to a new runtime object. When building the model, the last checkpoint file and the model's hyperparameter-meta file can be loaded in order to rebuild the graph with the identical weights and model configuration. Last but not least, the testing process can commence, as depicted in line 7 of Figure 7.4.

```
1  with DefaultRuntime("/tmp/train") as rt:
2    rt.register_model(ConvAutoencoderModel())
3    rt.register_datasets(test_ds=MovingMNISTTestDataset("/tmp/data", as_binary=True,
4                                                        input_shape=[10, 64, 64, 1]))
5    rt.build(is_autoencoder=True, restore_model_params=True
6             restore_checkpoint=tl.core.LATEST_CHECKPOINT)
7    rt.test(batch_size=100, epochs=100)
```

Figure 7.4: Code sample for testing a model on a single device in TensorLight. Before, it restores the checkpoint file containing all weights, as well as the model's hyperparameters of the previous training.

8 Conclusion

We presented existing unsupervised deep learning approaches for future frame prediction in videos and incorporated their mentioned insights to build a neural network model that combines the its strengths. The proposed network architecture utilizes the recurrent decoder-encoder framework using ConvLSTM cells that are able to preserve the spatio-temporal correlations of the data. Further, the extensive use of batch normalization and the scheduled sampling training strategy enables our model to outperform many existing approaches in at least synthetic or simple videos, even though our network contains a lower model complexity and is trained for fewer iterations. An identified key driver to obtain accurate prediction results has been the choice of an appropriate loss function that considers human perception. However, the best composition of different objective functions strongly depends on the underlying data. The network was further evaluated quantitatively and qualitatively on three datasets of different complexity in several scenarios and investigated the models behavior regarding hyperparameter changes. The best performing model in each case was then compared to results from related works. Additionally, a high-level framework for TensorFlow projects was presented that enables to use advanced features out-of-the-box and radically reduces boilerplate code.

8.1 Discussion

After applying batch normalization and the scheduled sampling learning strategy, we were honestly surprised that our model was able to outperform related models by such a large margin. Nevertheless, we belief that is still space to improve the proposed architecture, the chosen hyperparameter configuration and particularly the data preprocessing. Regarding the latter, only simple rescaling of the data has been performed to be roughly zero mean, but dedicated the scaling of the values completely to batch normalization layers and the entire network.

We could also figure out that the choice of the appropriate loss function has a huge impact regarding the generated future frame predictions, if not even the most tremendous effects. However, as the evaluation in chapter 6 clearly shows, there is no perfect solution for this purpose. But it must be emphasized that the detailed properties of the used image or video data have to be analyzed in detail, in order to be able to achieve

good results. Having said that, the fine-tuning of neural networks in context of image processing tasks remains to be very difficult even with a good loss function at hand. This can be attributed to the discrepancy between the mathematical and perceptual similarity of two images. Also the use of perceptual motivated metrics presented in Section 2.6.2 is not always very helpful, because an increase in one metric can lead to a decline in others. It is easy to get lost when multiple metrics are used.

Finally, in can be assumed that the proposed model could be currently trained more effectively using a different deep learning framework than TensorFlow, at least at the time of this writing. This can be argued with the fact that the current batch normalization layer in this framework currently depends on some operations where no GPU kernel is implemented yet. Such a bottleneck might be the root cause why the training process of our model is so slow that it requires up to four days to train the network for only 100,000 steps. But this will certainly change in one of its next releases.

8.2 Future Work

There are at least the following five proposals for future work:

Firstly, the proposed network model should be examined and fine-tuned in more detail. We strongly belief that this architecture is able to obtain even better results after performing a more extensive hyperparameter search, training it for many more iterations or when using a larger dataset like Sports-1M. Unfortunately, this is beyond the timeframe of this thesis, that is why some evaluations were performed on networks which still had further potential in case of more training iterations.

Secondly, because the application of the scheduled sampling learning strategy for recurrent networks has improved our results in such an extent, it would be worthwhile experimenting with new variants of this approach. For instance, the recurrent network could dynamically grow in the course of the training process. Thereby, it could start to predict a single frame only until the validation loss reaches a specified threshold. Afterwards, the decoder RNN can be extended at runtime to predict more and more future frames per training iteration. As a result, such a network should be able to predict longer sequences with a higher stability regarding the quality of generated frames.

Thirdly, since the GAN approach described in Section 3.3 yields such promising results, one has to imagine what might be possible when the proposed model is plugged into this adversarial framework. Despite the fact that they use a very simple convolutional generator network, our proposed model as a replacement for their generator network would explicitly take advantage of the spatio-temporal properties of the data. Especially without the need that the model would have to learn these correlations from scratch.

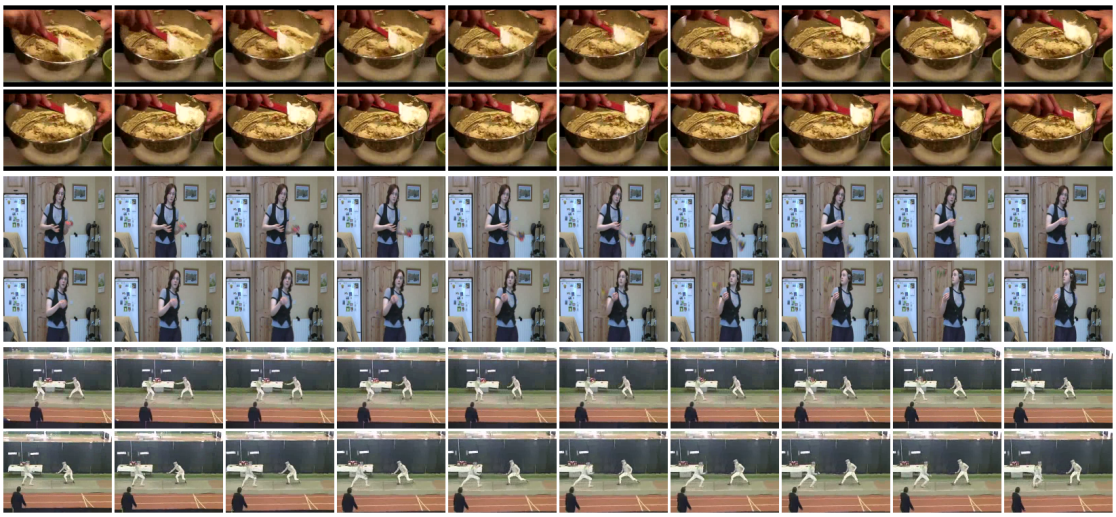
As a consequence, the additional adversarial network would introduce an additional objective function in feature space, whose benefits have already been mentioned in the end of section 4.2.1.

Next, a trained network instance of our model can be examined to serve as a pre-training for supervised learning tasks like human action recognition, which can be very helpful according to [Ben08, p. 20]. Similar efforts have already been taken in [SMS15] with positive results. In detail, it should be possible to detach the encoder components of our trained model, including its ability to generate a useful feature space representation given a sequence of frames, and plug it into a different network architecture specialized for classification. Unfortunately, performing such experiments with labeled video data is beyond the scope of this thesis.

Lastly, the proposed network architecture itself can be further extended to cope with different unsupervised tasks. To name just one, the recurrent components can be updated to bidirectional RNNs in order to solve tasks like slow motion video generation or video compression more effectively.

Appendix

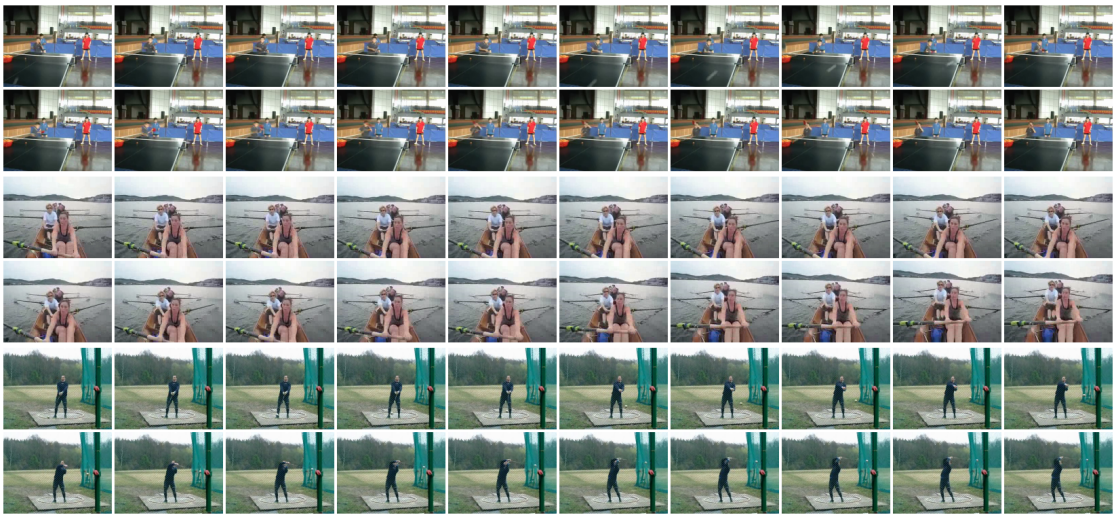
A.1 Dataset Images



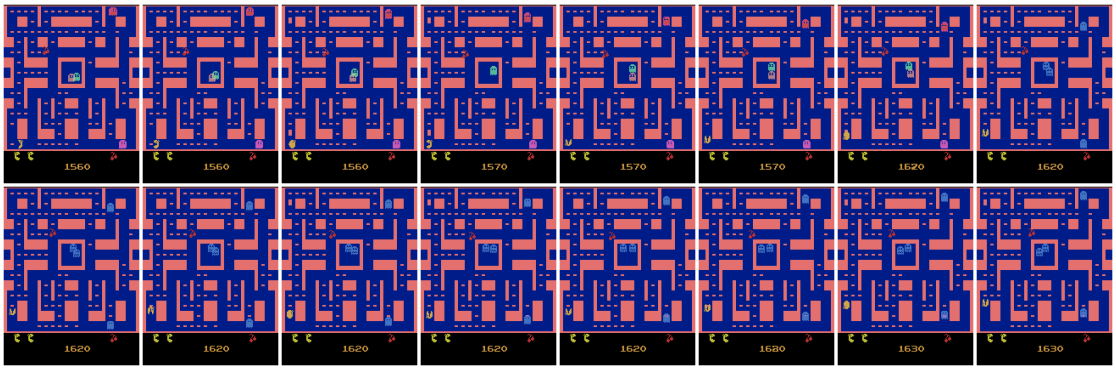
(a) Training set

Figure A.1: Randomly chosen and rescaled clip samples of size 160 × 120 from the UCF-101 dataset. There is only a very small portion of motion to see between one frame to the next, because the videos have a frame rate of 25 FPS.
(continues on next page)

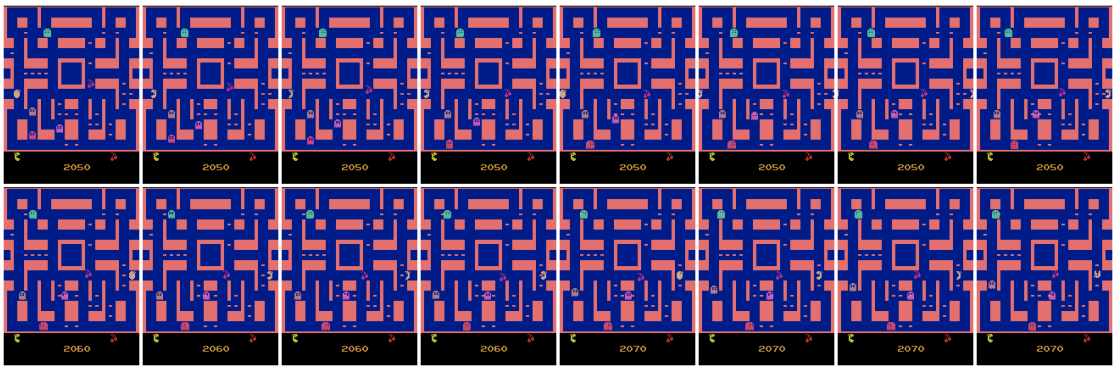
(b) Validation set



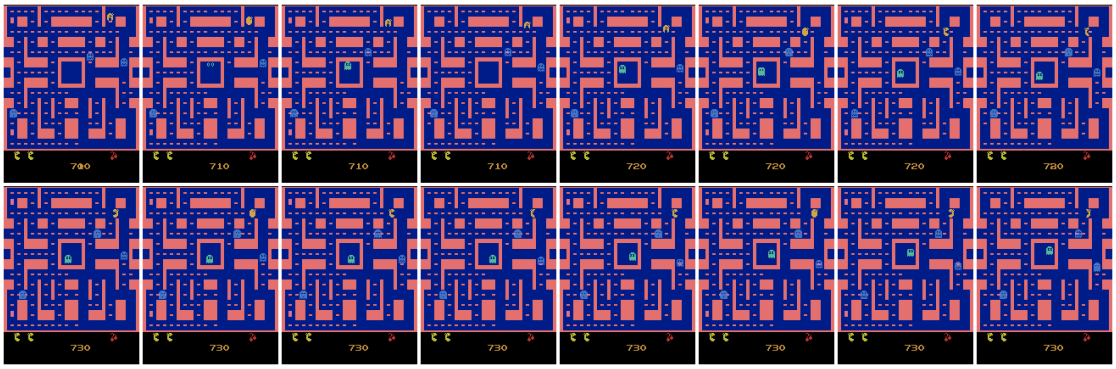
(c) Test set



(a) Training set



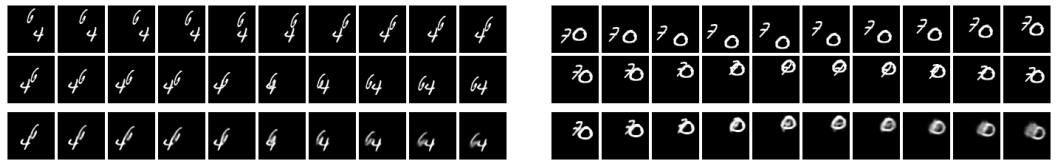
(b) Validation set



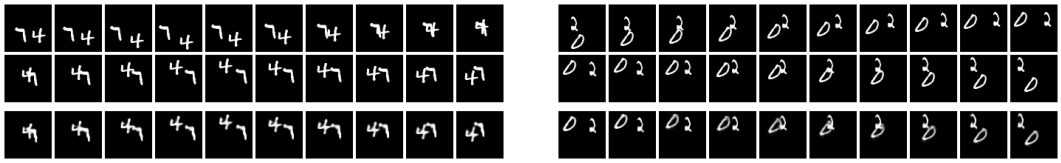
(c) Test set

Figure A.2: Exemplary sequences of game recordings from MsPacman dataset that have been samples randomly from the different splits. Each shown clip has a length of 16 frames of size 160×210.

A.2 Further Frame Predictions

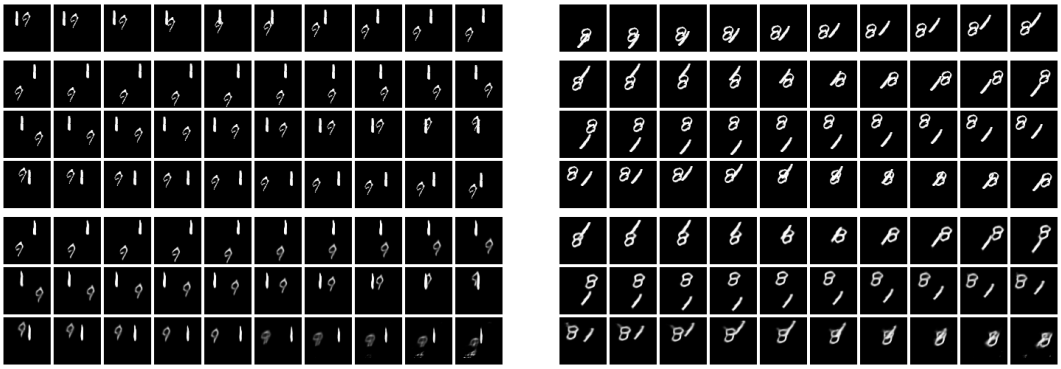


(a) (b)



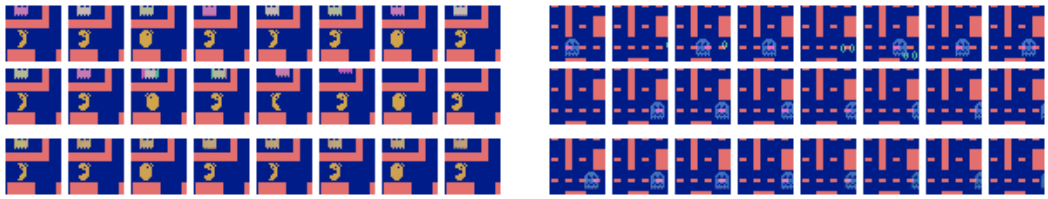
(c) (d)

Figure A.3: Further random prediction samples on Moving MNIST. From top to bottom: ground truth input and output sequence; predicted sequence of our 2-layer model.



(a) (b)

Figure A.4: Further out-of-domain runs on Moving MNIST. Our 2-layer model predicts 30 frames ahead, even that it is trained to predict 10 frames only. The spatio-temporal decoder is therefore unrolled for a longer time range.



(a) (b)

Figure A.5: Two further prediction examples of the 2-layer network on MsPacman. We have picked the best out of 8 random samples which contain at least some motion.

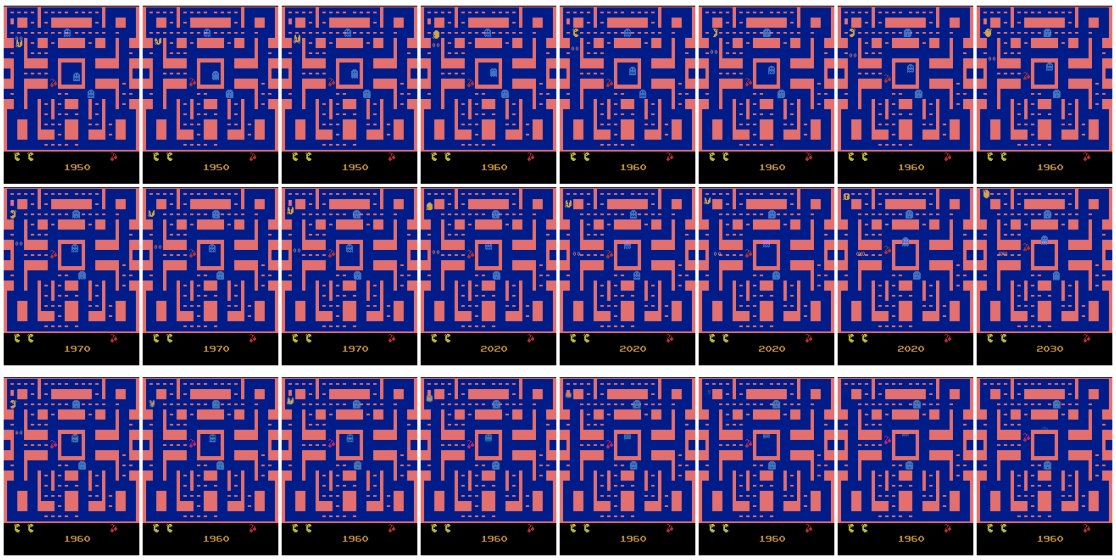


Figure A.6: Full screen prediction sample using the 2-layer model.

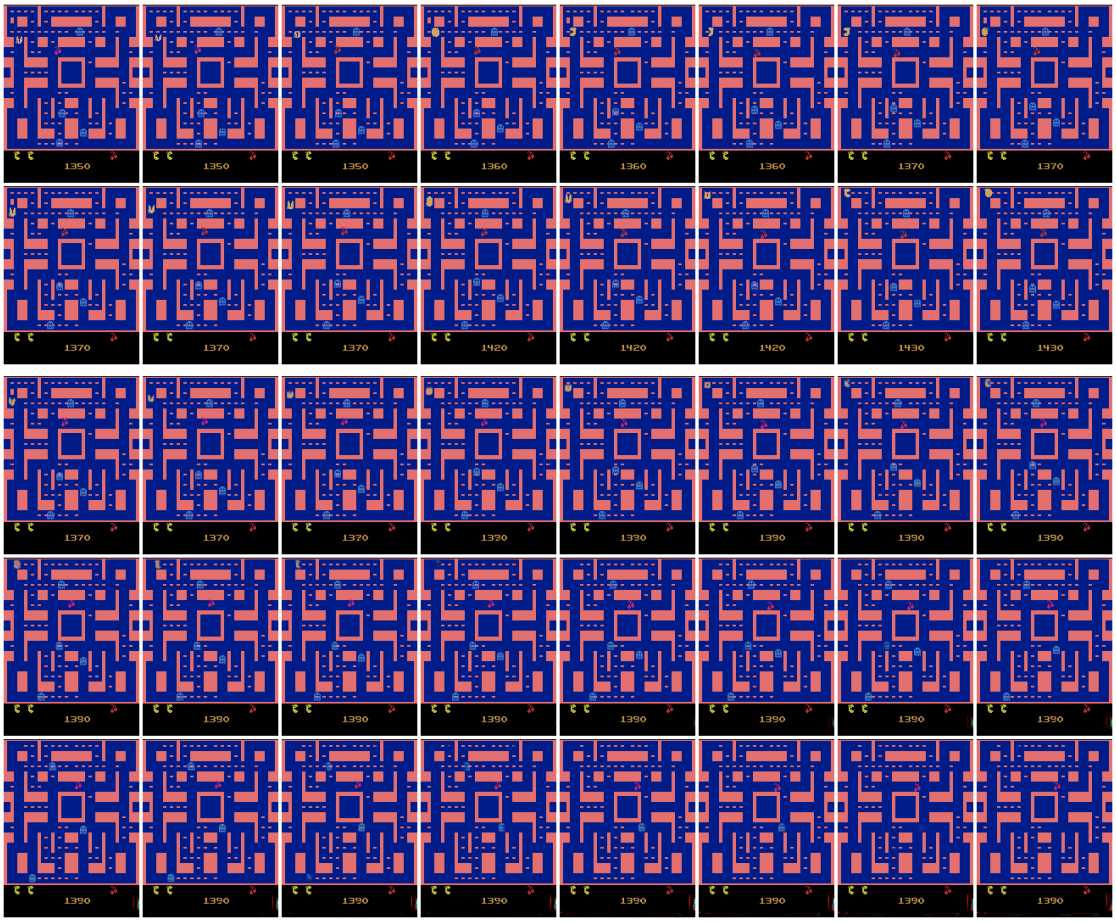
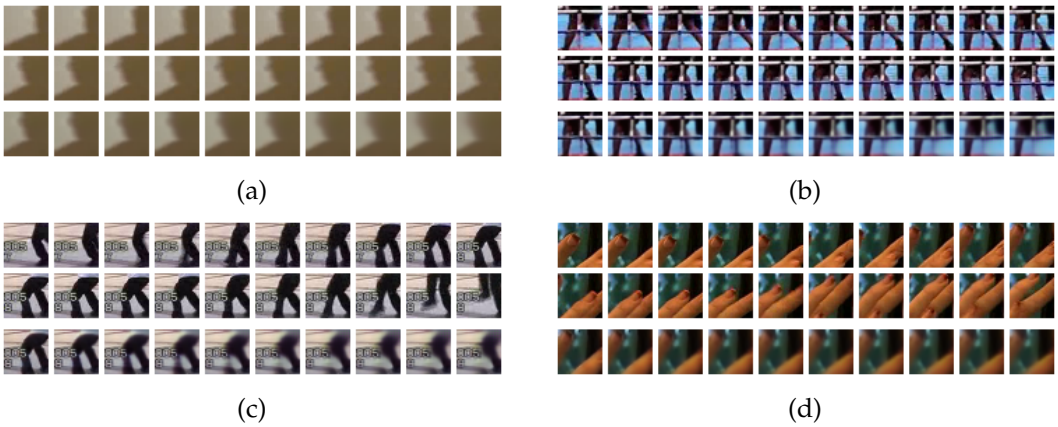


Figure A.7: Out-of-domain test using the 2-layer model to predict a longer-term future.



(a) (b) (c) (d)

Figure A.8: Further prediction samples of our model on UCF-101. From top to bottom: Input and ground truth target; forecasted future frames.

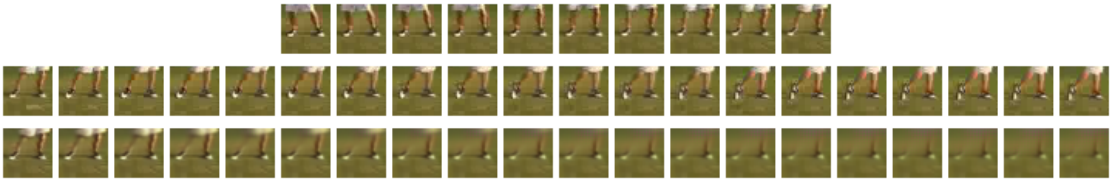
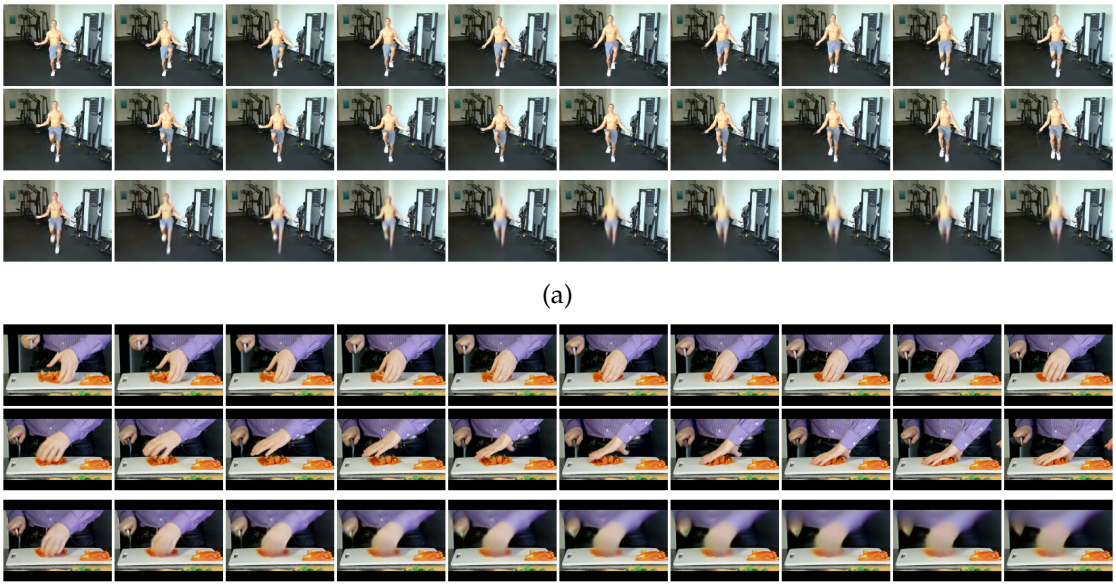


Figure A.9: Further long-term results on UCF-101 using our 3-layer ConvLSTM model.

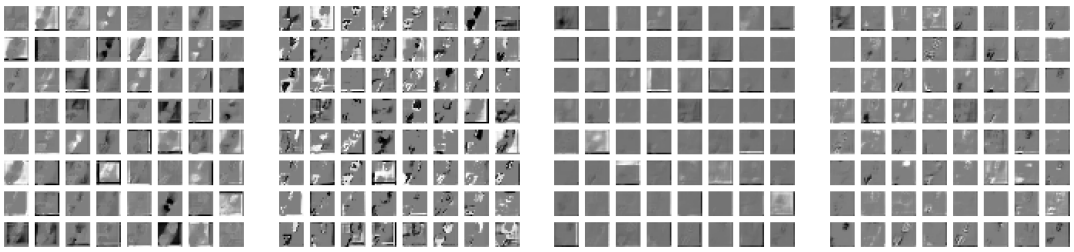


(a) (b)

Figure A.10: Further randomly selected full frame predictions of our 3-layer ConvLSTM model using the MAE-based triplet loss function on UCF-101.

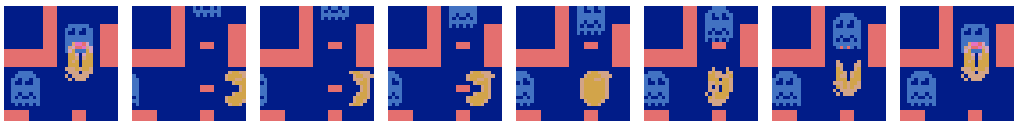
A.3 Learned Representations

(a) input sequence

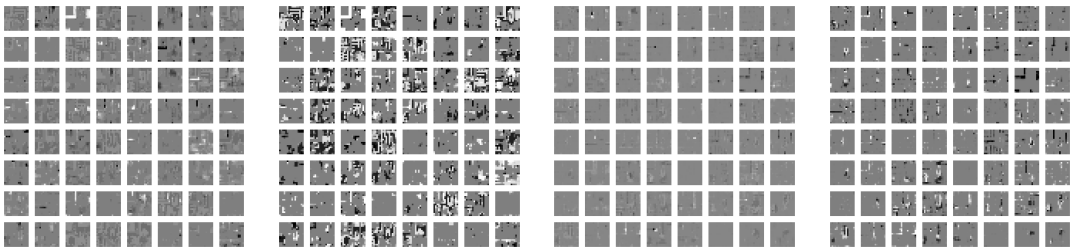


(b) C (1st layer) (c) h (1st layer) (d) C (2nd layer) (e) h (2nd layer)

Figure A.11: Example of the encoded representation using a 2-layer network in Moving MNIST, separated by each layer, cell state C and hidden output h. Each tensor of shape $16 \times 16 \times 64$ is visualized by splitting each single feature map.

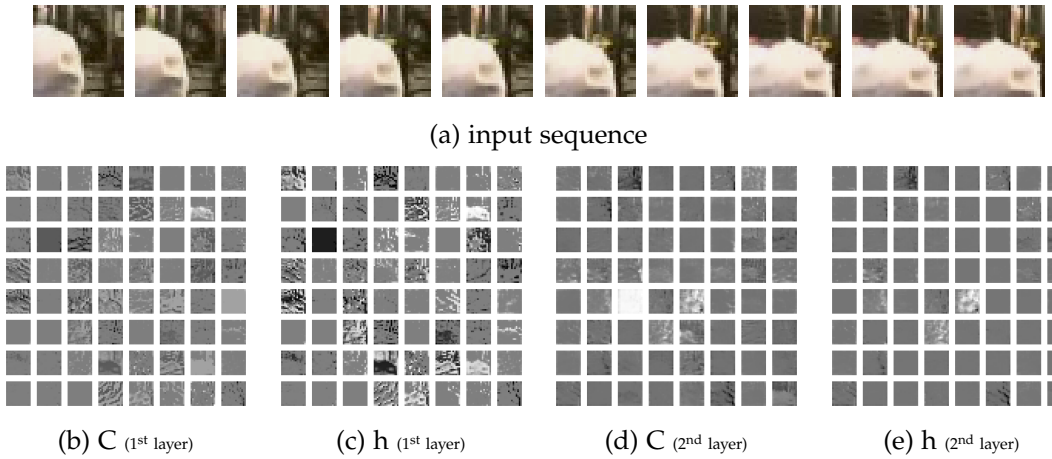


(a) input sequence



(b) C (1st layer) (c) h (1st layer) (d) C (2nd layer) (e) h (2nd layer)

Figure A.12: Encoded representation example using a 2-layer network in the MsPacman dataset, separated by each ConvLSTM layer, cell state C and hidden output h. Each tensor representation of shape $16 \times 16 \times 64$ can be illustrated by showing each feature map separately.



(a) input sequence

(b) C (1st layer) (c) h (1st layer) (d) C (2nd layer) (e) h (2nd layer)

Figure A.13: Example of an encoded representation using the 2-layer ConvLSTM model in UCF-101, separated by each layer, cell state C and hidden output h. Tensors of shape $16 \times 16 \times 64$ are visualized by splitting each single feature map, resulting in 64 grayscale images.

A.4 Code Listings

```
import tensorflow as tf
import tensorlight as tl
from tf.contrib.layers import *

class ConvAutoencoderModel(tl.model.AbstractModel):
  def __init__(self, weight_decay=0.0):
    super(ConvAutoencoderModel, self).__init__(weight_decay)

  @tl.utils.attr.override
  def inference(self, inputs, targets, feeds, is_train, device_scope, memory_device):
    with tf.variable_scope("Encoder"):
      conv1 = tl.network.conv2d("Conv1", inputs, 4, (5, 5), (2, 2),
                                weight_init=xavier_initializer_conv2d(),
                                regularizer=l2_regularizer(self.weight_decay),
                                activation=tf.nn.relu)
      conv1_bn = batch_norm(conv1, is_training=is_train, scope="conv1_bn")
      conv2 = tl.network.conv2d("Conv2", conv1_bn, 8, (3, 3), (2, 2),
                                weight_init=xavier_initializer_conv2d(),
                                regularizer=l2_regularizer(self.weight_decay),
                                activation=tf.nn.relu)
      conv2_bn = batch_norm(conv2, is_training=is_train, scope="conv2_bn")
      learned_rep = conv2_bn
    with tf.variable_scope("Decoder"):
      convt = tl.network.conv2d_transpose("Convt1", learned_rep, 4, (3, 3), (2, 2),
                                          weight_init=tl.init.bilinear_initializer(),
                                          regularizer=l2_regularizer(self.weight_decay),
                                          activation=tf.nn.relu)
      convt_bn = batch_norm(convt, is_training=is_train, scope="convt_bn")
      return tl.network.conv2d_transpose("Convt2", convt_bn, 1, (5, 5), (2, 2),
                                          weight_init=tl.init.bilinear_initializer(),
                                          regularizer=l2_regularizer(self.weight_decay),
                                          activation=tf.nn.sigmoid)
  @tl.utils.attr.override
  def loss(self, predictions, targets, device_scope):
    return tl.loss.bce(predictions, targets)

  @tl.utils.attr.override
  def evaluation(self, predictions, targets, device_scope):
    psnr = tl.image.psnr(predictions, targets)
    sharpdiff = tl.image.sharp_diff(predictions, targets)
    ssim = tl.image.ssim(predictions, targets)
    return {"psnr": psnr, "sharpdiff": sharpdiff, "ssim": ssim}
```

Figure A.14: Example implementation of a convolutional autoencoder model.

Acronyms

Adam	**Ad**aptive **M**oments **E**stimation
AI	**A**rtificial **I**ntelligence
ANN	**A**rtificial **N**eural **N**etwork
API	**A**pplication **P**rogramming **I**nterface
AS	**A**lways **S**ampling
BCE	**B**inary **C**ross-**E**ntropy
BN	**B**atch **N**ormalization
BPTT	**B**ack**p**ropagation **T**hrough **T**ime
BRNN	**B**idirectional **R**ecurrent **N**eural **N**etwork
CNN	**C**onvolutional **N**eural **N**etwork
ConvLSTM	**Conv**olutional **LSTM**
CPU	**C**entral **P**rocessing **U**nit
DCGAN	**D**eep **C**onvolutional **G**enerative **A**dversarial **N**etwork
FC	**F**ully-**C**onnected
FC-LSTM	**F**ully-**C**onnected **LSTM**
FCN	**F**ully-**C**onvolutional **N**etwork
FR	**F**ull **R**eference
GAN	**G**enerative **A**dversarial **N**etwork
GDL	**G**radient **D**ifference **L**oss
GIF	**G**raphics **I**nterchange **F**ormat
GPU	**G**raphics **P**rocessing **U**nit
GRU	**G**ated **R**ecurrent **U**nit
GT	**G**round **T**ruth
JPEG	**J**oint **P**hotographic **E**xperts **G**roup
JSON	**J**ava**S**cript **O**bject **N**otation
LSTM	**L**ong **S**hort-**T**erm **M**emory
MAE	**M**ean **A**bsolute **E**rror
MIT	**M**assachusetts **I**nstitute of **T**echnology
ML	**M**achine **L**earning
MLP	**M**ulti**l**ayer **P**erceptron
MNIST	**M**ixed **N**ational **I**nstitute of **S**tandards and **T**echnology

MS-SSIM	**M**ulti-**S**cale **S**tructural **Sim**ilarity
MSE	**M**ean **S**quared **E**rror
NN	**N**eural **N**etwork
OCR	**O**ptical **C**haracter **R**ecognition
PCA	**P**rinciple **C**omponent **A**nalysis
PSNR	**P**eak **S**ignal-to-**N**oise **R**atio
ReLU	**R**ectified **L**inear **U**nit
RGB	**R**ed **G**reen **B**lue
RNN	**R**ecurrent **N**eural **N**etwork
SDG	**S**tochastic **G**radient **D**escent
SS	**S**cheduled **S**ampling
SSIM	**S**tructural **Sim**ilarity
TUM	**T**echnische **U**niversität **M**ünchen
UCF	**U**niversity of **C**entral **F**lorida

List of Symbols

$C^{(\tau)}$	Cell state at time step τ
$Var(x)$	Variance of x
W_i	Weight element
$\Omega(x)$	Regularizer term
$*$	Convolution operation
ℓ_1	Least absolute deviations
ℓ_2	Least square errors
η	Learning rate
λ	Regularization coefficient
$\mathbb{E}(x)$	Expectation value of x
$\mathcal{L}(x)$	Loss function
μ_x	Mean value of x
∇	Gradient
$\odot$	Hadamard product
$\phi(x)$	Activation function
$\sigma(x)$	Sigmoid activation function
σ_x	Standard deviation of x
τ	Time step
$\mathbf{X}_{c,r}$	Element at column c and row r of matrix X
$\mathbf{x}_{max}$	Maximum possible value of x
U	Uniform distribution
$\arg\min_x$	Value of x for which the minimum is attained
$\tilde{x}$	Corrupted value of x
b	Bias (or negative threshold)
d_x	Dimensionality of x
$h^{(\tau)}$	Hidden state at time step τ
k	Kernel size value
p	Padding value / probability
s	Stride value
x_i	Input element
y_i	Output element

List of Figures

List of Tables

Bibliography

[Aba+15] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015.

[ABP05] J. Ascenso, C. Brites, and F. Pereira. "Improving Frame Interpolation with Spatial Motion Smoothing for Pixel Domain Distributed Video Coding." In: *5th EURASIP Conference on Speech and Image Processing, Multimedia Communications and Services*. 2005, pp. 1–6.

[Bal+16] N. Ballas, L. Yao, C. Pal, and A. Courville. "Delving Deeper into Convolutional Networks for Learning Video Representations." In: *ICLR*. Mar. 2016.

[Ben+15] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer. "Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks." In: Sept. 2015.

[Ben08] Y. Bengio. *Learning Deep Architectures for AI*. Tech. rep. Université de Montréal, July 2008.

[Bis06] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer Publishing, 2006. ISBN: 9788132209065.

[BSF94] Y. Bengio, P. Simard, and P. Frasoni. "Learning Long-Term Dependencies with Gradient Descent is Difficult." In: *IEEE Transaction on Neural Networks* 5.2 (Mar. 1994), pp. 157–166.

[Cho+14] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation." In: *EMNLP*. 2014.

[Chu+14] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling." In: 2014.

[Coo+15] T. Cooijmans, N. Balls, C. Laurent, and Ç. Gülçehre. "Recurrent Batch Normalization." In: *Proc. of IEEE*. 2015.

[Coo16] M. Cooper. *Adversarial Video Generation*. 2016. URL: https://github.com/dyelax/Adversarial_Video_Generation (visited on 10/14/2016).

[Cou+13] C. Couprie, C. Farabet, Y. LeCun, and L. Najman. "Causal Graph-Based Video Segmentation." In: *ICIP*. 2013.

[DB16] A. Dosovitskiy and T. Brox. "Generating Images with Perceptual Similarity Metrics based on Deep Networks." In: Feb. 2016.

[Dea+12] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. "Large Scale Distributed Deep Networks." In: *NIPS*. 2012.

[Don+14] J. Donahue, L. A. Hendricks, M. Rohrbach, S. Venugopalan, S. Guadarrama, K. Saenko, and T. Darrell. "Long-term Recurrent Convolutional Networks for Visual Recognition and Description." In: 2014.

[DV16] V. Dumoulin and F. Visin. "A guide to convolution arithmetic for deep learning." Mar. 2016.

[Fis+15] P. Fischer, A. Dosovitskiy, P. Häusser, C. Hazırbas, and V. Golkov. "FlowNet: Learning Optical Flow with Convolutional Networks." In: 2015.

[GB10] X. Glorot and Y. Bengio. "Understanding the difficulty of training deep feedforward neural networks." In: *AISTATS*. 2010.

[GBB11a] X. Gloro, A. Bordes, and Y. Bengio. "Domain Adaptation for Large-Scale Sentiment Classification: A Deep Learning Approach." In: *ICML*. June 2011.

[GBB11b] X. Glorot, A. Bordes, and Y. Bengio. "Deep Sparse Rectifier Neural Networks." In: *AISTATS*. 2011.

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. "Deep Learning." Book in preparation for MIT Press. 2016.

[Goo+14] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. "Generativ Adversarial Nets." In: *NIPS*. 2014.

[Gra+06] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber. "Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks." In: *ICML*. 2006, pp. 369–376.

[Gre+15] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber. "LSTM: A Search Space Odyssey." In: 2015.

[GS00] F. A. Gers and J. Schmidhuber. "Recurrent Nets that Time and Count." In: *IEEE Trans. on Neural Networks* 3 (2000), pp. 850–855.

[Hin13] G. Hinton. *Neural Networks for Machine Learning: Training RNNs with Back Propagation*. 2013. URL: https://www.coursera.org/learn/neural-networks (visited on 09/25/2016).

[Hoc91] J. Hochreiter. "Untersuchungen zu dynamischen neuronalen Netzen." MA thesis. Arcisstr. 21, 80333 Munich , Germany: Technische Universität Munchen, June 1991.

[HS06] G. E. Hinton and R. R. Salakhutdinov. "Reducing the Dimensionality of data with Neural Networks." In: *Sience* 313.5786 (July 2006), pp. 504–507.

[HS97] S. Hochreiter and J. Schmidhuber. "Long short-term memory." In: *Neural Computing*. 1997, pp. 1735–1780.

[IS15] S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." In: *Proc. of IEEE*. 2015.

[Ji+13] S. Ji, W. Xu, M. Yang, and K. Yu. "3D Convolutional Neural Networks for Human Action Recognition." In: *IEEE Trans. PAMI*. 2013, pp. 221–231.

[Jon15] A. L. Jone. *An Explanation of Xavier Initialization*. 2015. URL: `http://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization` (visited on 09/22/2016).

[Kar+14] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. "Large-scale Video Classification with Convolutional Neural Networks." In: *CVPR*. 2014, pp. 1725–1732.

[Kar12] A. Kar. "Future Image Prediction using Artificial Neural Networks." In: 2012.

[Kar15] A. Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks*. May 2015. URL: `http://karpathy.github.io/2015/05/21/rnn-effectiveness` (visited on 09/25/2016).

[KB15] D. P. Kingma and J. L. Ba. "Adam: A Method for Stochasitc Optimization." In: *ICLR*. 2015.

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." In: *NIPS*. 2012.

[LeC+98] Y. LeCun, L. Bottou, Y. Bengio, and P. Hafner. "Gradient-Based Learning Applied to Document Recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[LZR16] L. Lu, X. Zhang, and S. Renals. "On Training Recurrent Neural Network Encoder-Decoder For Large Vocabulary End-to-End Speech Recognition." In: *ICASSP*. Mar. 2016.

[MCL16] M. Mathieu, C. Couprie, and Y. LeCun. "Deep Multi-Scale Video Prediction Beyond Mean Square Error." In: *ICLR*. Feb. 2016.

[MF14] A. Makhzani and B. Frey. "k-Sparse Autoencoders." In: *ICLR*. Dec. 2014.

[Ng+13] A. Ng, J. Ngiam, C. Y. Foo, Y. Mai, and C. Suen. *Standford: Unsupervised Feature Learning and Deep Learning*. 2013. URL: http://ufldl.stanford.edu/wiki/index.php/Data_Preprocessing (visited on 10/03/2016).

[Ng+16] J. Y.-H. Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici. "Beyond Short Snippets: Deep Networks for Video Classification." In: *CVPR*. 2016.

[Nie15] M. A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[Ola15] C. Olah. *Understanding LSTM Networks*. 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs (visited on 09/20/2016).

[Pen+16] X. Peng, R. Feris, X. Wang, and D. Metaxas. "A Recurrent Encoder-Decoder Network for Sequential Face Alignment." In: Aug. 2016.

[PHC16] V. Pătrăucean, A. Handa, and R. Cipolla. "Spatio-temporal video autoencoder with differentiable memory." In: *ICLR*. 2016.

[Rid+16] K. Ridgeway, J. Snell, B. D. Roads, R. S. Zemel, and M. C. Mozer. "Learning to Generate Images with Perceptual Similarity Metrics." In: Mar. 2016.

[Rud16] S. Ruder. *An overview of gradient descent optimization algorithms*. 2016. URL: http://sebastianruder.com/optimizing-gradient-descent (visited on 09/22/2016).

[Shi+15] X. Shi, Z. Chen, H. Wang, and D.-Y. Yeung. "Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting." In: 2015.

[SMS15] N. Srivastava, E. Mansimov, and R. Salakhutdinov. "Unsupervised Learning of Video Representations using LSTMs." In: *ICML*. 2015.

[Sri+14] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." In: *Journal of Machine Learning Research* 15 (June 2014), pp. 1929–1958.

[SVL14] I. Sutskever, O. Vinyals, and Q. V. Le. "Sequence to Sequence Learning with Neural Networks." In: *NNIPS*. 2014.

[SZ14] K. Simonyan and A. Zisserman. "Two-Stream Convolutional Networks for Action Recognition in Videos." In: *Proc. NIPS*. 2014, pp. 568–576.

[SZS12] K. Soomro, A. R. Zamir, and M. Shah. "UCF101: A Dataset of 101 Human Actions Classes From Videos in The Wild." In: *CRCV-TR-12-01*. Nov. 2012.

[Ver12] N. K. Verma. "Future Image Frame Generation using Artificial Neural Network with Selected Features." In: *AIPR*. Oct. 2012.

[Wan+04] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. "Image Quality Assessment: From Error Visibility to Structural Similarity." In: *IEEE Trans. Image Processing* 13.4 (Apr. 2004), pp. 600–612.

[WB02] Z. Wang and A. C. Bovik. "A Univeral Image Quality Index." In: *IEEE Signal Processing Letters* 9 (Mar. 2002), pp. 81–84.

[WGH15] J. Walker, A. Gupta, and M. Hebert. "Dense Optical Flow Prediction from a Static Image." In: Dec. 2015.

[WSB03] Z. Wang, E. P. Simonselli, and A. C. Bovik. "Multi-Scale Structural Similarity for Image Quality Assessment." In: *IEEE Conference on Signals, Systems and Computers*. Nov. 2003.

[Yeh+16] R. Yeh, C. Chen, T. Y. Lim, M. Hasegawa-Johnson, and M. N. Do. "Semantic Image Inpainting with Perceptual and Contextual Losses." In: July 2016.

[Zha+16] H. Zhao, O. Gallo, I. Frosio, and J. Kautz. "Loss Functions for Neural Networks for Image Processing." In: June 2016.