Temporal Super-Resolution of Multiple Fast-Moving Objects

Temporal Super-Resolution of Multiple Fast-Moving Objects

Adrian Klaeger

MSc Thesis
August 20, 2021

Supervisor: Denys Rozumnyi

Abstract

Blurring is a common problem in image-related tasks and deblurring techniques are an active area of research. *DeFMO: Deblurring and Shape Recovery of Fast Moving Objects* has set the state of the art in deblurring objects moving at high speed through an otherwise static scene, using deep neural networks in an encoder-decoder configuration to generate temporal super-resolution frames of the object along its trajectory, given only the image of the blurred object and its background.

This work re-implements DeFMO to create an extensible, user-friendly system for experimentation and exploration of alternative approaches, mainly to address the requirement that the background be given and the undefined direction of motion of the object causing the blur.

A new double-blur approach is used, replacing the input to the system with two consecutive images, each containing the blurred object in a different part of its trajectory, thus disambiguating the direction of motion. This approach is shown to be extensible to multiple blurred objects in a single image, producing decent results even when the trajectories of the objects intersect.

Acknowledgements

I would like to thank my supervisor, Denys Rozumnyi, for his creative input, patient guidance, and kind support over all these months.

Contents

Chapter 1

Introduction

Blurring is a common issue everyone has faced when using a camera. An object appears blurred when it moved relative to the camera while the camera's aperture was open, with the severity of the blur determined by the amount of motion in that time. The problem is therefore exacerbated by long exposure times, such as in low-light conditions, or by fast-moving objects.

Many deblurring approaches study scenes where any object, or commonly the whole image is blurred. These blurred images can be modeled as being formed through the convolution of the sharp image with a blur kernel and added noise. Koh et al. [11] conducted a survey of current techniques in the deblurring of such images, which they group into two broad categories: blind and non-blind methods. Non-blind methods have the blur kernel available as an input, focusing either on the task of using this kernel for deconvolution of the blurred image, or on the following denoising of the result. Blind methods do not have the blur kernel given, and must either begin by estimating this kernel, followed by a non-blind approach, or they must perform a direct image-to-image regression, without estimating the blur kernel.

Blind image-to-image regression without estimating a blur kernel is mainly done using deep learning. Not basing the approach on a fixed blur kernel allows considering more realistic, non-uniform blurs with complex motion, including varying depth and occlusions [11]. In general, these approaches will estimate a sharp image from the blurred input, which is then compared to the true sharp image. For example, DeblurGAN [13] uses the typical GAN architecture with a separate network that tries to discriminate between the real sharp image and the deblurred version.

Jin et al. [10] take a new approach by instead reconstructing a sequence of images from a single blurred image, representing different instants of the scene as it might have been captured by a high-speed camera. Using synthetic datasets can further facilitate this approach, as these sub-frames can be explicitly generated and used as the ground truth for supervised learning.

The combination of these ideas brings us to this work, based on the 2020 paper *DeFMO: Deblurring and Shape Recovery of Fast Moving Objects* [16], which defines fast-moving objects (FMOs) as "objects that move over a distance larger than their size within the camera exposure time". The specific type of blurring studied by DeFMO - and consequently this work - is that of images captured by a stable camera, which feature such an FMO, leaving a blurred streak across an otherwise in-focus image.

Such images commonly occur in diverse fields such as sports, moving traffic, nature photography, or even satellites in astrophotography, but can occur in any scene under the right, or rather the wrong conditions, especially with poor cameras in low light.

The task is now to deblur such an FMO, but due to the fast-moving nature of the FMO, a single deblurred image will not tell the whole story, as the object can rotate freely in 3D space during its trajectory. Instead, the entire motion should be reconstructed, at least as seen from the camera, by reconstructing a series of individual frames along the trajectory of the FMO, as would have been captured by a slow-motion camera with a much shorter exposure time.

1.1 DeFMO

DeFMO represents the current state of the art for this task. While previous work - e.g. *Tracking by Deblatting* [12] - was restricted to simpler, mostly spherical objects with a uniform appearance along their trajectories, DeFMO allows full 3D translation and rotation for objects with arbitrary geometries.

DeFMO uses deep neural networks to analyze images featuring FMOs and then generate the corresponding high-speed frames. To train the neural network, DeFMO uses a synthetic dataset of FMO images, that also provides the ground truth for what the high-speed intermediate frames should look like. With this approach, the problem mainly becomes a supervised learning task, although the DeFMO system uses additional self-supervised loss functions, as these were shown to improve its generalization performance to real-world datasets.

The input to the DeFMO system is a pair of images: the main image featuring the FMO, as well as a plain background image without the FMO. Needing to have the plain background available rules out some applications, but in many applications such as sports, the source material is video, with the FMO appearing as a traveling blur across multiple consecutive frames of the video, in which case the background can be estimated as a median of these frames.

Further details on the DeFMO Architecture are provided in Chapter 2.

1.2 This Work

The objective of this work is to build on DeFMO. However, the idea is not primarily to improve the system as-is, e.g. by tinkering with hyper-parameters and the neural network architecture to improve the score, but rather to try to address some of the weaknesses of the current approach by exploring alternative approaches while keeping the basic setup mostly the same.

Engineering

Before new approaches can be explored, the current system needs to be re-implemented to facilitate this task. The creation of a modular, extensible, and user-friendly system for experimentation in this area is a major focus of this work. The goals and further details of this implementation are described in Chapter 3.

Some of the issues addressed by the re-implementation were:

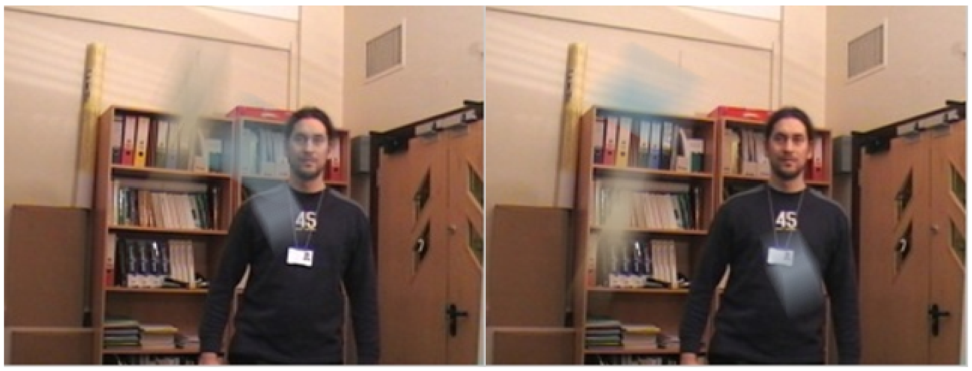
- The existing dataset generation method produced ready-made input images, which could consequently not be reused for some of the alternate input formats described in the next section.

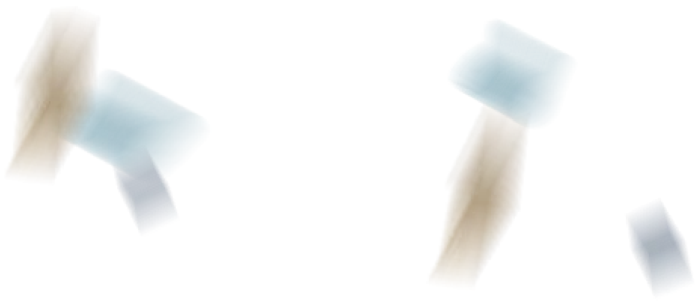
  Instead, only the blurs were rendered with transparency information, allowing them to be combined in flexible ways and overlaid on different backgrounds on-demand at training time.

- For 50'000 training sequences, the existing dataset consisted of 1.5 million files and nearly 1 TB of data and took about 5 days to generate on a computing cluster.

  Using the modern WebP [7] image format and a ZIP file as a container, a dataset of this size was reduced to a single 2GB file. Using a GPU rasterization renderer, the process took only 12 hours on a single GPU.

- The parameters of the existing approach, such as the input format and the loss functions used, were hard-coded in many ways, making modifications and extensions difficult.


(a) "double blur" input to the network


(b) above input without background for reference

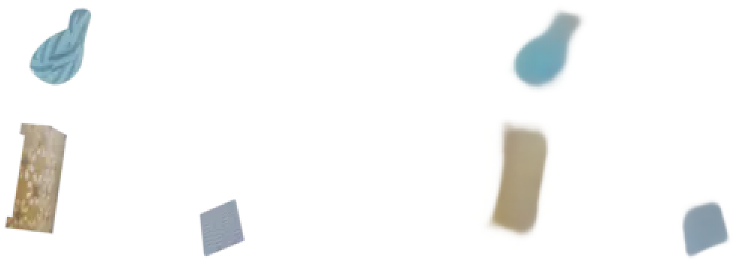
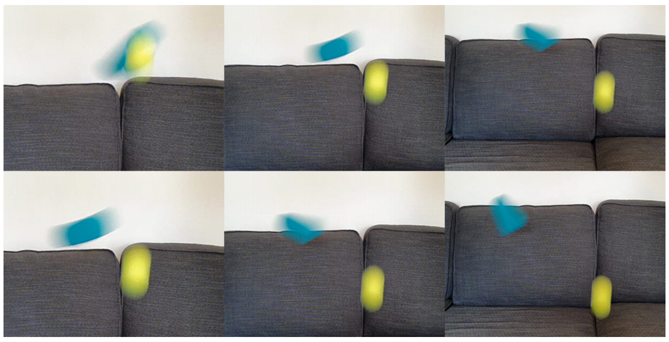
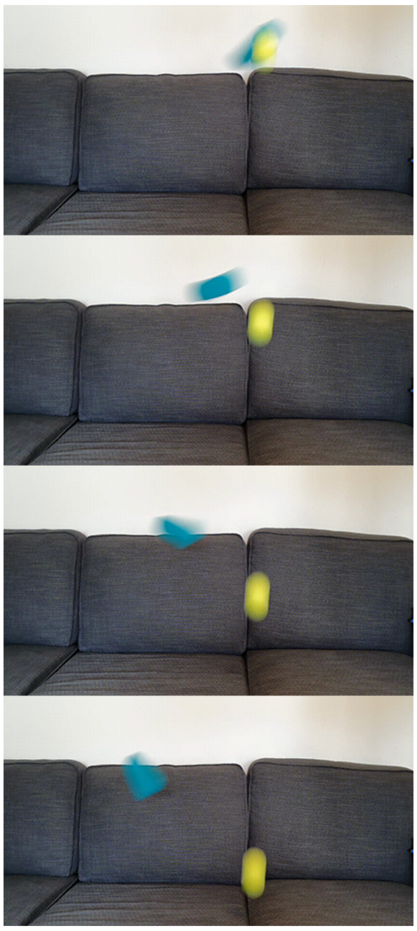

(c) sample ground truth frame and corresponding output of the network

Figure 1.1: Example result of this work on synthetic data: the "double blur" approach disambiguates the direction of motion, enabling the extension of the DeFMO method to multiple FMOs in a single image.

(a) real-world objects used


(c) pair-wise consecutive frames cropped and used as input to the network


(b) sequence of frames from the input video showing the trajectories of the objects

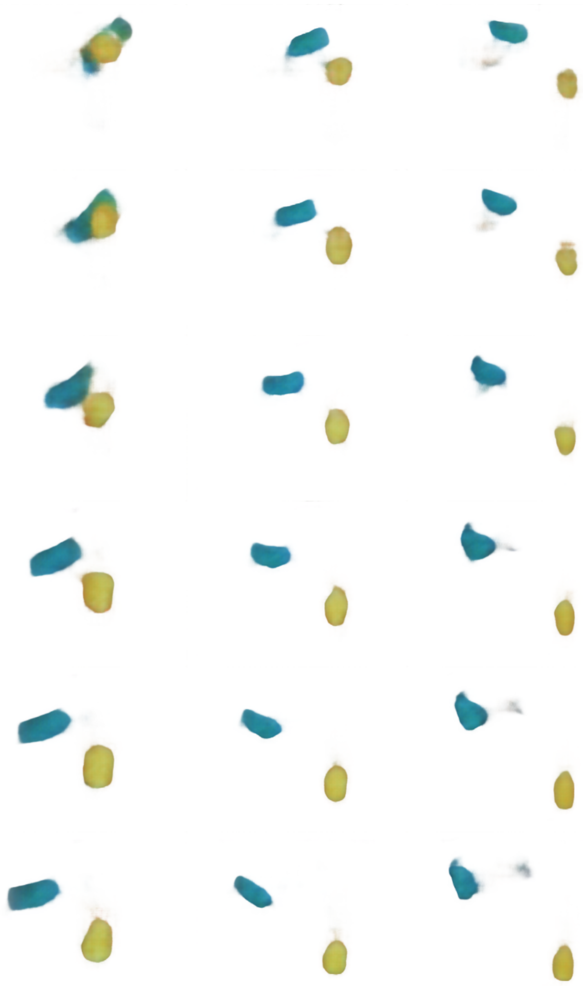

(d) predicted sub-frames for each of the input pairs above

Figure 1.2: Example result of this work on real-world data, containing a tennis ball and an eyeglass case, captured using a smartphone camera at 30fps.

A modular and extensible system was implemented, for example making any loss function addressable and configurable by name, as well as simplifying the definition of loss functions using object-oriented inheritance.

- The existing system was written in pure PyTorch, which is a good choice, but requires a lot of boilerplate code, which can be error-prone, especially when dealing with more complex tasks such as distributed learning.

  Instead, the PyTorch Lightning [6] framework was used, which automates many of the most common engineering requirements.

Experimentation

The alternate approaches explored were chosen in part to address the following weaknesses of the current approach:

- Requiring a background frame.

  The background frame being provided significantly helps to define the problem to be solved, as the mask of the FMO is almost given as the difference between the input image and the background. Completely eliminating the background frame makes the problem much more difficult, as the system then needs to learn to localize the FMO in addition to its other tasks. Also, disentangling the FMO from the background becomes more difficult, as the pixel information of the FMO is mixed with that of the background, with no indication of which belongs to which.

- Undefined direction of motion.

  Given only a single image of an FMO, it is impossible to determine in which direction the FMO moved, as both possible directions would produce an identical blur. In DeFMO, both possible directions are considered, but only the one that happens to fit the ground truth is used for evaluation.

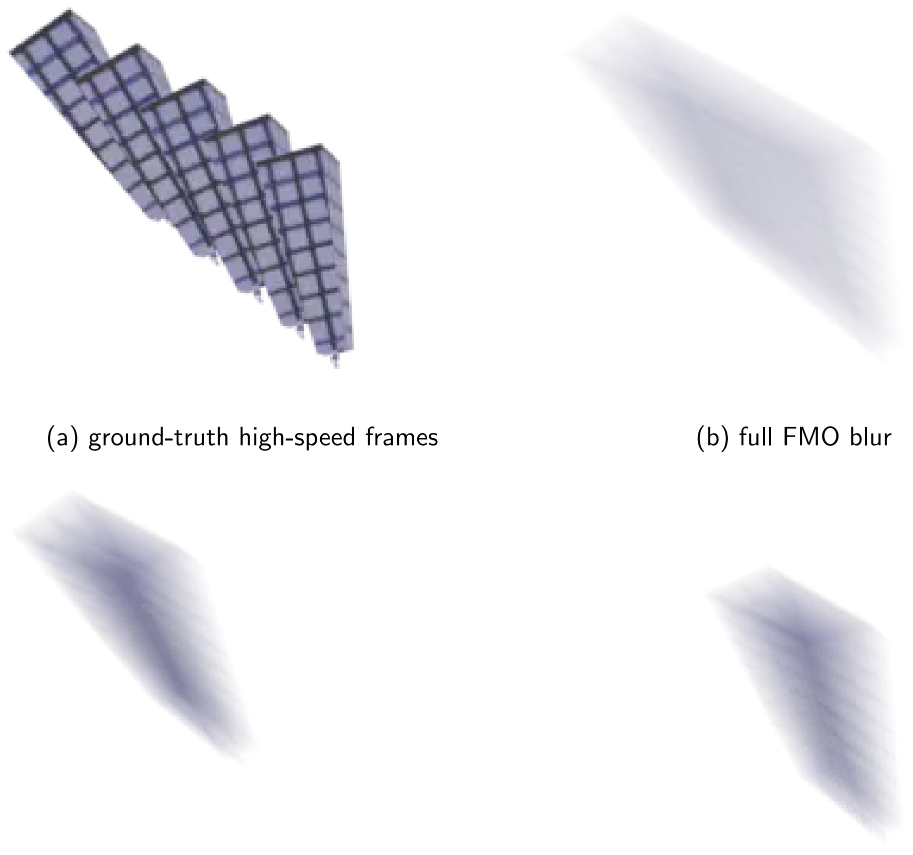
- Requiring ground truth data.

  The main reason why synthetic data is used is that a ground truth of the high-speed frames is required for training. Eliminating the need for ground-truth data would allow the system to be trained using any available real-world FMO dataset.

One of the main options explored in this work is the idea of using two consecutive video frames as input, each featuring the blur in a different but adjacent part of its overall trajectory, as shown in Figure 1.3. While this still requires more input than a single frame would provide, it has two main advantages:

- Estimating the background as the mean of several frames requires at least three frames, but typically more are used for a robust result. This option reduces the number of required frames to two.
- Having two consecutive frames available disambiguates the direction the FMO was traveling in.

Having two frames of the FMO available could potentially also provide more image data for the reconstruction, but also poses new challenges. For example, the background is less well-defined, especially since the two parts of the blur will often have a region of overlap in the center.



(a) ground-truth high-speed frames

(b) full FMO blur

(c) first half-blur

(d) second half-blur

Figure 1.3: Instead of creating a single blur from the ground-truth frames, the blur can be split into multiple parts, as though from multiple frames of an input video.

The third issue above is also explored: might it be possible to use a fully self-supervised approach - that is to say one that does not require ground-truth data? Defining the task for the network to learn becomes significantly more difficult without ground-truth data.

The main approach considered here involves input reconstruction, i.e. turning the output of the network back into an FMO by averaging the rendered frames, then comparing the result to the original image. If the network correctly predicted the high-speed frames, the two should match.

The main challenge faced here is that it is very easy for the network to "cheat" by finding other means of reconstructing the input, such as by returning the whole FMO as a semi-transparent streak instead of opaque individual high-speed frames.

All experiments and results are detailed in Chapter 4.

Contributions

In summary, the contributions of this work are:

- Creation of a user-friendly system for experimenting with new approaches to the DeFMO task.
- Showing that the number of frames required from an input video can be reduced to just two, achieving comparable performance while disambiguating the travel direction of the FMO.
- Showing that the approach can also be extended to multiple FMOs in one scene.
- Promising first steps towards a fully self-supervised learning approach.

Chapter 2

DeFMO Architecture

At its core, the DeFMO system has an Encoder/Decoder architecture. Input images are stacked in the channel dimension and are encoded into a latent space. For each desired output frame, this latent representation is augmented with the corresponding timestamp in the range [0,1] and passed through the decoder, which renders the appearance of the object at that timestamp.

2.1 Encoder

The Encoder is based on the ResNet-50 [8] architecture. This architectural style was introduced as a breakthrough in image classification and has since been widely used in many applications, as it allows the training of deeper neural networks while avoiding some commonly faced problems such as vanishing gradients.

Vanishing gradients refers to an issue in deep neural networks, where gradients can get smaller and smaller as they are back-propagated through the network, resulting in slow or no learning in the earlier layers of the network. Residual neural networks (ResNets) add *skip connections* that create shortcuts between layers in the network, thereby providing more direct paths for the gradients during back-propagation, offsetting these issues.

As a model built for image classification, ResNet-50 contains pooling layers and a final fully connected layer to perform the actual classification. We remove these layers, keeping the output of the final convolutional block as the latent representation of the input images. The pooling layers at the beginning and the end of the network down-sample the spatial dimensions, which can lead to more robust classification, but would lose valuable spatial information for our purposes.

The result is a latent representation of 2048 channels with spatial dimensions reduced by a factor of 16.

2.2 Decoder

The decoder (or "renderer"), initially convolves the 2049 channels (2048 latent representation + 1 timestamp channel) down to 1024 channels, then inflates the spatial dimensions back to the original size by applying four PixelShuffle [18] operations, interspaced with ResNet Bottlenecks.

A PixelShuffle operation simply rearranges elements from the channel dimension to the spatial dimensions, resulting in a doubling of the spatial dimensions and quartering of the channels.

ResNet Bottlenecks are basic building blocks of the ResNet architecture. These are convolutional blocks that reduce the number of channels by a given amount, then perform a convolution on this less dense feature vector, before finally increasing the number of channels back to the original amount. The block also features a skip connection as described above.

The resulting output has the same spatial dimensions as the input to the decoder and four channels, as desired for three color dimensions and an alpha mask.

A sigmoid activation is applied to the output, to constrain the values to the [0,1] range required for an image.

2.3 Losses

The original DeFMO architecture applies five different loss functions to the above network, each of which guides a different part of the system towards the desired state.

Appearance Reconstruction Loss

The principal loss function is a supervised loss, comparing the produced frame renders to the ground truths available in the synthetic dataset.

Image Reconstruction Loss

The image reconstruction loss attempts to recreate the input image by averaging the produced frames and overlaying them on the provided background image.

Time-Consistency Loss

The time-consistency loss tries to enforce a consistent appearance of the frame by penalizing large differences between consecutive frames.

Latent Loss

The latent loss is applied directly to the latent representation. Specifically, the same blur is overlaid on two different backgrounds, then their respective latent representations are compared. Ideally, the latent representations should be equal, since they should only be encoding the blur and not the background.

Sharpness Loss

Finally, the mask of the rendered frames should be sharp in the sense that there should not be any semi-transparent regions, only fully transparent or fully opaque regions.

Chapter 3

Building on DeFMO

In order to achieve the goal of exploring new approaches within the framework of DeFMO, the existing codebase needs to be reimplemented in a modular, extensible way that facilitates experimentation.

DeFMO is a complex system involving multiple components, such as the data loaders, the encoder, the renderer, and the various loss functions. Each of these parts should be easily swappable and configurable, without requiring code changes that break earlier experiments.

Additionally, the results and checkpoints of individual experiments should be automatically logged and saved, such that they can be compared and training can be resumed for promising experiments.

3.1 Synthetic Dataset Generation

Before experiments can be run, the synthetic dataset needs to be generated.

A single data point should consist of an input and ground truth pair, e.g. an image of a blurred object following some trajectory overlaid on some background, the corresponding high-speed frames, and the plain background for reference.

Goals

The generated dataset should fulfill the following criteria:

- Compact: the dataset should be minimized in file size and file count while retaining a reasonable quality.
- Reusable/flexible: the dataset should provide the ability to run varied experiments without requiring the generation of a new dataset for each new approach.
- Fast: generating the dataset should be reasonably fast.
- Self-documenting: the parameters and inputs used to generate the dataset should be stored with the data for future reference.

Procedure

The dataset is generated using the free and open-source 3D modeling and rendering software Blender [5], which provides a Python API for automating such tasks.

Blender is set up for rendering with the camera at the origin with no rotation and environment lighting. This static setup allows the object's trajectory to be generated in such a way that it is guaranteed to be in frame for all of it.

The steps for generating a datapoint are:

1. Sample a 3D model from a collection of objects - we use the ShapeNetCore.v2 [2] collection.

   This collection features over 50'000 3D models in 55 categories, such as furniture, vehicles, and household objects. The models are approximately normalized in size to a diameter of 1.

2. Sample a texture to be applied to the object - we use the Describable Textures Dataset [3].

3. Sample a random trajectory and rotation within given bounds to describe the object's movement.

   The start and endpoint are generated to lie within the viewing frustum of the camera with some room to spare around the edges. A minimum and a maximum travel distance in the X/Y plane can be specified, as well as a maximum Z difference, representing movement towards and away from the camera.

   The starting rotation is set randomly and the ending rotation is set as a small random offset from the starting value.

   These points and rotations are set as keyframes in Blender, which then interpolates the values for each frame.

4. Render sharp, "high-speed" images of the object at equally-spaced intervals along the position and rotation trajectory.

5. Additionally render the object as an FMO by adding motion blur to its movement during the given path.

   Blender provides motion blur as an option with a configurable number of sub-steps.

6. Check if the produced result satisfies our requirements, otherwise discard it and proceed to the next object.

   We impose a minimum opacity/coverage of the FMO. If for example the object is very small and is blurred over a long trajectory, it becomes nearly invisible, making it of little use later on.

The data is rendered using the EEVEE engine provided by Blender, which is a GPU-based rasterization engine providing very fast render times.

The fully random nature of the parameter selection above also means that distributing the dataset generation over multiple processes or machines is as simple as starting the rendering process multiple times, since no synchronization is required. The outputs of each worker can then easily be merged.

Output Format

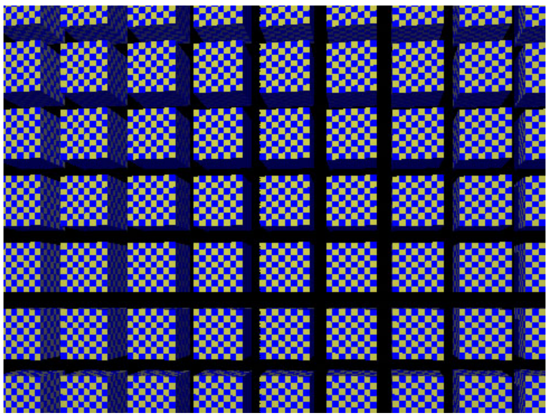
By default, Blender will output PNG images of the rendered scenes, which is a common image format that supports the required transparency information. However, PNG only supports lossless compression, which leads to larger file sizes than desired. A complete dataset of 50'000 - 100'000 datapoints, each consisting of 25 - 30 images, would require hundreds of gigabytes.

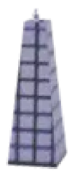
The most common alternative format is JPEG, which supports lossy compression but not transparency.

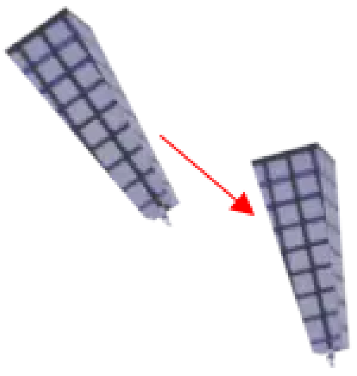
To combine both transparency and lossy compression, the newer WebP [7] image format is used. This format was introduced in 2010 by Google and claims 25-34% smaller file sizes than JPEG at equal SSIM [21] quality measure.

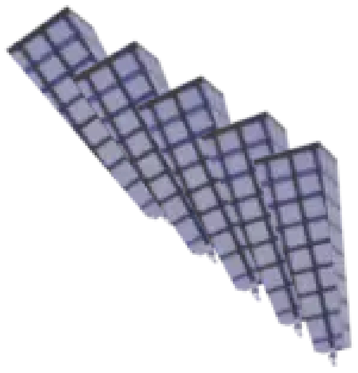
As an added bonus, WebP supports animations, allowing the individual rendered frames of each datapoint to be stored in a single file. This not only drastically reduces the number of files to be

(a) sample a random object


(b) sample a random texture


(c) apply texture to object


(d) sample random start and endpoints and rotations within bounds


(e) render linearly interpolated frames

(f) render FMO blur

Figure 3.1: Synthetic dataset generation procedure

managed, but also allows the datapoints to be easily visualized as an animation in any modern web browser.

Using WebP in this way reduces the space requirement for a datapoint of 27 images at 320x240 pixels from 150-250kB to just 40kB on average, leading to a total dataset size of 2-4GB.

Dataset Container

The individual rendered sequences could be stored as files, but packing them into a single container file simplifies managing the dataset, such as when copying or listing contents. It is also beneficial on systems where maximum file count may be limited and enables metadata to be stored with the image sequences.

Since this use case is fairly simple, a ZIP file is sufficient. ZIP files provide a file list, thereby allowing random access, and feature "comment" fields at the root level and for each contained file, where we store the metadata in the form of a JSON string.

This data stores the render parameters, such as resolution, number of frames, and trajectory settings; and which 3D model and texture were used for each individual sequence.

What to include?

A complete input to an FMO deblurring system requires an FMO rendered on a background. This background could be added in the dataset generation step, but this would lead to less flexibility later on. Instead, the rendered blurred FMOs can be stored without a background and with transparency information, allowing it to be superimposed on a background on-demand at training time.

This gives the flexibility to add multiple FMOs to one image or to choose different backgrounds for every epoch, which additionally reduces the chance of overfitting.

The FMO itself could even be rendered at training time by averaging the individual frames, but pre-rendering these allows a larger value to be chosen for the number of motion blur steps than the number of high-speed frames, leading to a smoother result. In the following experiments, 24 high-speed frames are used but the FMOs are generated with 40 motion blur steps.

Motion blur can of course still be rendered at training time if desired, for example if something unusual is required, such as breaking the FMOs trajectory into multiple sub-blurs.

3.2 Neural Network Training

The neural network is built and trained using the open-source machine learning library PyTorch [15]. Besides providing all the building blocks required to create deep neural networks, PyTorch features automatic differentiation, which means that the user does not need to manually define the gradient calculations required for backpropagation, but can freely perform operations during the forward pass, which are recorded by PyTorch and used to automatically calculate the required gradients.

While gradients are handled automatically, PyTorch leaves a lot of the logistics of training a neural network to the user, such as defining data loaders and the training loop, handling which device models and tensors reside on, distributing training over multiple GPUs or machines, and logging statistics and results. This means pure PyTorch will require a lot of boilerplate code, which can also be error-prone.

Instead, we use PyTorch Lightning [6], which is a framework built on top of PyTorch with the goal of "decoupling research from engineering", that is to handle the standard engineering concerns mentioned above and allow the user to focus on developing and experimenting with their machine learning models. PyTorch Lightning provides a standard way of defining a machine learning system, but provides ways of overriding nearly every aspect of the training and validation process should it be required.

Goals

The architecture of the training system should fulfill the following quality-of-life goals:

- Automated logging.

  The system should automatically log training progress, validation metrics, benchmark scores, and prediction visualizations in an easily accessible way, without first requiring parsing of log files or other manual efforts.

- Automated checkpointing.

  Checkpoints containing all model and training information such as parameter weights, training progress, and hyperparameter configuration, should automatically be created at regular intervals during training. These checkpoints should capture all the information required to resume training.

- Configurable from the command line.

  All the relevant aspects of a training run, such as which dataset, model, and loss functions to use, should be configurable from the command line used to start the training. In particular, setting these parameters should not require code changes.

  The idea is to simplify managing training runs, especially when using a compute cluster. Making all aspects of training configurable from the command line makes parallel runs and queuing runs from the same code base trivial.

- Modular and extensible design.

  Extending the system, such as by adding a new loss function, should not require code changes beyond adding the actual code of the loss function in one location. Recurring code, such as registering and logging the output values of loss functions, should not need to be duplicated.

- Easily distributable.

  The system should be easily runnable using any number of GPUs or even multiple machines to accelerate training.

The use of the PyTorch Lightning framework already brings us significantly closer to achieving these goals, as logging, checkpointing, and distributed learning are already provided for the most part.

Architecture

The system can roughly be divided into four areas of concern:

- Data - loading and preparing the data to be used for training, validation, and possibly testing.
- Model - defining the neural network architecture and the associated loss functions.
- Trainer - handling the training and validation of the model on the provided data.
- Callbacks - all non-essential code, such as supplementary logging, should be defined in callbacks according to the PyTorch Lightning system.

3.2.1 Data

PyTorch Lightning defines the *LightningDataModule* class to act as a standard interface for accessing a dataset's training, validation, and prediction data loaders. The datasets are defined by the user, then PyTorch and PyTorch Lightning handle batching, shuffling, and distributing these datasets automatically.

Datasets

The dataset calls the following sub-modules:

- An FMO loader.

  This class acts as an interface to a dataset of synthetic FMO and ground truth data generated as described previously. It handles opening and loading the dataset, as well as parsing the metadata. It then provides an interface with which any data point within the dataset can be accessed by its index, returning the FMO blurs and the ground truth frames as lists of Pillow [4] images - a common Python image library.

  The FMO loader also supplies options for filtering, such as restricting the range of returned data points, which is useful for train/validation splits.

- A background adder.

  The FMO blurs from the dataset do not yet have a background. These are added dynamically by the data loader at training time. This results in a lot of flexibility when defining datasets, such as the freedom to add multiple blurs to one image or return the same blur with multiple backgrounds for contrastive learning if desired.

  The background adder is configured with a dataset of background videos, from which it will randomly select a consecutive sequence of frames when it is called from the main dataset class with a corresponding sequence of blurs. A minimum contrast requirement between the FMO and the background can also be specified, such that the background adder will continue to select random background sequences until it finds one that satisfies the contrast requirement, or until some maximum number of tries is reached, at which point it will choose the one with the best contrast so far.

These modules have simple APIs and can easily be exchanged for other versions, such as if a new FMO dataset is used with a different structure. The dataset return requested batches as a dictionary of values (containing images, blurs, frames, backgrounds, parameters, etc.), which can thus later be referenced by name, making it simple to add new components to be returned without breaking anything down the line.

3.2.2 Model

The encoder and renderer classes are PyTorch modules, which are set up to be easily extensible with new model architectures. The desired model can be selected by name when the class is instantiated.

These modules handle any special input requirements independently, such as restructuring multiple input images into stacked channels or augmenting the input with a number of timestamps, which keeps this specialized model-dependent code out of the main training logic.

Any relevant outputs produced by the network, such as the renders of high-speed frames, latent representation, or reconstructed blurs, are collected in an output dictionary. This means that all loss functions can now be defined with a simple and consistent API: simply pass the input dictionary and the output dictionary to the loss function, which can then use what it needs from each. For example, a supervised loss might compare the frames in the input dictionary to the renders in the output dictionary, while a reconstruction loss would overlay the blurs from the output dictionary onto the backgrounds from the input dictionary and compare the result to the images in the input dictionary.

The loss functions themselves are also PyTorch modules, which use inheritance to avoid duplicating common code. For example, the *BaseLoss*, which is a superclass of all loss functions, handles weighting the loss and generating a string representation when required. The *SupervisedBase* then inherits from the *BaseLoss* to provide the methods and handling common to all supervised loss functions.

This approach makes prototyping and experimenting with new loss function variations extremely simple: just add a new class that inherits from the most appropriate base. Any defined loss function can then be selected by name for training. Any required parameters can also be passed in the same string and will be parsed to keyword arguments automatically.

A separate list of comparison losses can also be defined, the values of which will be calculated and logged, but not used for backpropagation. This is useful for defining some loss functions that serve as a baseline for comparison between runs using different loss functions.

3.2.3 Trainer

The *LightningModule* is the core of the training system when using PyTorch Lightning. Its behavior is configured by overriding its methods. By default, only methods defining the training and validation steps and for configuring the optimizer need to be overridden, but many others are available to customize the training process.

The main trainer can be configured using Python's argument parser, which we extend to include all configuration options for selecting the model and loss functions, as well as to correctly reload all the relevant configuration options from a checkpoint file if one is available. This allows a training run of many epochs to be spread over multiple program launches, which is beneficial when using a computing cluster, as it allows shorter tasks of only a few epochs to be submitted to the faster task queues.

3.2.4 Callbacks

Callbacks in PyTorch Lightning are intended to contain all optional or non-essential code, such as logging or running tests. Every callback provides 'hooks', which can be overridden to specify when in the training process the callback should be run.

Continuous Model Checkpointing

PyTorch Lightning has model checkpointing built-in as a standard callback, which can monitor a logged metric and create a checkpoint whenever the metric improves. Saving after the last epoch of a training run is also available, but when training is resumed from this checkpoint, the metric tracked for the best model is not restored correctly.

This callback is an extension of the default checkpoint callback that fixes this issue.

Prediction Logging

It is extremely helpful to have renders of results available for visual evaluation of training progress. This callback is called after every validation epoch and creates renders of a few validation samples, which are then logged directly to TensorBoard. The created videos feature the ground-truth animation, the output of the renderer network, and the alpha mask determined by the network.

This creates a browsable timeline of validation renders that visually illustrate the training progress.

Benchmark Logging

The authors of DeFMO [16] published the code used to benchmark their results on real-world data. This benchmark was adapted for this work, extending it to enable it to accept the double blur experiments tested here. This callback runs the benchmark after each validation epoch to get a comparison value for how training is progressing when measured on real-world data.

Chapter 4

Experiments and Results

4.1 Group Norm

While the neural network architecture used in this work is mostly unchanged from the original DeFMO formulation, one key change was made to the normalization layers. In general, normalization layers are used because they increase the speed and stability of training, although the exact reason they have this effect is not well-understood, with multiple competing theories having been presented [17].

The most common type of normalization layer used in many modern networks including ResNet is batch normalization [9], which normalizes the inputs to a layer such that every channel has zero mean and unit variance across the spatial and batch dimensions. The normalized inputs are then shifted and scaled again to a new mean and variance based on batch statistics learned during training. During training, batch normalization will use statistics from the current batch to adjust these learned parameters, while during validation it will only use the learned parameters.

The performance of batch normalization during validation therefore depends on the accuracy of the learned batch statistics, which means that the error rate of the network can increase drastically if the batch size is too small to learn accurate statistics [23]. Small batch sizes are unfortunately common in deep neural networks for image-based tasks since both the models and the inputs tend to be comparatively large, quickly reaching the memory limits of the GPUs the networks are trained on.

The experiments in this chapter typically use a batch size of only 3 or 4 due to these constraints, which led to issues with validation error as is illustrated in Figure 4.1. Specifically, the validation error started to increase after the point of "animation", which is the moment during training where the network learns the importance of the timestamp specified for each rendered frame and starts producing distinct frames, leading to an animated rendering of the FMO rather than a static image. Figure 4.2 shows what renders should look like before and after animation, without the issues caused by batch normalization. The training error on the other hand would decrease sharply at this point, as the individual frames would better match their ground truth.

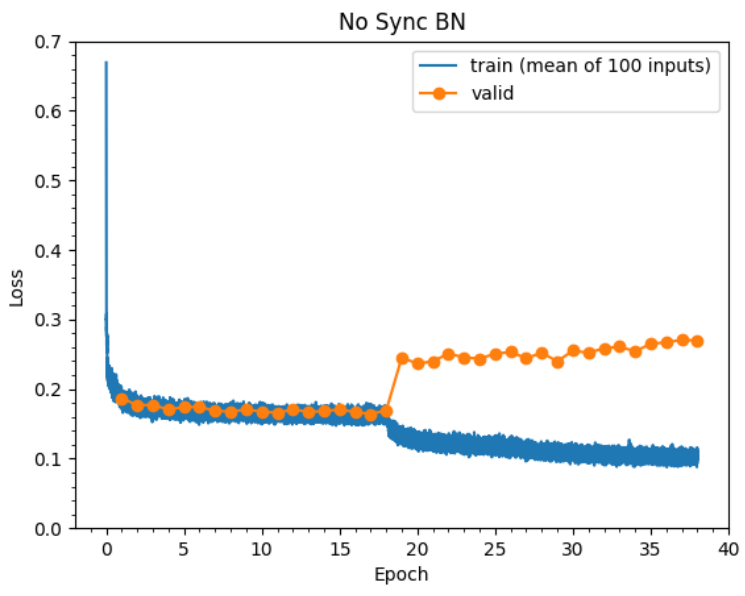
Three possible solutions for small batch sizes are:

- Increase batch size.

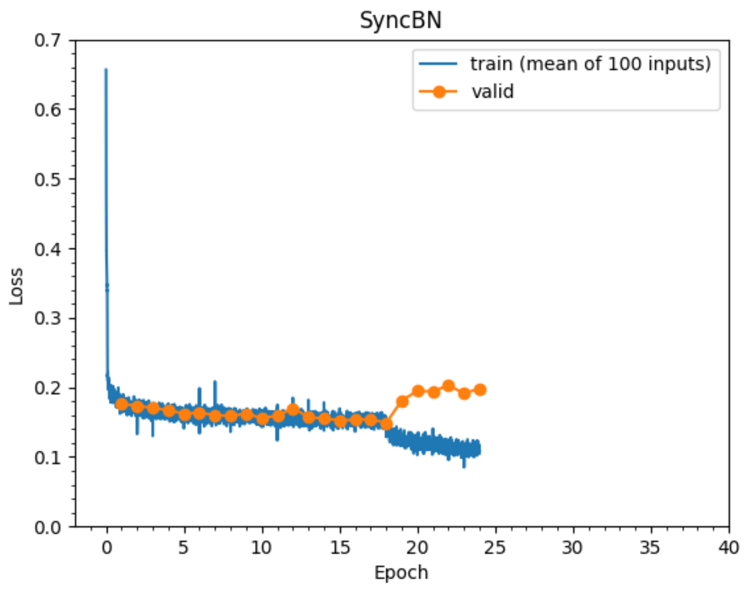
  This is not easily possible due to the memory limits of the GPUs used for training.

- Increase "effective" batch size for batch normalization.

  Training is done on multiple GPUs - 8 for these experiments - each of which performs the forward and backward steps independently on separate data. The calculated gradients are then aggregated by the main process and the new parameter weights are redistributed after the optimization step. A similar system could be used to synchronize the batch statistics used



(a) without synchronization across GPUs



(b) with synchronization across GPUs

Figure 4.1: Evolution of training and validation loss progression during training, with and without batch norm synchronization across GPUs. The sharp drop in training loss around epoch 18 marks the beginning of "animation", i.e. the moment when renders for different timestamps start to diverge.

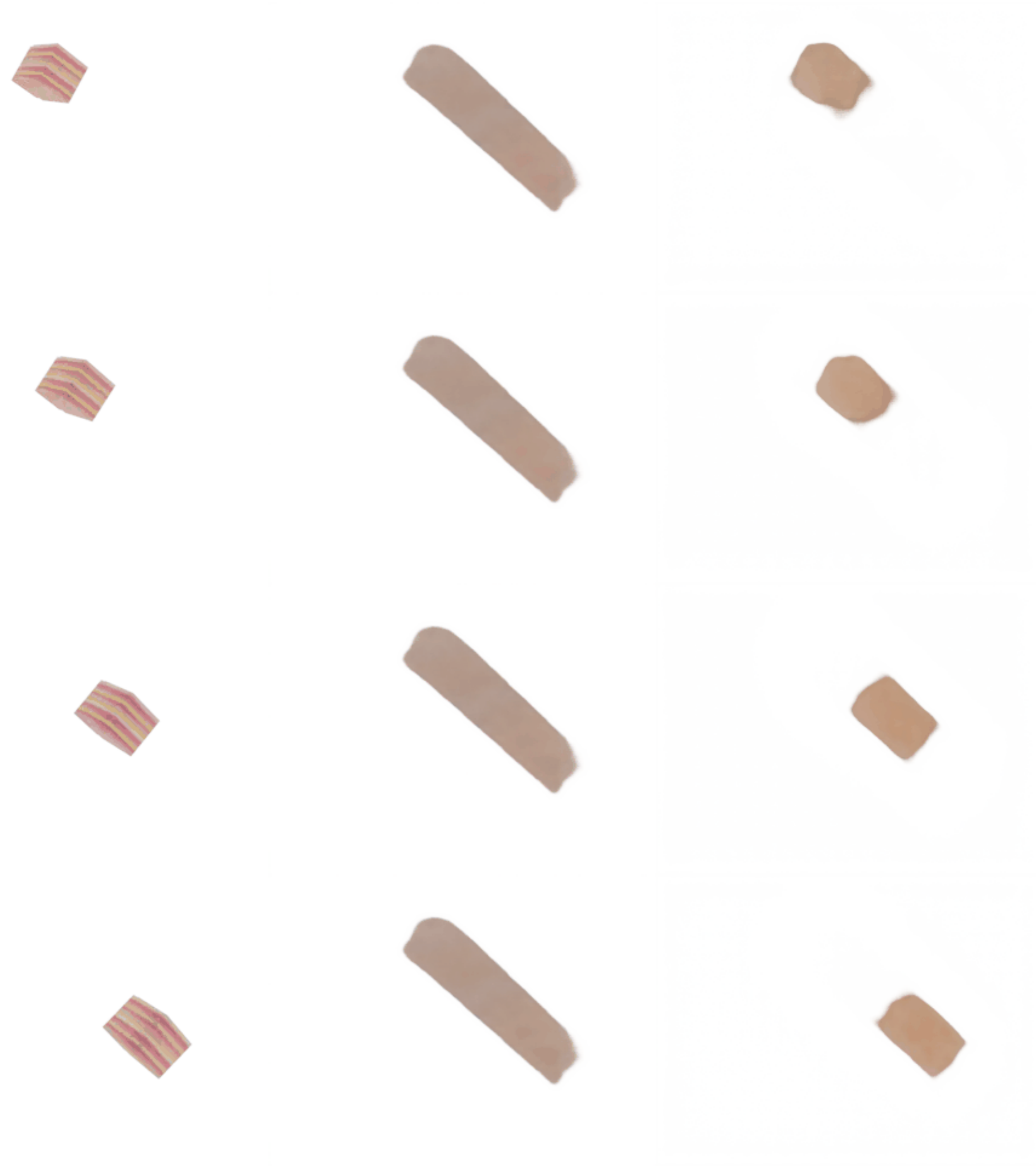


Figure 4.2: Renders of a validation sample. The columns are the ground-truth frames, the rendered frames before the network learns to pay attention to the added timestamp ("animation"), and the rendered frames shortly after animation. This network was trained with group normalization, thus exhibiting proper animation during validation.

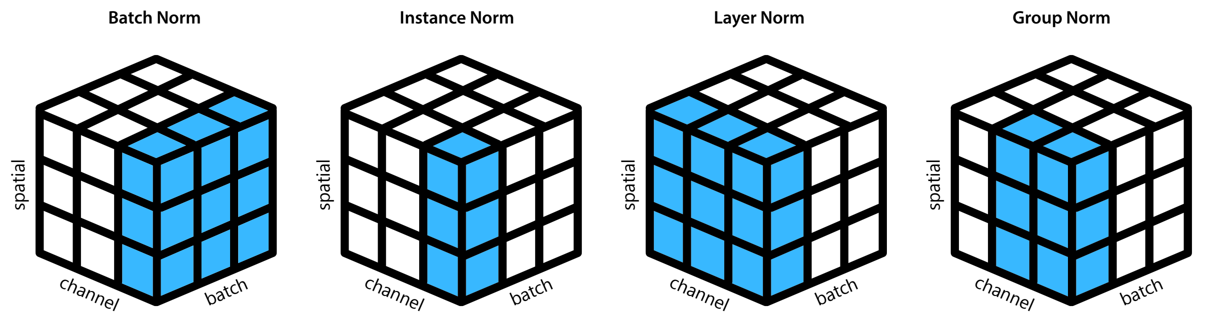


Figure 4.3: Illustration of different normalization methods, showing which dimensions each acts on. Light blue represents values that are normalized together. Recreated from [23].

for batch normalization, which would lead the statistics to be based on a batch size of 24 in the example of 8 GPUs with a batch size of 3 each.

Both PyTorch and PyTorch Lightning have built-in methods that purport to perform this synchronization. Both were tested under multiple circumstances, but unfortunately, the problem persisted as illustrated in Figure 4.1.

- Use an alternative normalization method that is not affected by the batch size.

  There are three dimensions to consider for normalization: the batch dimension, the channels, and the spatial dimension. Batch Norm normalizes all values in the batch and spatial dimensions but acts separately on each channel. The scope of values that are normalized could be reduced by also acting separately on all elements of the batch dimension - this is referred to as Instance Norm [19]. Alternatively, elements of the batch dimension could be normalized separately while combining all channels - this is referred to as Layer Norm [1].

  See Figure 4.3 for a visual comparison of these normalization methods.

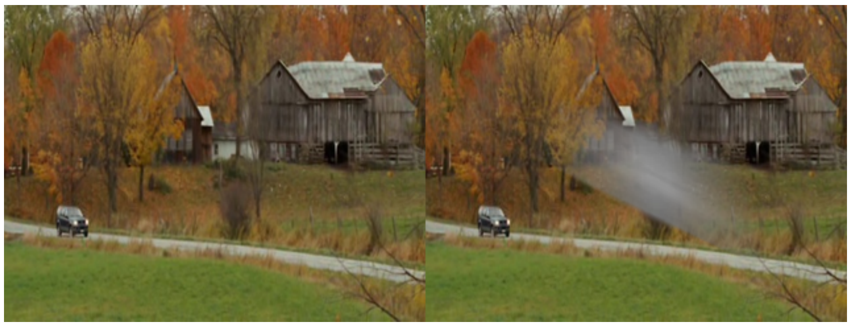
  Unfortunately, both Instance and Layer Norm perform significantly worse than Batch Norm on image-related tasks [23], but a compromise between the two, known as Group Norm [23], can achieve almost equal performance. Group Norm also acts separately on the elements of the batch dimension and is thereby independent of batch size, but rather than normalizing all channels together like Layer Norm or all channels separately like Instance Norm, it forms groups of channels that are normalized together.

Group Norm is used in the following experiments. The pre-trained ResNet-50 encoder is maintained, but the Batch Norm layers are exchanged with Group Norm layers with 32 groups, which was found to be the optimal number of groups in [23]. The renderer also uses Group Norm with 32 groups, except for the bottleneck layers with only 64 and 16 channels, where 8 and 4 groups are used respectively.

This change solves the issue of the network not managing to predict animations in evaluation mode. Additionally, it increases convergence speed, in the sense that the animation point happens sooner during training - when using the original DeFMO supervised loss function for a double blur task as described in the next section, it happens around epoch 7 rather than 18.

4.2 Double Blur

The main experimental focus of this work is on the "double blur" approach, which replaces the input with two consecutive frames of the FMO, rather than the original background/FMO pair. The


(a) original DeFMO input: background with and without FMO

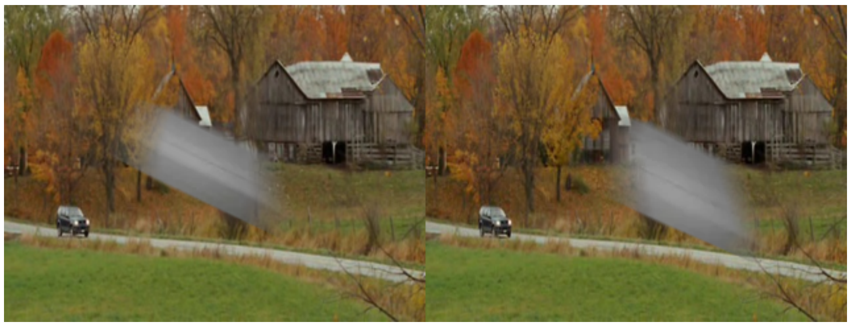

(b) "double blur" input: two consecutive FMO frames each containing "half" of the blur, on two consecutive frames of the background video

Figure 4.4: Input comparison between the original background/blur approach and the "double blur" approach.

background for these two consecutive FMO frames is also two consecutive frames from a randomly sampled video, thus recreating the scenario of an FMO having been captured on video as accurately as possible. See Figure 4.4.

The main rationale for this approach is to:

- eliminate the need to provide an explicit background since this will usually not be available in practice and must instead be estimated from multiple frames, thus increasing the total number of required frames.
- disambiguate the direction of motion.

It is to be expected that this formulation leads to a more difficult training task, as the foreground/background segmentation is no longer clearly defined by the input images, especially in the region of overlap between the two half-blurs.

The first basic experiment is to compare the new input 1:1 to the old input format by running training tasks using the original supervised loss as defined by DeFMO. This loss function is a combination of three equally-weighted L1 losses: a mask loss averaged outside the ground-truth object's area, an RGB loss inside, and a mask loss inside.

For the background/blur input, the loss function will be applied twice, once with the ground-truth frames in the original order and once in the reversed order, since the correct order cannot be determined. Only the correct order as determined by the smaller loss value will be used for training and validation. As a curiosity, not performing this extra step and simply applying the loss function directly results in the rendered frames containing two copies of the object, traveling in both possible directions. A visual comparison of results from the different methods can be seen in Figure 4.5.

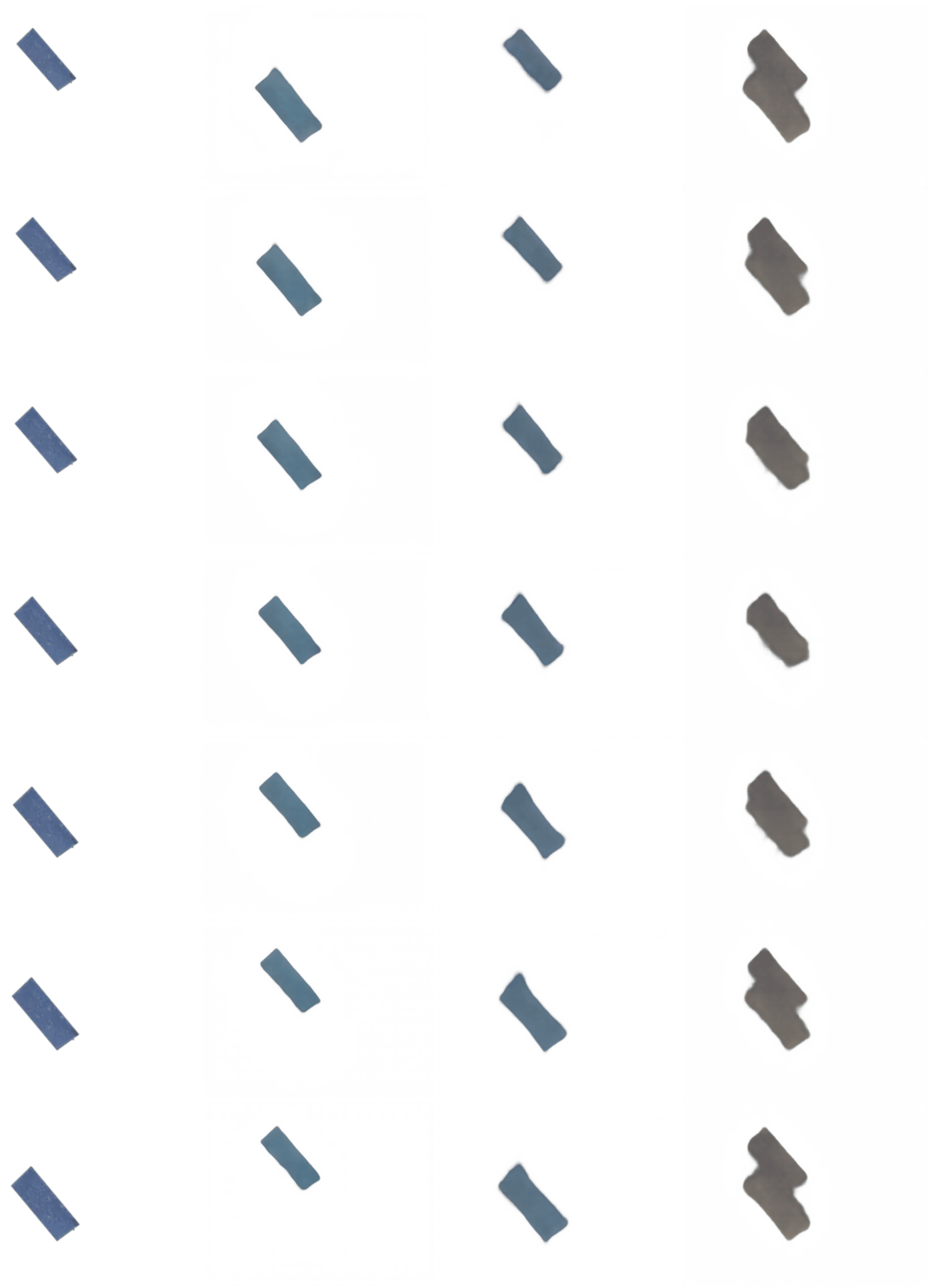
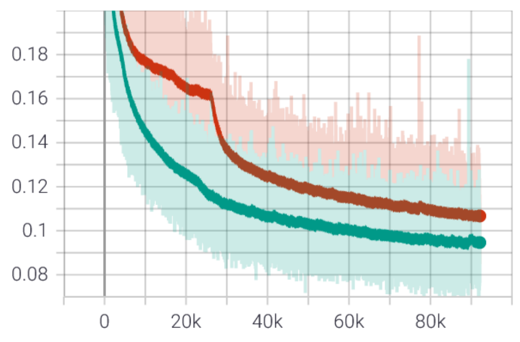
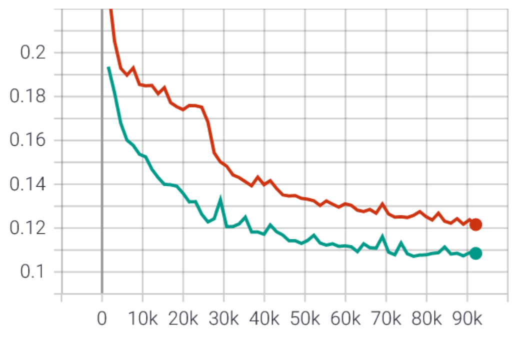


Figure 4.5: Renders of a validation sample. The columns are the ground-truth frames, renders using background/blur input, renders using double blur input, renders using background/blur input without synchronizing direction of the loss function. The background/blur version has a higher quality mask than the double blur version at equal training time, but the direction is incorrect.



(a) training loss (smoothed)



(b) validation loss

Figure 4.6: Loss evolution during training of background/blur input (green) vs double blur input (red) using the original DeFMO supervised loss. Each training step on the X-axis represents a batch of 24 inputs.

The evolution of the value of this loss function can be seen in Figure 4.6. As expected, the double blur is slower to converge and also features a clearly visible "animation point" around training step 27k, going from identical rendered frames to diverging frames forming an animation, as illustrated in Figure 4.2. The background/blur input on the other hand converges smoothly with animation already occurring at training step 4k, as determined by validation renders.

4.3 Alpha and RGB Losses

The FMO learning task consists of two parts: learning the outline and shape of the FMO, represented by the alpha layer of the renders, and learning the color and texture, represented by the RGB layers of the renders. Since any RGB information outside of the masked area will be invisible, correctly learning and evaluating the RGB layers very much depends on the mask being accurate.

This section treats the alpha and RGB layers as two separate learning tasks and explores how their interaction during training can be improved through tweaking the formulation of their respective loss functions.

To recap, while the RGB layers encode the red, green and blue components of a pixel, the alpha layer encodes its opacity, from fully transparent to fully opaque. Compositing an image with an alpha layer $[\mathrm{RGB}_{img},\ \alpha]$ onto a background $[\mathrm{RGB}_{bg}]$ is done as follows:

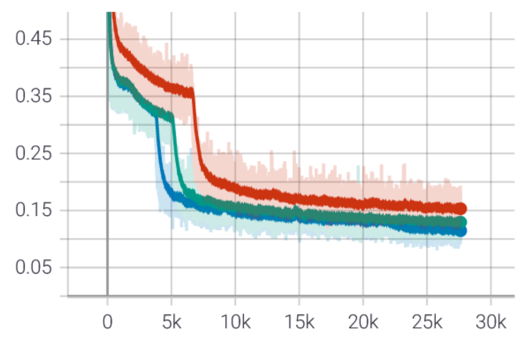
$$\mathrm{RGB}_{comp} = \mathrm{RGB}_{img} \cdot \alpha + \mathrm{RGB}_{bg} \cdot (1 - \alpha)$$

Similarly, to be able to correctly compare pixel values between images, their RGB values must first be multiplied by their alpha values to get the correct result, since non-zero RGB values are meaningless where the alpha value is zero.

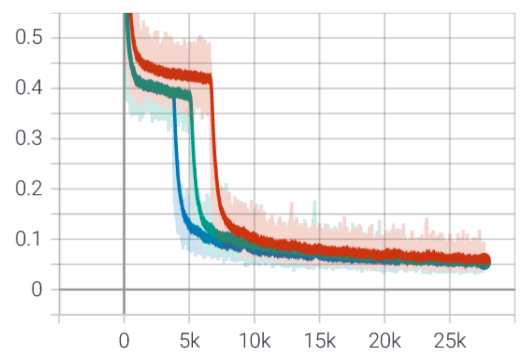
Supervised L1 losses comparing the renders $[\mathrm{RGB}_{re},\ \alpha_{re}]$ to the ground truth $[\mathrm{RGB}_{gt},\ \alpha_{gt}]$ for RGB and alpha layers respectively would therefore be calculated as follows:

$$\begin{aligned} \text{RGB loss:} \quad & |\mathrm{RGB}_{re} \cdot \alpha_{re} - \mathrm{RGB}_{gt} \cdot \alpha_{gt}| \\ \text{alpha loss:} \quad & |\alpha_{re} - \alpha_{gt}| \end{aligned}$$

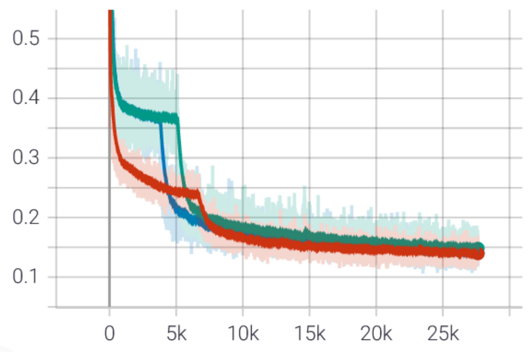
where the per-pixel RGB loss values are aggregated using a weighted average based on the α_{gt} value since the RGB values are irrelevant where the alpha value is zero.



(a) overall training loss



(b) alpha loss



(c) original DeFMO supervised loss for comparison

Figure 4.7: Loss evolution during training using a fixed Dice loss for the alpha layer but varying the mode of the RGB L1 loss: standard (red), RGB-Direct (green), RGB-GT (blue). Each training step on the X-axis represents a batch of 24 inputs.

However, this creates an unwelcome coupling, as despite splitting the RGB and alpha losses, the RGB loss still depends on the predicted alpha values. If for example the RGB loss determines that the RGB values are too high or too low, this information would also backpropagate to decrease or increase the alpha value respectively, which would hinder the correct learning of the mask as determined by the alpha loss. Conversely, an imperfect mask can hinder RGB learning, as an incorrect zero-value in the mask would stop the backpropagation of RGB information.

The two considered alternatives are:

$$\begin{aligned} \text{RGB-Direct loss:} \quad & |\mathrm{RGB}_{re} - \mathrm{RGB}_{gt}| \\ \text{RGB-GT loss:} \quad & |\mathrm{RGB}_{re} \cdot \alpha_{gt} - \mathrm{RGB}_{gt} \cdot \alpha_{gt}| \end{aligned}$$

each still averaged based on α_{gt}, as before.

These two formulations only differ where the ground-truth alpha value is non-binary, so while the RGB-Direct loss is the most natural choice, the RGB-GT loss might help by further reducing the weight of RGB differences in pixels where the alpha value is low, although this should mostly already be accounted for by the averaging process.

The results of training with each mode of RGB loss while keeping the alpha loss fixed can be seen in Figure 4.7. Focusing just on the alpha loss, using the RGB-GT mode shows clear improvement in the initial convergence speed with the sharp drop corresponding to the animation moment happening in around half the number of training steps. Comparing with the original DeFMO supervised loss

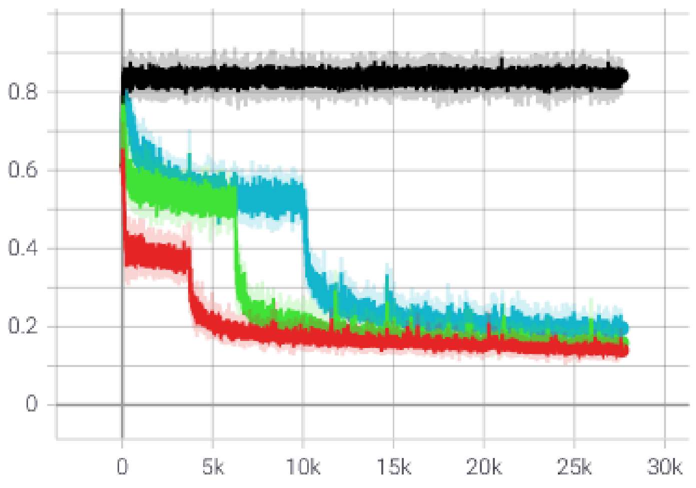


Figure 4.8: Loss evolution of original DeFMO supervised loss function to act as comparison for networks trained with L2 RGB-Direct and different alpha loss functions: Dice loss (red), binary cross-entropy (green), L2 (blue) and **L1 (black)**, which did not escape the all-zero local minimum.

shows the standard RGB mode performing better at first, which is to be expected since the original DeFMO supervised loss itself uses the standard mode of RGB loss, so this metric would naturally prefer optimization toward the most similar objective.

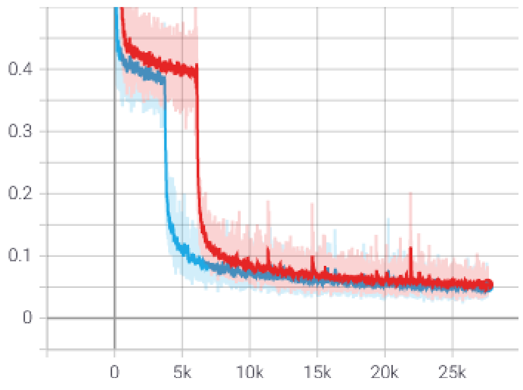
The accelerated convergence when using the Direct or GT modes is even more pronounced for alpha losses that exhibit slower convergence. The Dice loss [14] used in Figure 4.7 was chosen as the alpha loss for this and later experiments for its fast convergence. For other tested alpha loss functions, such as binary cross-entropy or a simple L2 loss, the difference between standard and RGB-Direct modes are very significant, as seen in Figure 4.9.

A pure L1 alpha loss was also tested but failed to escape the local minimum of all-zero alpha values. Since the area covered by an FMO frame can be very small, sometimes as low as 1%, predicting the alpha layer correctly involves very unbalanced "classes", which would need to be balanced for L1 to succeed, such as by considering the area within and outside of the FMO separately, as was done by the original DeFMO formulation.

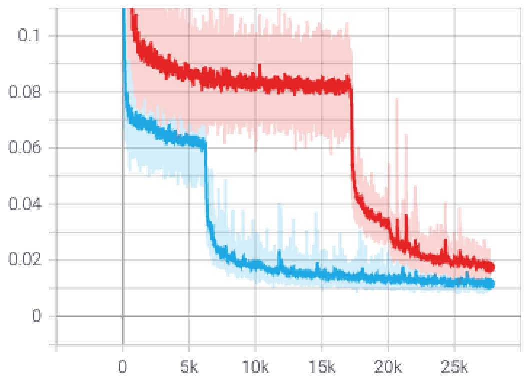
Convergence behaviour of various alpha loss functions with fixed L2 RGB-Direct loss as measured by the original DeFMO supervised loss can be seen in Figure 4.8. The exact formulations for these loss functions are as follows, where x_i and y_i are the values of individual pixels whose losses will be aggregated:

L1:	$\lvert x_i - y_i \rvert$
L2:	$(x_i - y_i)^2$
binary cross-entropy:	$y_i \cdot \log x_i + (1 - y_i) \cdot \log(1 - x_i)$
Dice loss:	$\frac{2 \cdot \sum_i^N x_i y_i}{\sum_i^N x_i^2 + \sum_i^N y_i^2}$

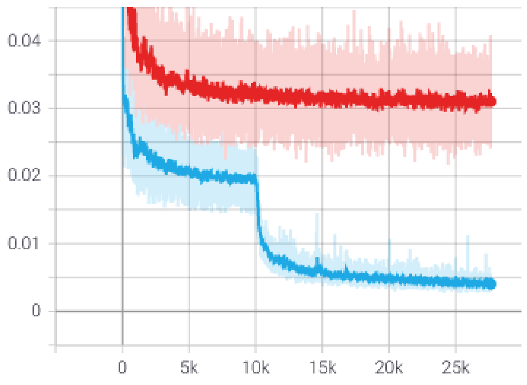
In conclusion, this separation of RGB and alpha objectives can lead to faster convergence, especially early in training. However, it appears that the final outcome will still mostly converge to a reasonable value if given enough time. Faster convergence is still a huge benefit for experimentation



(a) Dice loss



(b) binary cross-entropy loss



(c) L2 loss

Figure 4.9: Difference in convergence speeds of alpha losses with L2 RGB loss in standard (red) and RGB-Direct (blue) modes. The L2 alpha loss in standard mode did not animate within the given 28k training steps.

though. Usually, when experimenting with machine learning, comparisons of techniques and ideas will be done with a reduced dataset and short training time, to find out what looks promising before starting long and resource-intensive training runs with the full dataset. Such an approach was not possible here given the complexity of learning for this task. Only running a few epochs previously would not even lead to animated frames yet. With the shorter convergence times, shorter tests can be run to experiment with an idea.

To put the difference it can make in perspective: with the loss functions used for the figures in this section, 3000 training steps - each a batch of 24 - take approximately one hour to run on a cluster using 8 GPUs.

4.4 Multi-Object Double Blur

An interesting extension of the FMO deblurring task is to allow multiple FMOs in a single image. This goal practically requires that the direction of motion be known for each object, such as by using the double blur method, since the number of possible combinations of motion directions increases exponentially with the number of objects.

For this task, the data loader is configured to select at random from one to three the number of FMOs to add to each training and validation image. The setup is otherwise the same as for a standard double blur experiment. Dice loss is used for the alpha layer and L1 is RGB-GT mode for the color layers. All renders are on FMOs from the left-out validation set with unseen backgrounds from the OTB dataset [22].

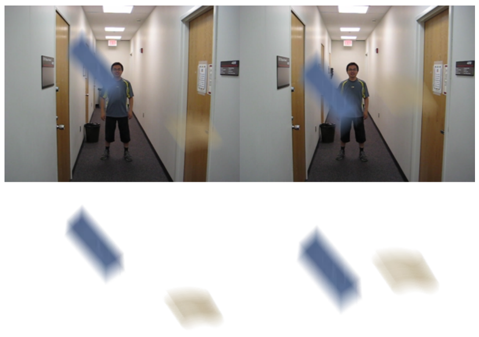
"Easy", "medium" and "hard" multi-object examples are illustrated in figures 4.10, 4.11 and 4.12. For each, the output produced by the network trained on multiple FMOs is compared to the output of a network trained only on a single FMO.

The easy example in Figure 4.10 shows that if the FMOs in the input are sufficiently separated, then even the single-object network can create separation in the rendered outputs. However, it can be seen that the single-object network will create artifacts between the FMOs, attempting to join them. In the medium and hard examples, where there is not always clear separation and even some overlap, the single-object network will almost always create a single contiguous blob, matching the expectation based on what this network has seen.

A second, less obvious effect that can be observed concerns the color. The textures in the dataset have varying colors, but any individual texture will usually be fairly uniform in color, especially at the scale at which they are applied to the objects and even more so after blurring. It appears that the single-object network therefore does not expect color to vary strongly within the FMOs of a single image.

In Figure 4.10, the two objects have clearly distinct colors: blue and beige, but while the single-object network correctly colors the larger blue object, the smaller beige object is given a neutral blue-gray color. This effect is even more pronounced in the more complex examples, where the color assignment by the single-object network fails almost completely due to the varying colors in the input FMOs.

The multi-object network on the other hand fares very well in all three examples, even keeping a relatively clean color separation during complex overlapping sequences. Note that these results are based solely on a very basic alpha and RGB supervised loss and could undoubtedly be improved through further training, since the validation losses do not seem to have fully plateaued yet, as seen in Figure 4.13, or through the addition of further specialized loss functions, as was done for the original DeFMO system.


(a) Input to the network (top) and input without background for comparison.

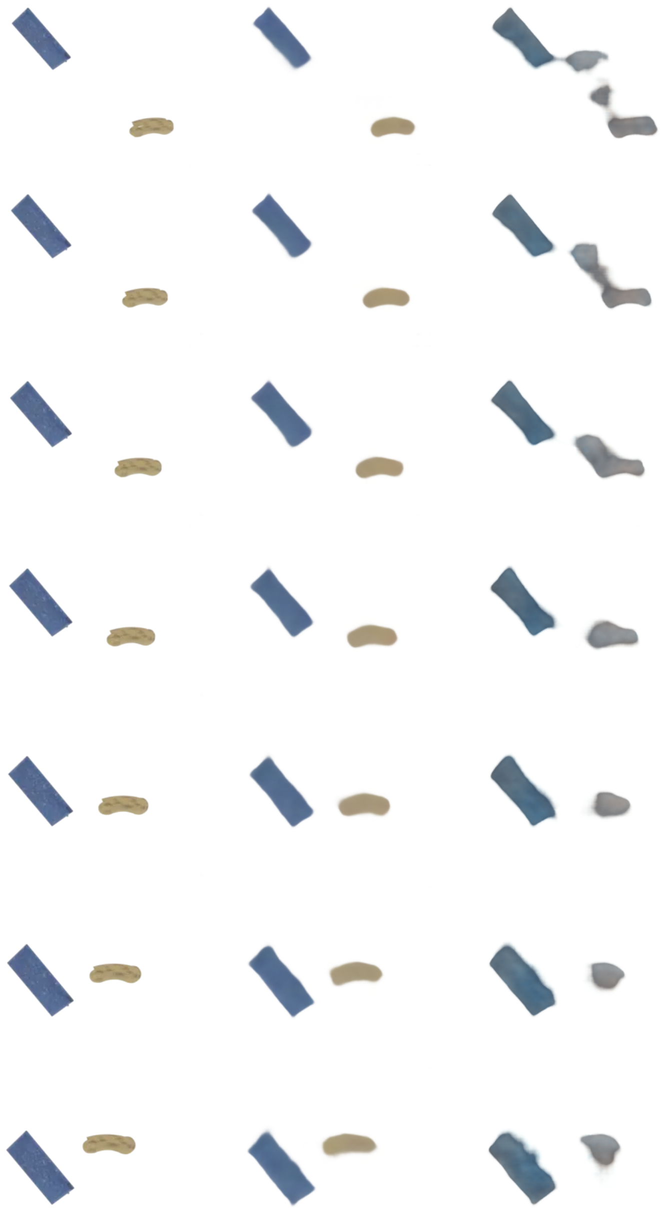
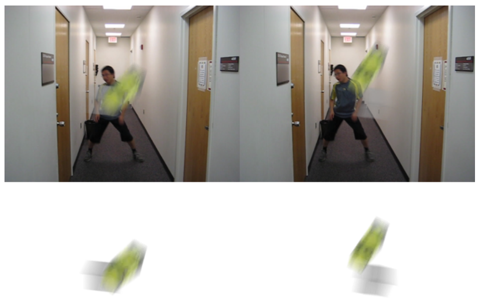
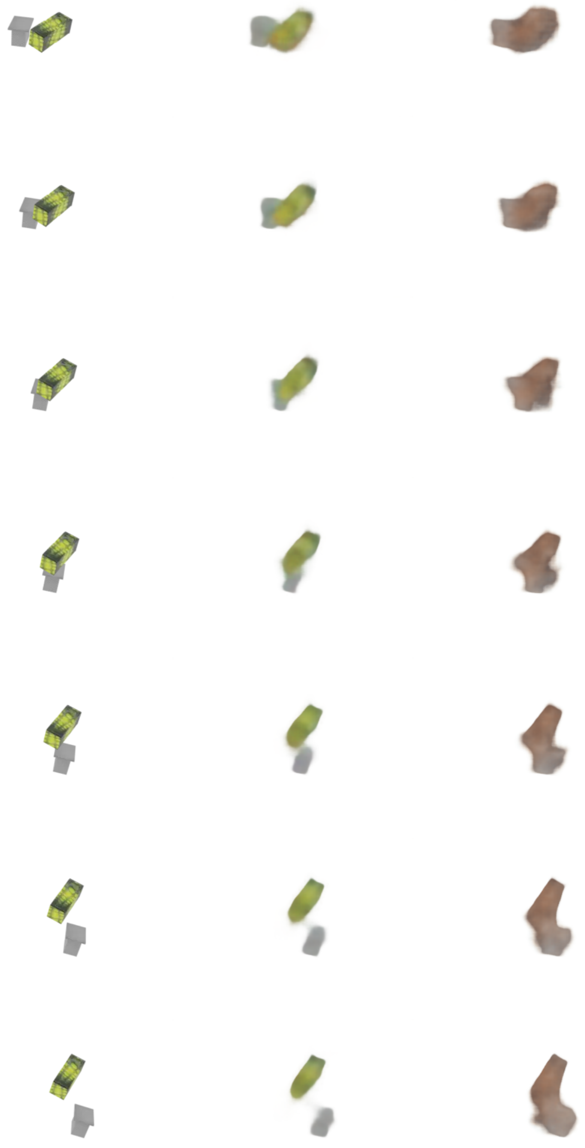

(b) ground-truth; multi-object network; single-object network

Figure 4.10: "Easy" multi-object scene: only two objects, no overlap, little motion.

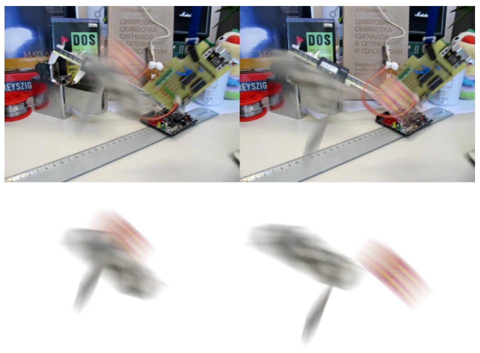


(a) Input to the network (top) and input without background for comparison.



(b) ground-truth; multi-object network; single-object network

Figure 4.11: "Medium" multi-object scene: only two objects, but object trajectories overlap.


(a) Input to the network (top) and input without background for comparison.

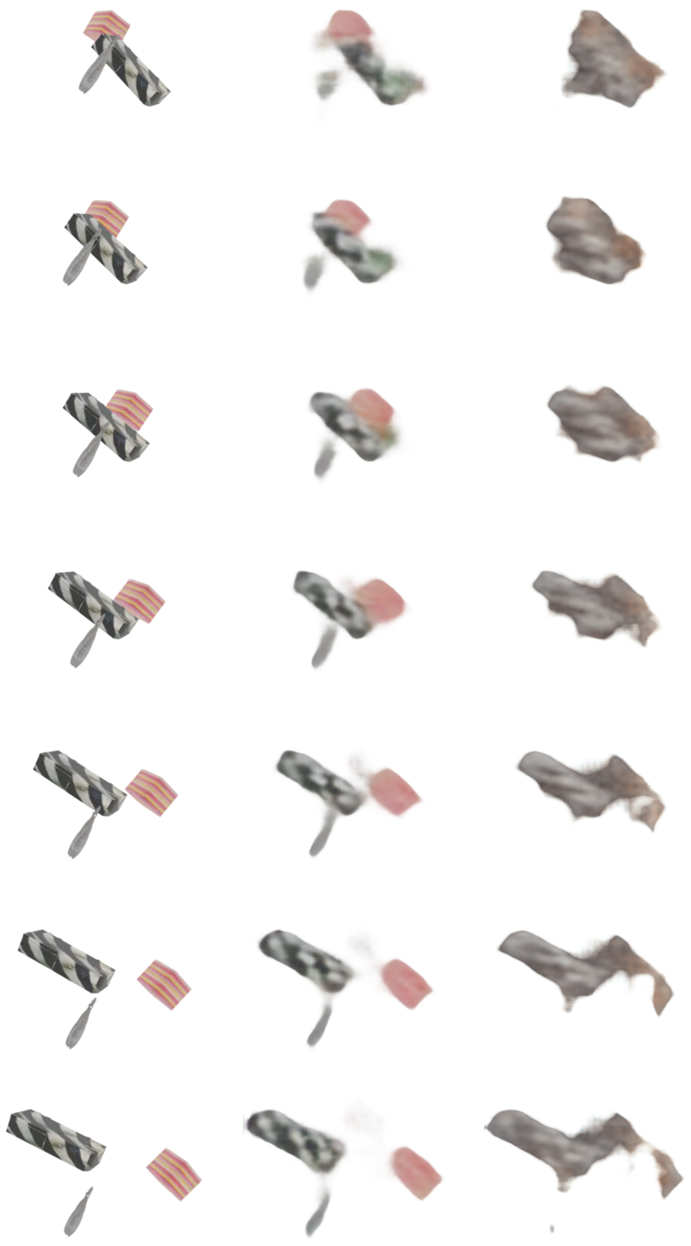
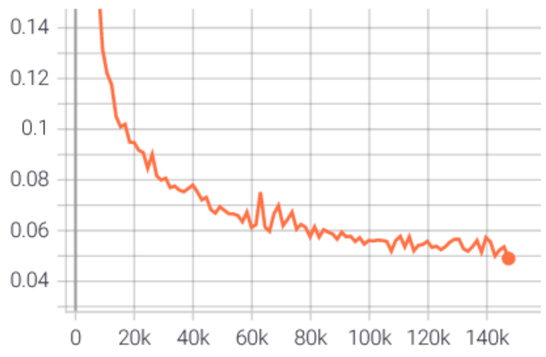
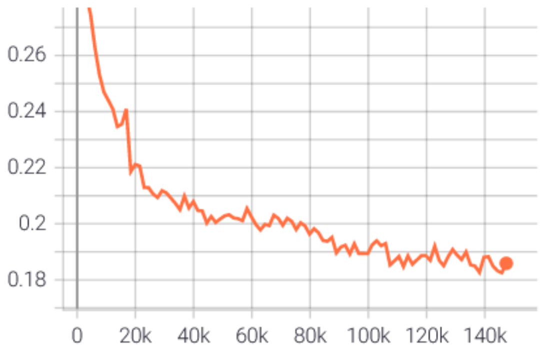

(b) ground-truth; multi-object network; single-object network

Figure 4.12: "Hard" multi-object scene with three objects moving in different directions on overlapping trajectories on a busy background.



(a) Alpha Dice loss



(b) RGB-GT L1 loss

Figure 4.13: Validation loss evolution of the multi-object network used in Section 4.4.

4.5 Self-Supervised Learning

Supervised learning refers to learning with ground-truth data, where the neural network can check and correct its outputs against a known solution that is provided, such as the synthetic dataset used here. Self-supervised learning refers instead to a system where no ground-truth data is provided, but the system autonomously creates a supervised task from the provided input data.

A natural approach for self-supervised learning of the FMO task is input reconstruction. The network has the same inputs and outputs as before, however, the rendered frames are not compared to a ground-truth but are instead averaged together to recreate the FMO blur, which is then overlaid on the background to recreate the input image. This process is essentially identical to the way in which the input data was originally created. The dissimilarity between this recreation and the original input is then used as the actual loss function.

While the reconstruction loss is straightforward to define, guiding training towards the desired result is tricky, since there are many conceivable ways in which the network could recreate the input, besides correctly predicting the high-speed frames.

For example, if the predicted alpha values can take semi-transparent values, the network might return the entire FMO in each frame with a semi-transparent alpha value, thus accurately recreating the input image. This tendency to disregard the timestamp, at least at first, is already exhibited in the supervised approaches above (see Figure 4.2), but is extra difficult to overcome in a fully self-supervised scenario, since there is no ground-truth to correct this behavior.

Technically, the network could even learn an identity transformation, passing the input through unchanged to achieve perfect reconstruction. The task is therefore to define additional loss functions, each encoding the desired approach and constraining the network to reconstruct the image according to the FMO model.

The possibilities here are somewhat limited by the motivation for a fully self-supervised approach: the FMO task should be learnable from any FMO dataset, without requiring a ground-truth or generation of a synthetic dataset. So for example a contrastive loss, which compares the outputs for two inputs consisting of the same FMO on different backgrounds, would be very useful in guiding the network towards focusing on the FMO blur and not the background, but it is hardly possible to generate such inputs without a synthetic dataset.

The main issue faced in most of the following experiments is that producing transparent renders and thereby reconstructing only the background without an FMO provides a fairly decent result in terms of the reconstruction score, especially since the area covered by the FMO will usually be fairly small. Approaching the correct result through imperfect renders will usually lead to worse reconstruction scores.

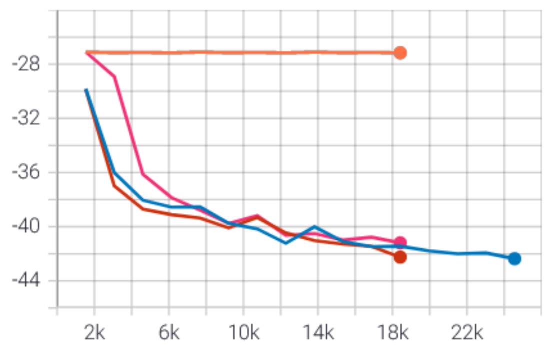


Figure 4.14: Reconstruction quality of validation images, measured by PSNR loss between input and reconstruction, for different reconstruction loss functions: L1 (orange), L2 (blue), SSIM (pink), MS-SSIM (red)

Experiments

The first experiment involves using only a reconstruction loss, without any other loss functions to further specify the desired output. Four loss functions were tested: L1, L2, Structural Similarity Index Measure (SSIM) [21] and its extension, the Multi-Scale SSIM (MS-SSIM) [20]. While L1 and L2 are per-pixel loss functions, SSIM and MS-SSIM are perceptual loss functions intended to evaluate image similarity based on human perception rather than numerical proximity.

Figure 4.14 shows that L1 failed to escape the all-transparent local minimum, whereas the other reconstruction losses produced some result. This result can be seen in Figure 4.15: all three loss functions succeeded in disentangling the FMO from the background, but because it is simply a pure reconstruction loss with no other guidance, the results are just the input blurs at half opacity. Since the input is a double blur, the renders are separately reconstructed into the two input blurs, resulting in the "jump" visible halfway in Figure 4.15.

One idea could be to encourage a smooth trajectory, which could be approached using cross-correlations. Performing a cross-correlation between two signals, such as two images, gives a measure of their similarity. If one image is approximately a translation of the other, then the cross-correlation can be repeated with the input images shifted to different possible offsets relative to each other to reveal the direction and distance of the translation. If this is done for each pair of consecutive frames produced by the network, a loss can be placed on the resulting offsets, encouraging them to be equal, which would translate into a smooth trajectory.

Adding this loss function creates a slightly smoother animation, mainly noticeable as the "jump" from one blur-half to the next being less sudden, as seen in Figure 4.16, but the effect is not extremely significant.

A more promising idea is to somehow force the alpha layer to be binary, which would force the network to create smaller objects with movement to create an accurate reconstruction, since only movement would allow for semi-transparency in the reconstructed image. However, all attempted ways of adding a loss function to discourage intermediate values and encourage zero and one values in the alpha layer failed, since this gives the network more incentive to default to the all-zero fully transparent renders, as discussed previously.

Adding another loss function to discourage an all-zero alpha layer based on the mean alpha value prevented this issue, but didn't lead to a useful result, mainly causing all pixels to be slightly opaque.

Another approach is to simply round the alpha layer directly, such that all values below 0.5 become 0, and all above 0.5 become 1. Since the derivative of the rounding function is always 0, implementing this requires a workaround by rounding during the forward pass of the network, but using an identity function on the backward pass, as though no rounding had taken place. This

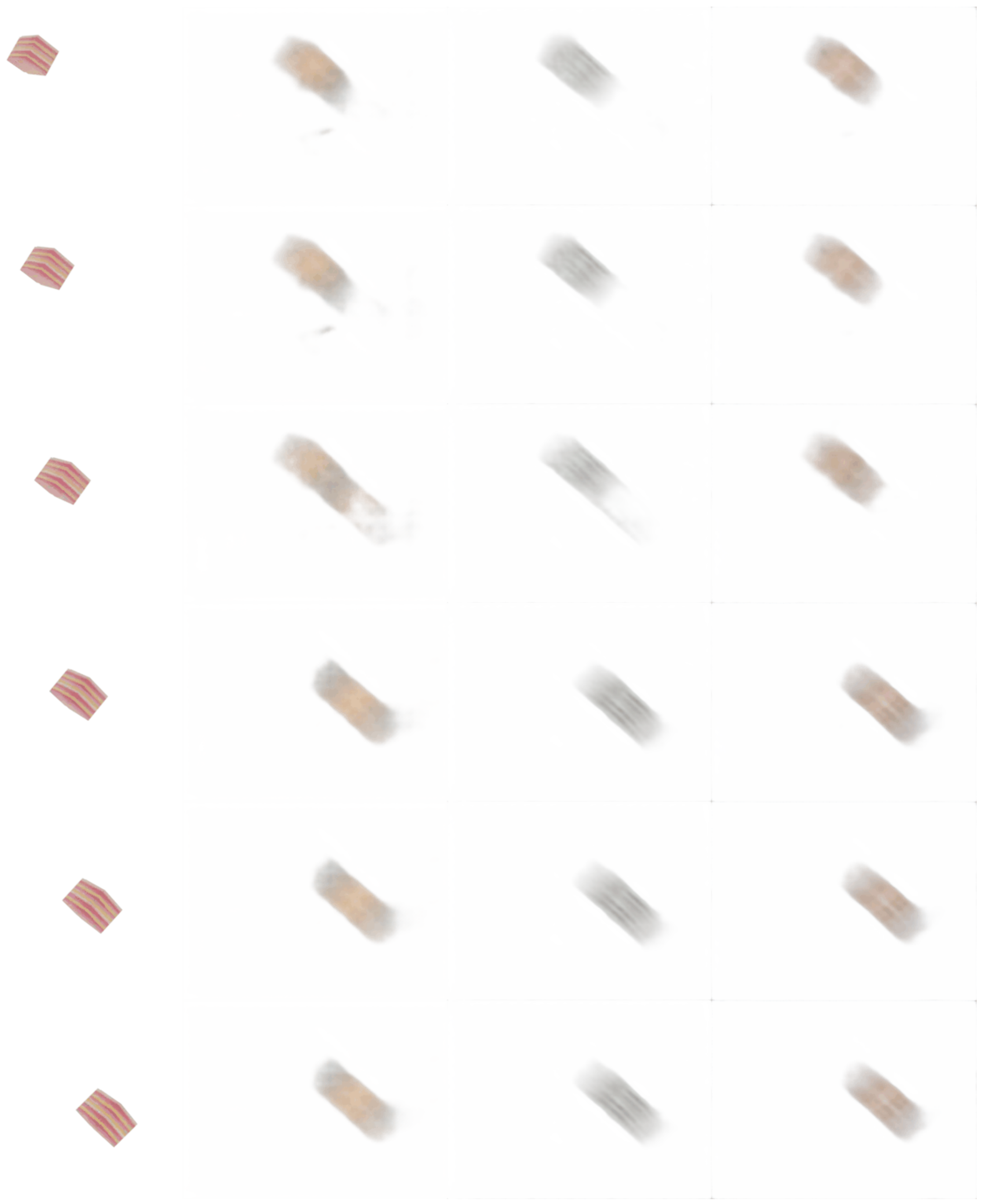
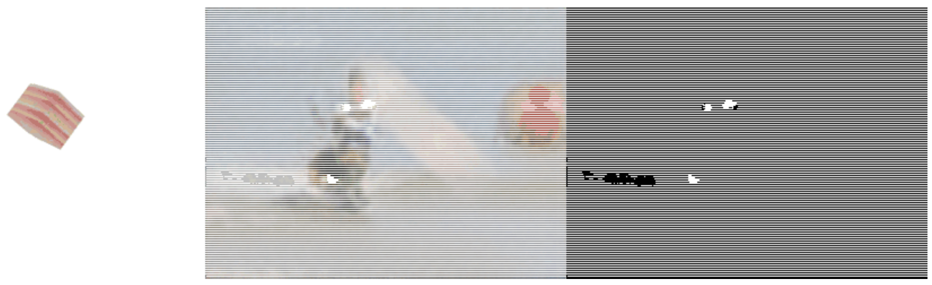
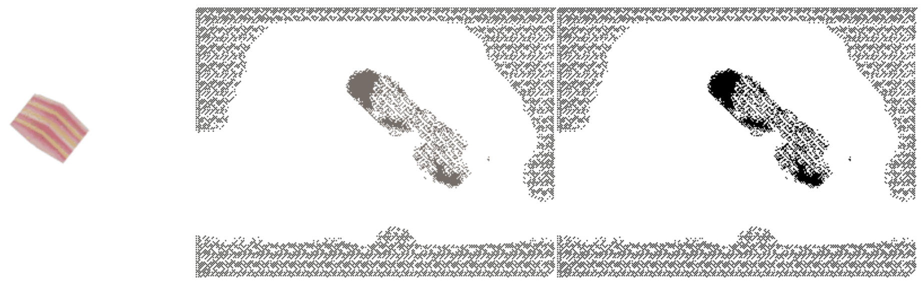


Figure 4.15: Renders of pure reconstruction network, using L2, SSIM, and MS-SSIM respectively.

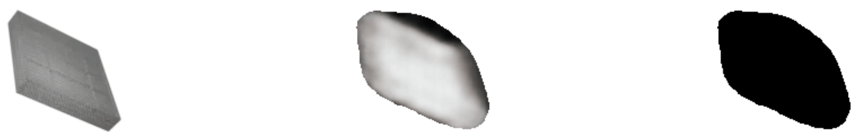
Figure 4.16: Reconstruction loss with cross-correlation trajectory-smoothing loss showing a slightly smoother transition between the first and second blur halves. The rightmost column is the alpha layer.



(a) reconstruction loss only



(b) reconstruction with background rejection loss



(c) reconstruction with contiguity loss

Figure 4.17: Input reconstruction-based experiments using a forced-binary mask. Columns are ground truth, predicted frame, and alpha mask of predicted frame.

approach may be dubious and more correct methods are presumably available to achieve a similar result, but the results are interesting.

Figure 4.17 shows various experiments run with this forced-binary version of the network output:

- A pure reconstruction loss resulted in a mask that was still semi-transparent since the alpha layer was striped with alternating rows of ones and zeros, while the image returned was mainly just the unchanged input. The output was also rather unstable and could be very different from one epoch to the next.
- Adding a "background rejection" loss, which penalizes similarity between the predicted frame and the background, shifted focus back to the FMO, but the artificial semi-transparency was still present as a sort of dither.
- Adding a "contiguity" loss, which encourages the average value of neighborhoods of the alpha layer to be either close to zero or close to one removes the dither or striping to create solid regions. The result is still somewhat unstable though.

4.6 Other Experiments

Many experiments were run in the course of this work, of which many didn't produce useful results. Some of these experiments are listed here.

- Perceptual losses.

  The paper *Loss Functions for Neural Networks for Image Processing* [24] compares the performance of loss functions for image-based tasks such as de-noising and de-mosaicing. The paper finds that using standard loss functions such as L1 and L2 produces visually unappealing results and instead promotes perceptual losses such as SSIM [21] and MS-SSIM [20], possibly in combination with an L1 loss for better color accuracy. Using these loss functions was shown to lead to better final results than L1 or L2, even as measured by L1 or L2 metrics.

  Many variations of such loss functions were tried in the course of this work, for example as an RGB loss for the basic supervised approach, replacing the L1 or L2 functions previously used there. However, these experiments showed worse results and slower convergence behaviors than using the L1 and L2 losses and were therefore abandoned.

- Weighted initialization.

  As has been mentioned multiple times throughout this work, fast convergence is desirable for efficient experimentation. The "animation point" has been discussed as the point during training where the network learns the importance of the timestamp added to the latent representation and begins producing animated outputs. Analyzing the parameters of a fully trained network does indeed show that the weights corresponding to these timestamps end up having a mean and variance that is many standard deviations greater than that of other channels.

  Knowing this, it would therefore be desirable to specify to the network that special attention should be paid to this timestamp channel from the start. However, perhaps not unexpectedly, simply increasing the corresponding weights during the initialization of the untrained network did not result in any significant changes to the convergence behavior.

- Staggered loss functions.

  When using multiple loss functions for the FMO task, such as an alpha loss, an RGB loss, and perhaps an additional perceptual loss, there is a hierarchy of importance - for example, the RGB loss requires a reasonable alpha mask to be effectively trained. Therefore, staggering these loss functions in the sense that the network first only trains with the alpha loss, allowing it to produce a reasonable mask before the RGB loss is introduced, could accelerate training.

  This did turn out to be effective, but the strategies for instead separating the concerns of the alpha and RGB layers as discussed in Section 4.3 lead to the same result and require less engineering overhead.

Chapter 5

Conclusion

The goals of this work were to explore extensions to the DeFMO method and to create a user-friendly system that enables efficient exploration and experimentation.

The created system is fast, modular, and extensible. Running and evaluating experiments is possible with minimal configuration and minimal manual effort. The required synthetic datasets can be generated in parallel, with the generation of a full dataset being achievable overnight on one GPU and requiring only a few Gigabytes of storage space.

Methods to optimize training for complex image tasks were studied, managing the issues caused by small batch sizes through the use of Group Norm and accelerating the convergence of dependent learning objectives through decoupling of their respective loss functions. Decoupling alpha and RGB losses lead to convergence after only 35%-60% of the number of training steps, depending on the loss functions used.

DeFMO was modified to use a new double-blur input approach, which reduces the number of required frames and disambiguates the direction of motion. While the double blur approach was shown to be a more difficult learning task, requiring more training to reach the same level of deblurring quality, the advantages it provides are significant, for example allowing a network to be successfully trained to deblur multiple FMOs in a single image with a high degree of visual quality on both synthetic and real-world data.

Promising first steps were taken in direction of a fully self-supervised approach to the FMO task, which could eliminate the need for a synthetic dataset altogether and make training possible on any dataset of fast-moving objects, such as videos of sporting events, although further research is needed to more explicitly define the learning task, ideally through explicit modeling of the object and its trajectory.

Bibliography

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[2] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. ShapeNet: An Information-Rich 3D Model Repository. Technical Report arXiv:1512.03012 [cs.GR], Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015.

[3] M. Cimpoi, S. Maji, I. Kokkinos, S. Mohamed, , and A. Vedaldi. Describing textures in the wild. In *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[4] Alex Clark. Pillow (pil fork) documentation, 2015.

[5] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018.

[6] et al. Falcon, WA. Pytorch lightning, the ultimate pytorch research framework. *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning*, 3, 2019.

[7] Google Developers developers.google.com. A new image format for the Web — WebP. `https://developers.google.com/speed/webp/`. [Accessed 02-Aug-2021].

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[9] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[10] Meiguang Jin, Givi Meishvili, and Paolo Favaro. Learning to extract a video sequence from a single motion-blurred image. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6334–6342, 2018.

[11] Jaihyun Koh, Jangho Lee, and Sungroh Yoon. Single-image deblurring with neural networks: A comparative survey. *Computer Vision and Image Understanding*, 203:103134, 2021.

[12] Jan Kotera, Denys Rozumnyi, Filip Sroubek, and Jiri Matas. Intra-frame object tracking by deblatting. *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, Oct 2019.

[13] Orest Kupyn, Tetiana Martyniuk, Junru Wu, and Zhangyang Wang. Deblurgan-v2: Deblurring (orders-of-magnitude) faster and better, 2019.

[14] Fausto Milletari, Nassir Navab, and Seyed-Ahmad Ahmadi. V-net: Fully convolutional neural networks for volumetric medical image segmentation, 2016.

[15] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[16] Denys Rozumnyi, Martin R. Oswald, Vittorio Ferrari, Jiri Matas, and Marc Pollefeys. Defmo: Deblurring and shape recovery of fast moving objects. In *CVPR*, Nashville, Tennessee, USA, June 2021.

[17] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization?, 2019.

[18] Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P. Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network, 2016.

[19] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization, 2017.

[20] Z. Wang, E.P. Simoncelli, and A.C. Bovik. Multiscale structural similarity for image quality assessment. In *The Thrity-Seventh Asilomar Conference on Signals, Systems Computers, 2003*, volume 2, pages 1398–1402 Vol.2, 2003.

[21] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[22] Yi Wu, Jongwoo Lim, and Ming-Hsuan Yang. Online object tracking: A benchmark. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2013.

[23] Yuxin Wu and Kaiming He. Group normalization, 2018.

[24] Hang Zhao, Orazio Gallo, Iuri Frosio, and Jan Kautz. Loss functions for neural networks for image processing, 2018.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Temporal Super-Resolution of Multiple Fast-Moving Objects

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Klaeger	Adrian

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 20.08.2021	Adrian Klaeger

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.