Master Thesis
Computer Science
Artificial Intelligence

Optical Flow Upsamplers Ignore Details: Neighborhood Attention Transformers for Convex Upsampling

Alexander Sebastiaan Gielisse

March 2023

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Alexander Sebastiaan Gielisse: *Optical Flow Upsamplers Ignore Details: Neighborhood Attention Transformers for Convex Upsampling* (2023)

Supervisors:	Dr. J.C. van Gemert Dr. N. Tömen
Graduation Committee:	Dr. J.C. van Gemert (Associate Professor) Dr S.E. Verwer (Associate Professor)

Acknowledgements

I would like to express my gratitude to my supervisors Dr. van Gemert and Dr. Tömen, for their time, useful insights and wonderful guidance throughout the project. Next to that, I would like to thank Dr. Verwer for agreeing to be part of my graduation committee.

I would also like to thank my parents for supporting me. In special, I would like to thank my father for his willingness to learn new concepts and theories in order to help me better understand them. And my mother, for her insightful advises and support. I would also like to thank my girlfriend, Simone, whose willingness to listen to my technical problems and support me throughout has helped immensely.

Sander Gielisse,
March 2023, Delft.

Contents

1 Introduction

The problem of optical flow has long been seen as a fundamental and difficult computer vision problem. The optical flow problem is defined as finding the displacement vector for every pixel in $image_1$, with respect to a subsequent $image_2$. Traditional methods for solving optical flow were generally based on predefined mathematical constraints, but this has numerous issues. More recently, a strong increase in performance was found by formulating optical flow as a learning-based approach. In learning-based approaches, a learnable model is defined with optical flow as the learning objective. There no longer exists the need for predefined constraints. While this sets a strong new baseline, there exist some unsolved challenges. One of these challenges is the computational complexity of current-day learning-based optical flow models. Processing high-resolution images quickly becomes costly, which makes it infeasible to use them for training or use them in practise. As a solution, almost all current-day optical flow models predict flow at a lower resolution that is factor 8 smaller than the input. The final high-resolution flow is then obtained via upsampling. Unfortunately, traditional upsampling approaches such as bilinear or nearest neighbor interpolation have properties that makes them unsuitable for upsampling optical flow. To resolve these issues, convex upsampling was proposed. While convex upsampling improves performance compared to traditional upsamplers, we believe there to remain some shortcomings. Mostly, these shortcomings apply to areas with high amounts of detail, as we will show in our scientific article.

The first section of this thesis consists of a scientific article in ICCV format, which we will try and submit to ICCV 2023. This article is written with the target audience to be an expert in computer vision related theories and practises. In order for a more general reader to be able to understand it, supplementary information might be needed. We provide this supplementary information in Section 3.

2 Scientific Article

The scientific article starts on the next page due to its fixed full-page ICCV format.

Optical Flow Upsamplers Ignore Details: Neighborhood Attention Transformers for Convex Upsampling

A.S. Gielisse
Computer Vision Lab, Delft University of Technology

Abstract

Most recent works on optical flow use convex upsampling as the last step to obtain high-resolution flow. In this work, we show and discuss several issues and limitations of this currently widely adopted convex upsampling approach. We propose a series of changes, inspired by the observation that convex upsampling as currently implemented performs badly in high-detail areas. We identify three possible causes; wrong training data, the non-existence of a convex combination, and the inability of the convex upsampler to find the correct convex combination.

We propose several ideas in an attempt to resolve current issues. First, we propose to decouple the weights for the final convex upsampler, making it easier to find the correct convex combination. For the same reason, we also provide extra contextual features to the convex upsampler. Then, we increase the convex mask size by using an attention-based alternative convex upsampler; Transformers for Convex Upsampling. This upsampler is based on the observation that convex upsampling can be reformulated as attention, and we propose to use local attention masks as a drop-in replacement for convex masks in order to increase the mask size. We provide empirical evidence that a larger mask size increases the likelihood of the existence of the convex combination. Lastly, we propose an alternative training scheme to remove bilinear interpolation artifacts from the model output.

We investigate whether an increase in accuracy can be achieved while leaving the low-resolution flow prediction architecture unchanged. Due to that, our proposed ideas could theoretically be applied to almost every current state-of-the-art optical flow architecture. On the FlyingChairs + FlyingThings3D training setting we reduce the Sintel Clean training end-point-error of GMA from 1.31 *to* 1.18*, which is a* 10% *decrease caused solely by changes regarding the convex upsampler.*

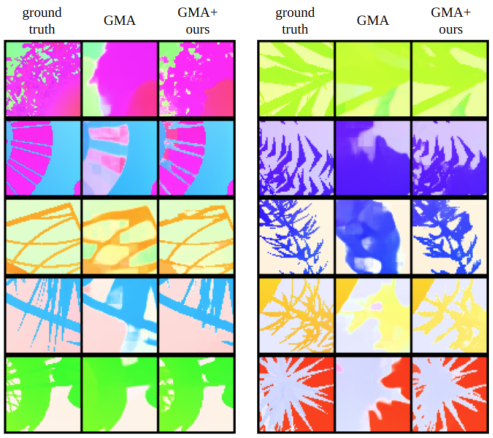

Figure 1. High-detail 64×64 patches from the FlyingThings3D [19] dataset where performance difference is clearly visible. We find 128 high-detail patches using our edge detection approach as described in Section 4.1, after which we manually choose 10 for preview.

1. Introduction

In recent years, optical flow prediction using deep learning has seen a strong increase in accuracy. Unfortunately, one of the problems of optical flow prediction is the computational complexity. Due to this complexity, the current state-of-the-art (SOTA) models [10, 15, 25, 28, 31, 34, 36] predict flow at $1/8$th of the full resolution. Obtaining the high-resolution flow map is then achieved by upsampling the low-resolution flow map. Upsampling methods for upsampling optical flow maps ideally have different properties than is provided by traditional upsampling methods like bilinear interpolation or nearest neighbor upsampling. If two objects at an edge move in different directions, interpolating them in the middle will give values around the mean of the two directions. This is never correct in terms of optical flow; a pixel either follows one object, or the other. Alternatively, interpolation methods that do not use interpolation like nearest neighbor upsampling could be used. Nearest neighbor interpolation when upsampling by factors of 2

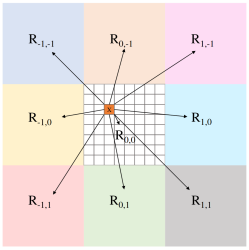

Figure 2. Original convex upsampling as proposed by RAFT [28]. A high-resolution sub-pixel value is written as a convex combination of the low-resolution neighbors, including itself.

simply sets sub-pixel values to be the same as the 'parent' low-resolution pixel. Unfortunately, this has the problem of the flow edges becoming jagged and not aligned with object edges.

To solve these issues, RAFT [28] proposes convex upsampling. There, the main idea is to take a convex combination of the direct neighbors of a pixel. A visual example hereof is provided in Figure 2. That is, for every pixel P, find sub-pixel values p_i by first predicting a 3×3 scalar mask_{p_i} for every sub-pixel p_i. This mask_{p_i} has 9 scalars values; $\text{mask}_{p_i} = (w_0, w_1, ..., w_{(m^2-1)})$ for mask size m. R_{x_r,y_r} denotes the value of a relative low-resolution neighbor. All these relative direct neighbors together, including the pixel itself, form the neighbors of P, called N_P. Then, the value of the sub-pixel p_i is calculated using the dot-product as following.

$$p_i = \langle \text{mask}_{p_i}, N_P \rangle \tag{1}$$

While so far this approach can already be used for upsampling, it uses a linear combination, and not a convex combination. For convex upsampling, a convex combination must be used, which means that the restriction is added that all weights must be positive and sum to 1. Fortunately, this is easily obtained by applying the softmax [2] function on each mask_{p_i}. On a high level, this allows the network to 'choose' which low-resolution pixels to follow, by predicting scalar mask values mask_{p_i} as part of the model output.

By predicting the masks as part of the model output, the network can learn to avoid including multiple neighbor pixels that have different directions, and thereby avoid strong interpolation. Due to the accuracy of convex upsampling compared to the traditional approaches, convex upsampling is currently adopted in all current SOTA models [10, 15, 25, 28, 31, 34, 36] for optical flow.

In this work, we investigate three main possible causes for decreased performance of the convex upsampling method as currently implemented. We first provide our reasoning for each of these three causes based on Equation 1. That is; either p_i is trained against the wrong objective, mask_{p_i} contains wrong values, or N_P is incorrect or insufficiently representative.

1.1. Wrong Training Data

Noticeably, almost all recent works on optical flow are based on the same original PyTorch implementation of RAFT [28], and all use bilinear sampling for augmentation of their training data. While this on itself is not a problem for image_1 and image_2, it does create difficulties for the ground truth flow. The main reason for the proposal of convex upsampling was mostly so that no bilinear upsampling on flow maps had to be done. However, in the current setting convex upsampling is used, but with the learning objective to predict bilinearly interpolated flow. Interestingly, to the best of our knowledge, all public submissions to the Sintel [3] leaderboard show bilinear interpolation artifacts in the form of white and non-crisp edges, even though these artifacts are not in the non-augmented training data. We wonder whether this bilinear interpolation artifact in the submissions is caused bilinear interpolation in the augmentation.

1.2. Difficulty Predicting the Convex Combination

Predicting a convex mask_{p_i} for a sub-pixel p_i of low-resolution pixel P is the main learning objective of the convex upsampler. When upsampling by factor f, a low-resolution pixel gets f^2 sub-pixels p_i. Predicting f^2 masks of size 3×3 has its challenges, especially in areas with many small details. If the upsampler is unable to predict the correct mask, the output will be incorrect, even though a correct convex combination might actually exist. Next, we consider why a trained model would be unable to find a correct convex combination if it exists.

1.2.1 Incomplete Input Features

A possible reason for the convex upsampler to predict the wrong convex mask while a correct convex mask exists, would be if the correct mask can simply not be predicted from the input data of the convex upsampler. In such case, the convex upsampler has an impossible objective; it cannot know the correct answer based on the input.

1.2.2 Lack of Inductive Spatial Bias

In convex upsampling as currently implemented, masks are predicted as a single vector of size $m^2 f^2$, for mask size $m = 3$ and upsampling factor $f = 8$. This fully-connected approach does not allow for any spatial inductive bias to be hard-coded into the model. This makes it more difficult to learn the mask prediction problem given the limited complexity of the convex upsampler.

1.2.3 First Steps are Noisy Variants

Another possible problem is that many flow prediction architectures take a multi-step approach. There, each step can be seen as a refinement step of the low-resolution flow. After several refinements, the final flow is obtained. To train the network, each step is supervised. That is, a loss is calculated for each intermediate flow map. To compare the low-resolution output of each step with the ground truth flow, the low-resolution flow maps are upsampled to high resolution using the convex upsampler. Many SOTA works [10, 15, 25, 28, 34, 36] choose to share the same convex upsampler and its weights over all steps. However, we consider the first flow predictions to be extremely noisy variants of the final flow. When the convex upsampler and its weights are shared over all steps, this is effectively equivalent to just adding strong noise to a part of the input data. In general, the loss is down-weighed for the first steps, but this could be insufficient.

1.3. Existence of a Convex Combination

For our last possible cause for decreased performance, we consider that a strong limitation of convex upsampling is that a high-resolution pixel can only be predicted correctly if there exists a convex combination of the low-resolution neighbor pixels N_P such that the dot-product $\langle \text{mask}_{p_i}, N_P \rangle$ forms the desired output value. Therefore, if the low-resolution flow map N_P is not correct or not locally informative enough and no such convex combination exist, the desired output value can never be obtained. Note that this is not due to mask_{p_i} being incorrect. Namely, from the perspective of the convex upsampler, this is again an impossible objective as there exist no solution for the given N_P. One possible way to overcome this is to improve the underlying low-resolution flow predictor such that the values of N_P ensure that the convex combination exists, but this is difficult and the problem cannot be easily defined.

1.4. Main Contributions

In this work, we provide several main contributions, which are listed below.

- An alternative training scheme to remove bilinear interpolation artifacts from the model predictions.
- Decoupling the convex upsampler and its weights for the last refinement step, to avoid noisy early steps from influencing the upsampler of the last refinement step.
- Adding more input features to the convex upsampler at multiple scales, to better align flow with object edges.
- A 3-step Transformer-based hierarchical convex upsampling method, that uses local attention maps as a drop-in replacement for convex upsampling masks.
- Analysis on the performance of our proposed ideas, based on the level of detail in a patch.

2. Related Work

2.1. Optical Flow

In many recent SOTA models [10, 15, 25, 28, 34, 36] optical flow is formulated as an optimization problem. Here, a single resolution flow is kept throughout, which is step-wise optimized. RAFT [28] was the first optical flow model to use a gated recurrent unit (GRU) [4] and use it to mimic a step-wise optimization process. Given an initial displacement of $(0, 0)$ for every pixel, the GRU is given the objective to provide a Δ_{flow} repeatedly, based on the information with respect to the current location. This drastically reduces the search space compared to earlier works, and can provide highly accurate flow. Due to running on $1/8th$ of the resolution, upsampling is needed as the last step. While their initial work used bilinear upsampling, it was noted to be sub-optimal, and hence their proposal for convex upsampling followed.

Following RAFT, GMA [15] proposes the use of global attention on the features of $image_1$ to better estimate hidden motions. If objects from $image_1$ have no matches in $image_2$, the ability to correctly predict the flow will depend strongly on the global contextual understanding. This model is used as a general baseline among most of our experiments.

FlowFormer [10] and GMFlowNet [36] both take attention one step further, and incorporate Transformer blocks [29, 17] in their network design. By stacking several Transformer blocks, they enhance their features and make them more contextual, which for their objective shows to be beneficial as they show to strongly outperform the earlier set benchmarks of RAFT [28] and GMA [15]. GMFlow [31] takes the idea of using Transformers a step further and does not use any step-wise optimization approach, but solely uses contextual image features to construct the correlation volume and directly extract optical flow.

MS-RAFT(+) [13, 14] tries to increase performance by taking another approach. Their work is based on RAFT [28], but as a multi-scale approach with different resolutions, or scales. Important for us is that the convex upsampling is different, as they upsample 3 times by a factor of 2, rather than once by a factor of 8.

2.2. Upsampling

In general, upsampling optical flow using traditional upsampling by a low factor like 2 is not necessarily a big problem with respect to the error, as the resulting artifacts are small. Therefore, methods like FlowNet [6], FlowNet2 [12], and SpyNet [21] that already output flow at a relatively high resolution, simply adopt traditional upsampling like bilinear- or nearest neighbor interpolation. All the current SOTA models [10, 15, 25, 28, 31, 34, 36] output significantly lower-resolution flow. The upsampling that follows

is often by factor 8, which makes traditional upsampling unsuitable, and hence convex upsampling is widely adopted.

2.2.1 Super Resolution

Repeatedly upsampling feature maps to high resolution can become costly in terms of memory. In super resolution, a similar problem exists. ESPCN [24] proposes to directly interpret different channels in a low-resolution feature map as sub-pixel values. By doing so, an r^2-dimensional feature map can be reshaped to a 1-dimensional feature map that is factor r larger in both the height and width dimension. Effectively convex upsampling is similar, as it simply predicts f^2 low-resolution flow maps that are merged together to obtain high-resolution flow that is factor f larger.

2.3. Attention and Local Attention

Attention was proposed as part of the Transformer architecture [29]. A great advantage of attention is that it is global, rather than local as is the case with convolutions. In an attempt to bring Transformers to images, Vision Transformer (ViT) [5] were proposed. By directly inputting flattened non-overlapping image patches, the ViT is able to model the entire global context. Unfortunately, ViT lack a strong inductive bias, and high performance is often only achieved for enormous datasets and incredibly long training. To resolve this, research has aimed at bringing back the hierarchical structure of convolutional neural networks. As such, SwinFormer [17] was proposed. There, the authors gradually reduce the image resolution while increasing the dimensionality, similar to how this is often done in convolutional neural networks. To make this approach computationally feasible and to focus on local features first, global attention is reduced to local attention; attention is done inside of a sliding window. In a similar way, Neighborhood Attention (NA) [8] was proposed.

3. Method

3.1. Increasing Mask Size

Improving the low-resolution flow map and thus the direct neighbors of a pixel N_P is difficult, and has already received great attention in recent works. Instead, we investigate a different approach. We propose to simply increase the size of N_P and include more low-resolution pixels, rather than just look at the direct neighbors using a 3×3 mask. By allowing more values to be included in the convex combination, the problem that the convex combination does not exist becomes less likely. However, increasing the mask size is not straightforward as the current convex upsampler predicts all masks in a single vector which has size $m^2 f^2$ for mask size m and upsampling factor f. Due to this vector being predicted in a fully-connected manner, directly increasing the mask size m comes with a too strong increase in the number of parameters.

3.1.1 Convex Combination using Attention

As a way of increasing the mask size without increasing the number of parameters, we consider attention maps. The masks as generated by the convex upsampler as proposed by RAFT [28] are normalized using softmax [2] to ensure they are positive and sum to 1. This same logic is used for attention maps [29]. Something that is considered useful here is that for attention the number of parameters is decoupled from the size of the attention map. This is because attention only uses a pixel-wise key, query and value function which are shared by all pixels in the input. We propose to simply use attention maps as a drop-in replacement for the convex masks, as they have the same mathematical properties. By doing so, an alternative formulation of convex upsampling is obtained where the number of parameters is decoupled from the mask size, and so the mask size can be increased without any increase in the number of parameters. Note that attention maps are generally as large as the whole input image. To reduce the size of the attention maps, we adopt local attention instead, where we set the local attention window size to the desired mask size m.

3.2. Architecture Components

To explain our overall method, we first make it explicit by providing a visual overview in Figure 3. We refer to this approach as Transformers for Convex Upsampling (TCU). Next, the individual components are described and explained.

3.2.1 Context Encoder

Possibly, the correct mask can simply not be predicted from the input data of the convex upsampler. Therefore, we investigate whether adding more features helps. The input image_1 cannot be directly concatenated to the input of the convex upsampler, due to the image_1 of size $(h \times w)$ being factor f larger than the input of the convex upsampler of size $(h/f \times w/f)$. However, most SOTA [10, 15, 25, 28, 31, 34, 36] models have some form of a context encoder, which is generally a shallow ResNet [9] architecture. As the output, the context encoder provides an encoded format of image_1 with matching resolution $1/8$. We propose to concatenate these features to the input of the convex upsampler.

3.2.2 Transformers for Convex Upsampling

Now we consider our attention-based convex upsampler. First, $\text{internal}_{1/8}$ is defined as the output state of the low-resolution flow predictor, which could be any recent op-

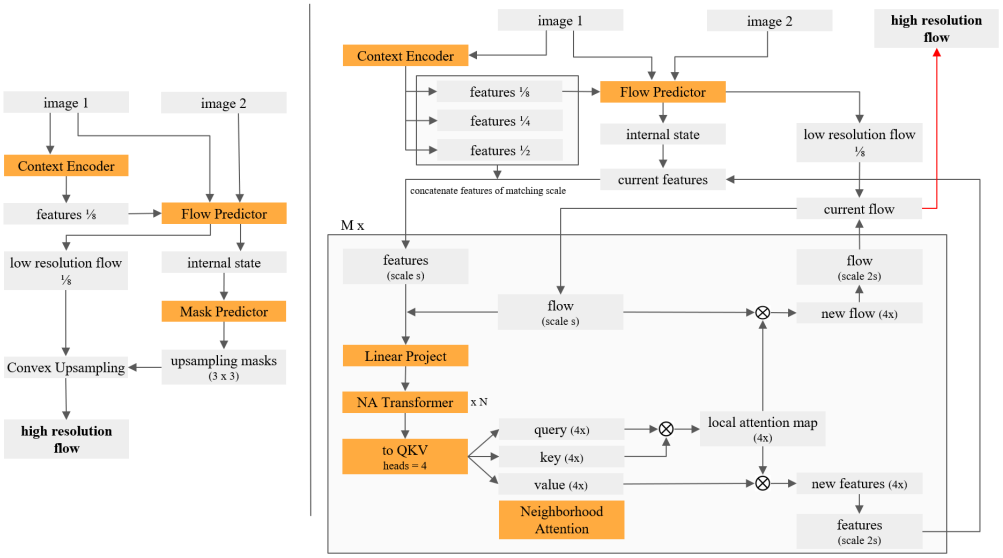

Figure 3. **Left:** the original convex upscaling method as proposed by RAFT [28]. **Right:** our proposed multi-step Transformer convex upsampling network. The feature extractor is adopted from RAFT [28] but we extract the features at 3 all intermediate scales. The Neighborhood Attention Transformer blocks [8] perform local neighborhood attention to enhance features. Attention is used to both upsample the low-resolution flow as well as the feature maps. The block at the bottom is executed M times, and hence for $M = 3$ the low-resolution flow is upsampled by factor 8.

tical flow model. Then, $\text{internal}_{1/8}$ is concatenated with $\text{image}_{1/8}$ and $\text{flow}_{1/8}$. This concatenated feature map is fed into $N_i = 2$ subsequent Transformer blocks for feature enhancement. From the thereby obtained features, a linear projection is used to obtain the key, query, and value, as needed for attention. Here, 4 attention heads are used as this results in 4 sub-pixels and hence upsamples by factor 2. Then, 4 local attention maps are constructed using the key and query vectors. These are aggregated with both the local values from the value function, as well as the local current flow. The same attention maps are used for both aggregations. The reason for this is that the flow vector is calculated via a convex combination of other flow vectors, which makes it only natural to construct the corresponding feature vector in the same manner. The double aggregation provides two outputs; 4 feature maps, and 4 flow maps. By interpreting each of these 4 maps as sub-pixels, an upsampled feature map as well as an upsampled flow map is obtained, both upsampled by factor $f = 2$. Then, the features for the new scale are concatenated, and the steps are repeated. For $f = 8$, this can be obtained using $M = 3$.

Note that this approach also allows for adding additional multi-scale input features. While the context encoder normally only outputs scale $1/8$, we also extract features from scales $1/4$ and $1/2$. We concatenate these additional features to the input of the convex upsampler between subsequent convex upsampling operations. Next to that, this multi-step approach also provides a stronger spatial inductive bias, as the network only has to learn the relative position of 4 sub-pixels at the time, rather than all 64 sub-pixels at once.

3.2.3 Neighborhood Attention Transformers

For the Transformer blocks Neighborhood Attention Transformer (NAT) [8] are used. These NAT layers use layer normalization [1], kernel size m, multi-head attention with k-heads, and a head dimensionality of $d_{\text{head}} = 32$. The number of heads is set to k based on the dimensionality d of the block as $k = d/d_{head}$. For each block a different dimensionality is used; $(128, 64, 32)$ in that order. Relative position bias [7, 8, 17, 23] is used to encode relative position, as standalone Transformers have no notion of relative or absolute spatiality. Unfortunately, this couples the mask size to the number of parameters again, as the bias matrix has $(2m-1)^2$ parameters for each attention head, where m is the mask size. However, the number of parameters that follow from this is small compared to the overall model, so it is not considered a problem.

3.3. Network and Training Details

3.3.1 Decoupled Upsampler for Last Refinement

By design, the step-wise optimization architectures [10, 15, 25, 28, 34, 36] supervise the output of every refinement

step, effectively running the convex upsampler $n \times I$ times, where n is the batch size, and I is the number of steps, or iterations. However, we take a different approach, as we want the convex upsampler of the final output refinement iteration to fully focus on its own objective, and not have its parameters shared with noisy estimates from earlier refinement iterations. To achieve this, we propose to decouple the convex upsampler of the last refinement iteration and give it its own weights.

This also brings the advantage that a different upsampling method can be used for each refinement iteration. Our convex upsampling approach, TCU, uses more memory than the original convex upsampling approach. So, we exploit the idea of a decoupled upsampler for the last refinement iteration to make our TCU model feasible in practise. Namely, the original shared convex upsampler is used for the first $(I-1)$ refinement iterations, and TCU is only used for the last refinement iteration which provides the final model output. Note that at test-time, only the upsampler of the last refinement iteration is used.

3.3.2 Optimization Settings

The training settings and AdamW [18] optimizer, as well as the L_1 loss function, are adopted from RAFT [28] and GMA [15]. During training 12 refinement iterations are used for the underlying step-wise flow predictor, which at test time is increased to 32 for Sintel [3] and to 24 for all other datasets [28].

3.3.3 Fine-tuning

Our training setting is highly similar to the original training setting. The only difference is the convex upsampler from the last refinement iteration. Training all model components from random initialization in this highly similar setting would be unnecessarily costly. Instead, all the training sessions are started with pre-trained weights for the flow predictor and the convex upsampler of the first $(I-1)$ iterations. Only the weights of the convex upsampler for the last refinement iteration are randomly initialized. For all experiments, we fine-tune for $100K$ iterations with a batch size of 3, on the dataset that the model was last trained on. A learning rate of $1e-4$ is used for the pre-trained weights, and a learning rate of $2e-4$ is used for the untrained upsampler of the last refinement iteration.

3.3.4 No Bilinear Augmentations

An additional training scheme is used to remove bilinear interpolation artifacts that are caused by training on bilinearly interpolated flow. Disabling this interpolation in general is likely not a good idea, as it is used to avoid overfitting. Instead, we propose an additional training scheme; (-AUG).

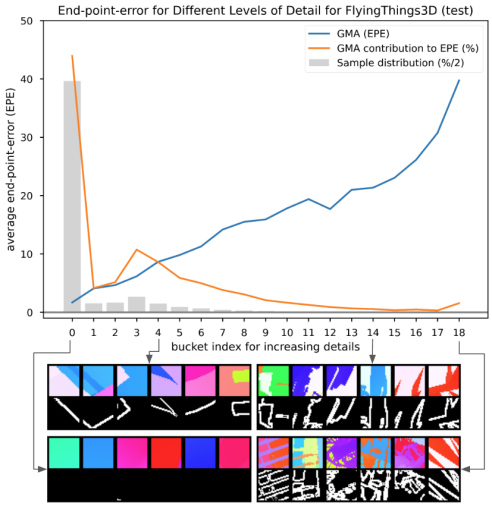

Figure 4. The average end-point-error for increasing amount of detail on FlyingThings3D (test) [19], after being trained on C+T. The performance degrades fast for higher amount of detail. To give an impression of the level of detail per bucket, 6 random patches from each bucket are shown for FlyingThings3D [19]. The contribution of each level of detail to the overall EPE is given as well. Although it is not provided here, the graph for the Sintel [3] dataset looks almost similar.

There, a trained model is trained for an additional 40K iterations with all interpolation-based augmentations disabled, after being trained as described in Section 3.3.3.

3.3.5 Dataset Notations

Among the experiments different datasets are used. C+T refers to training on FlyingChairs [6] and then on FlyingThings3D [19]. C+T+S+K+H refers to training on a mix of data of FlyingChairs [6], FlyingThings [19], Sintel [3], KITTI-15 [20] and HD1K [16].

4. Experiments

4.1. Performance on High-Detail Areas

In areas with no details, the convex upsampler has a remarkably easy task, simply due to the low-resolution flow being smooth, and hence the convex mask weights not being of large importance. The important reason for our proposed alternative convex upsampler is with regards to improved performance on high-detail areas, as we expect the convex upsampler to fail on these areas. To investigate this, we do not build a new dataset for this purpose, but instead look at the performance on non-overlapping 32×32 patches of the test data. For these patches, we look at the ground truth optical flow. We assume the number of edge pixels in the ground truth flow map to strongly correlate with the amount

Statistic	Bucket	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
number of samples		290,817	11,241	12,280	19,340	11,034	6,662	4,896	2,985	2,190	1,441	1,018	721	558	347	281	168	196	111	434
samples (percentage)		79.3%	3.1%	3.4%	5.3%	3.0%	1.8%	1.3%	0.8%	0.6%	0.4%	0.3%	0.2%	0.2%	0.1%	0.08%	0.05%	0.05%	0.03%	0.12%
samples (reverse cumulative percentage)		100%	20.7%	17.6%	14.3%	9.0%	6.0%	4.2%	2.8%	2.0%	1.4%	1.0%	0.8%	0.6%	0.4%	0.3%	0.2%	0.2%	0.15%	0.12%
contribution to error (percentage)		44.0%	4.1%	5.1%	10.7%	8.6%	5.9%	5.0%	3.8%	3.1%	2.1%	1.6%	1.3%	0.9%	0.7%	0.5%	0.3%	0.5%	0.3%	1.6%
contribution to error (reverse cumulative percentage)		100%	56.1%	52%	46.9%	36.2%	27.6%	21.7%	16.7%	**12.9%**	9.8%	7.7%	6.1%	4.8%	3.9%	3.2%	2.7%	2.4%	1.9%	1.6%

Table 1. Distribution of samples over each of the buckets in Figure 4 for FlyingThings3D [19].

Method	FlyingThings3D		Sintel (train)		KITTI-15 (train)		Number of
	Train	Test	Clean	Final	F1-epe	F1-all	parameters
GMA (recomputed)	10.34	3.07	1.31	**2.75**	4.48	16.86	443K
+DC	9.38	2.84	1.23	2.78	4.43	16.89	443K
+DC+FT	9.53	2.86	1.24	2.79	4.55	16.92	743K
+DC+TCU(3/3/3)+FT	9.51	2.73	1.22	2.83	4.52	16.80	695K
+DC+TCU(9/7/5)+FT	9.24	2.75	1.21	2.80	**4.36**	**16.26**	700K
+DC+TCU(9/7/5)+FT-aug	**8.97**	**2.61**	**1.18**	3.01	4.50	16.64	700K

Table 2. Training and generalization performance for our different convex upsampling approaches. Model names are explained in Section 4.2.

of details in the flow map, and hence with the difficulty for the convex upsampler. We extract spatial gradients using the Kornia library [22], and consider the L_2 norm on the 6-dimensional vector that comes from extracting the dx and dy gradients for each RGB channel as an edge detector. To obtain a binary edge map, a binary threshold of 8 is used where smaller values are set to 0 and larger values are set to 1. To get the level of detail for a patch, the average value is taken of the binary edge map of that patch. Next, we plot the average end-point-error (EPE) for each level of detail, for which a bin width of 0.02 is used. All samples with level of detail that is larger than the specified domain are placed in the last bucket. The results hereof are provided in Figure 4.

As we observe from the figure, the performance strongly degrades for increasing amounts of detail, which confirms that our hypothetical problem exists. We provide the statistics on the amount of samples per bucket from Figure 4 in Table 1. From this table, note for example that the buckets $(b >= 8)$ only contain 2% of the patches, yet contribute for 13% to the end-point-error. This strongly highlights the importance of our method, as even though the amount of high-detail patches is low, its impact on the end-point-error can be significant.

4.2. Transformers for Convex Upsampler

Next, we investigate the impact of the proposed individual components. **+DC** refers to decoupling the last refinement's convex upsampler, and giving it its own weights. **+FT** refers to concatenating the features from the context branch to the input of the convex upsampler. When TCU is used features are added at all scales, otherwise only the low resolution features are appended to the input. **+TCU(a/b/c)** refers to the use of our Transformers for Convex Upsampling (TCU) with mask size a for the first upsampling step, b for the step, and c for the last step. Lastly, **-AUG** refers to the additional fine-tuning steps with disabled interpolation-based augmentations.

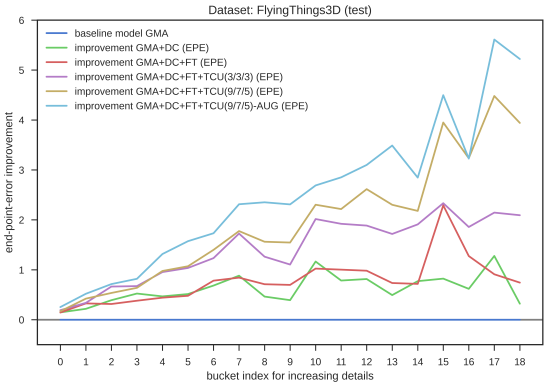

Figure 5. Improvement in end-point-error for increasing amount of detail on FlyingThings3D (test) [19] for a series of our proposed models.

4.2.1 FlyingThings3D

We first investigate the performance for several of our proposed models on the FlyingThings3D [19] test data, after being trained on C+T. The results hereof are shown in Figure 5. From this figure we observe that all our proposed changes result in improvements over the previous model that did not have the change. This is as expected, as we do not expect strong relations between each of our proposed changes, as each of our proposed changes aims at providing improvements in a different way. While these results are good, it is also important to consider situations where this might not be the case. Note that this result is calculated for the test data of the dataset that the model was trained on, so there is no measure of cross-dataset generalization here. For cross-dataset generalization performance, we consider its performance on Sintel Clean [3], which is done next.

4.2.2 Sintel Clean

We create the same graph as before, but now for the Sintel Clean [3] dataset. The results hereof are shown in Figure 6. In general, similar patterns as for FlyingThings3D [19] are observed. However, the gap between with and without (-AUG) is no longer as clear as before. This is somewhat unexpected, as we do not consider bilinear interpolation artifacts in the flow maps of the training data to be a good thing. Possibly, interpolation artifact on edges is a good thing for cross-dataset generalization. Reason for this could be that interpolation on edges results in the 'safe choice' for a model as it provides values around the mean, which then

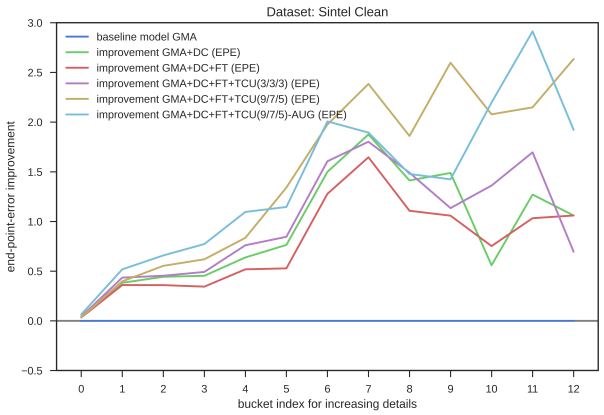

Figure 6. Improvement in end-point-error for increasing amount of detail on Sintel Clean (train) [3] for a series of our proposed models.

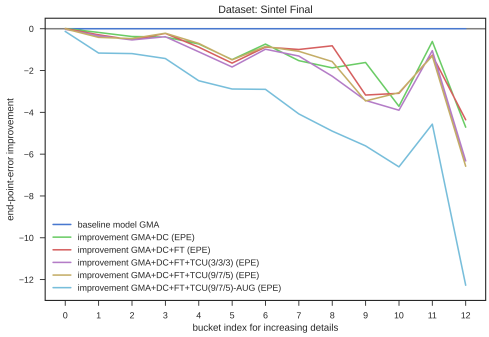

Figure 7. Improvement in end-point-error for increasing amount of detail on Sintel Final (train) [3] for a series of our proposed models.

on average provides similar performance as sharp edges that are sometimes wrong, when evaluated on the end-point-error. We find a similar result in the fact that GMA+DC outperforms GMA+DC+FT. Making it easier to align edges is apparently not always a good thing for cross-dataset generalization.

An interesting result from this is that increasing the mask size for TCU from (3/3/3) to (9/7/5) again provides clear improvements. This, together with the same result for FlyingThings3D [19], provides empirical evidence for our hypothesis that a larger mask size can increase the likelihood of the existence of the convex combination, which in turn improves performance.

4.2.3 Sintel Final

For Sintel Final [3] there exist some important differences. Mainly, Sintel Final contains many strong blurring-based effects in an attempt to mimic real-world camera effects such as motion blur. This introduces a very important aspect; aligning flow with image edges is not a good thing. For Sintel Final, we would instead like to have a model that learns where the actual object edges are based on an image that contains lots of motion blur. To inspect the performance, the same graph as before is generated, but now for Sintel Final [3]. The results hereof can be found in Figure 7. As expected, every step we take towards better aligning flow with objects edges, degrades the models performance for this dataset. Interestingly, removing the bilinear interpolation artifacts from the model output (-AUG) causes a steep decrease in model performance. Clearly, bilinear interpolation artifacts provide an advantage here. This again confirms our earlier idea that possibly bilinear interpolation artifacts on edges are a 'safe choice' when evaluated with the end-point-error, as the values are around the mean. If this is indeed the case, it makes sense that these artifacts help for strong cross-dataset generalization where edges do not align with flow. However, we wonder whether this effect is actually as desired. These interpolation artifacts around the mean of two opposite movements, would indicate that a pixel has $(0, 0)$ displacement. In practise, this is not the case; pixels on the motion boundary can never actually be stationary in such case. This can be of large importance to upstream tasks where the optical flow is used, as it should be noted what the actual meaning is of the presence of this bilinear interpolation artifact in the context of optical flow.

Overall, our +DC+FT+TCU(9/7/5) model sets a strong new baseline on all datasets, except Sintel Final. This is shown in Table 2. While cross-dataset generalization is an important aspect of optical flow models, we do not believe it to be realistic to build a model that generalizes to such impactful motion-blur artifacts that are not at all in the training data. It is important to note that generalization to KITTI-15 [20] is good, even though it consists of actual real-world images, taken by a camera. This raises the question whether Sintel Final is actually a good representation of the real world, or simply considered to be very difficult due to its, possibly too strong, blurring effects.

4.3. General Comparison

While the C+T can be seen as a good measure of cross-dataset generalization, training to a specific dataset is also considered interesting. Therefore, we integrate our approach on GMA [15] for the C+T+S+K+H setting. While theoretically TCU integration is possible in almost all current-day models, we leave its implementation and impact on performance for future work. We report the overall performance in Table 3.

5. Discussion

Overall, we observe that our -AUG training scheme decreases the amount of the bilinear interpolation artifacts in

Training Data	Method	Sintel (train)		KITTI-15 (train)		Sintel (test)		KITTI-15 (test)
		Clean	Final	F1-epe	F1-all	Clean	Final	F1-all
	HD3 [33]	3.84	8.77	13.17	24.0	-	-	-
	PWC-Net [26]	2.55	3.93	10.35	33.7	-	-	-
	LiteFlowNet2 [11]	2.24	3.78	8.97	25.9	-	-	-
	VCN [32]	2.21	3.68	8.36	25.1	-	-	-
	MaskFlowNet [35]	2.25	3.61	-	23.1	-	-	-
	FlowNet2 [12]	2.02	3.54	10.08	30.0	-	-	-
	DICL-Flow [30]	1.94	3.77	8.70	23.6	-	-	-
C+T	RAFT [28]	1.43	2.71	5.04	17.4			
	RAFT (recomputed)	1.42	**2.69**	5.01	17.5	-	-	-
	RAFT+ALL (ours)	1.26	2.74	4.92	17.4	-	-	-
	RAFT+ALL-aug (ours)	1.28	2.93	4.90	17.5	-	-	-
	GMA [15]	1.30	2.74	4.69	17.1	-	-	-
	GMA (recomputed)	1.31	2.75	4.48	16.9	-	-	-
	GMA+ALL (ours)	1.21	2.77	**4.47**	17.0	-	-	-
	GMA+ALL-aug (ours)	**1.18**	3.01	4.50	**16.6**	-	-	-
C+T+S+K+H	LiteFlowNet2 [11]	(1.30)	(1.62)	(1.47)	(4.8)	3.48	4.69	7.74
	PWC-Net+ [27]	(1.71)	(2.34)	(1.50)	(5.3)	3.45	4.60	7.72
	VCN [32]	(1.66)	(2.24)	(1.16)	(4.1)	2.81	4.40	6.30
	RAFT [28]	(0.76)	(1.22)	(0.63)	(1.5)	1.61*	2.86*	5.10
	GMA [15]	-	-	-	-	**1.39***	2.47*	5.15
	GMA (recomputed)	(0.63)	(1.05)	**(0.58)**	**(1.3)**	-	-	-
	GMA+ALL (ours)	(0.58)	(0.97)	(0.62)	(1.4)	1.45*	**2.44***	-
	GMA+ALL-aug (ours)	**(0.55)**	**(0.90)**	**(0.58)**	**(1.3)**	1.44*	2.47*	**5.03**

Table 3. General comparison of our proposed models against other works. Here, ALL refers to our '+DC+TCU(9/7/5)+FT' setting. Furthermore, C+T refers to training on FlyingChairs [6] and then on FlyingThings3D [19]. Next, C+T+S+K+H refers to training on a mix of data from FlyingChairs [6], FlyingThings [19], Sintel [3], KITTI-15 [20], and HD1K [16]. The values in parentheses '()' are calculated on training data that the model was already trained on. *The warm start strategy is used as described by RAFT [28].

the model output, as can be seen in our submission to the Sintel public scoreboard. Possibly, completely removing these artifacts would require longer training without augmentations. However, this will likely cause an overfit to the training data, which in turn will decrease performance on the test data. Therefore, we leave a better solution to this for future work. Most importantly, the presence of bilinear interpolation artifacts might result in good generalization when evaluated using the end-point-error, but it is important to realize its meaning, which in the context of displacement as is the case with optical flow, is completely wrong. We believe that ensuring that this interpolation artifact is no longer present in the flow maps is of great importance with regards to its meaning and correctness in the context of optical flow, and can be of great importance for upstream tasks.

To conclude, we find interesting results by reconsidering the design of the convex upsampler and the bilinear interpolation on the flow in the training data. In general, our methods +TCU, +DC, +FT, and -AUG all seem to achieve a lower error on the training data, meaning that it has achieved a better fit. This was our initial goal, and therefore we can confirm that our changes have the desired effect. However, generalization is an important aspect of optical flow models. As such, we ask for careful consideration for adopting our methods when aligning flow with image edges is no longer a good thing, as in such case our method may not result in improvements. However, when aligning flow with object edges is considered a good thing, we show in the cross-dataset generalization setting (C+T) that all our proposed changes can provide improvements.

References

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[2] John Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. *Advances in neural information processing systems*, 2, 1989.

[3] Daniel J Butler, Jonas Wulff, Garrett B Stanley, and Michael J Black. A naturalistic open source movie for optical flow evaluation. In *Computer Vision–ECCV 2012: 12th European Conference on Computer Vision, Florence, Italy, October 7-13, 2012, Proceedings, Part VI 12*, pages 611–625. Springer, 2012.

[4] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[5] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[6] Alexey Dosovitskiy, Philipp Fischer, Eddy Ilg, Philip Hausser, Caner Hazirbas, Vladimir Golkov, Patrick Van Der Smagt, Daniel Cremers, and Thomas Brox. Flownet: Learning optical flow with convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2758–2766, 2015.

[7] Ali Hassani and Humphrey Shi. Dilated neighborhood attention transformer. 2022.

[8] Ali Hassani, Steven Walton, Jiachen Li, Shen Li, and Humphrey Shi. Neighborhood attention transformer. 2022.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[10] Zhaoyang Huang, Xiaoyu Shi, Chao Zhang, Qiang Wang, Ka Chun Cheung, Hongwei Qin, Jifeng Dai, and Hongsheng Li. Flowformer: A transformer architecture for optical flow. *arXiv preprint arXiv:2203.16194*, 2022.

[11] Tak-Wai Hui, Xiaoou Tang, and Chen Change Loy. A lightweight optical flow cnn—revisiting data fidelity and regularization. *IEEE transactions on pattern analysis and machine intelligence*, 43(8):2555–2569, 2020.

[12] Eddy Ilg, Nikolaus Mayer, Tonmoy Saikia, Margret Keuper, Alexey Dosovitskiy, and Thomas Brox. Flownet 2.0: Evolution of optical flow estimation with deep networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2462–2470, 2017.

[13] Azin Jahedi, Maximilian Luz, Lukas Mehl, Marc Rivinius, and Andrés Bruhn. High resolution multi-scale raft (robust vision challenge 2022). *arXiv preprint arXiv:2210.16900*, 2022.

[14] Azin Jahedi, Lukas Mehl, Marc Rivinius, and Andrés Bruhn. Multi-scale raft: Combining hierarchical concepts for learning-based optical flow estimation. In *2022 IEEE International Conference on Image Processing (ICIP)*, pages 1236–1240. IEEE, 2022.

[15] Shihao Jiang, Dylan Campbell, Yao Lu, Hongdong Li, and Richard Hartley. Learning to estimate hidden motions with global motion aggregation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9772–9781, 2021.

[16] Daniel Kondermann, Rahul Nair, Katrin Honauer, Karsten Krispin, Jonas Andrulis, Alexander Brock, Burkhard Gussefeld, Mohsen Rahimimoghaddam, Sabine Hofmann, Claus Brenner, et al. The hci benchmark suite: Stereo and flow ground truth with uncertainties for urban autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 19–28, 2016.

[17] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10012–10022, 2021.

[18] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[19] Nikolaus Mayer, Eddy Ilg, Philip Hausser, Philipp Fischer, Daniel Cremers, Alexey Dosovitskiy, and Thomas Brox. A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4040–4048, 2016.

[20] Moritz Menze and Andreas Geiger. Object scene flow for autonomous vehicles. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3061–3070, 2015.

[21] Anurag Ranjan and Michael J Black. Optical flow estimation using a spatial pyramid network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4161–4170, 2017.

[22] Edgar Riba, Dmytro Mishkin, Daniel Ponsa, Ethan Rublee, and Gary Bradski. Kornia: an open source differentiable computer vision library for pytorch. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 3674–3683, 2020.

[23] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*, 2018.

[24] 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. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1874–1883, 2016.

[25] Xiuchao Sui, Shaohua Li, Xue Geng, Yan Wu, Xinxing Xu, Yong Liu, Rick Goh, and Hongyuan Zhu. Craft: Cross-attentional flow transformer for robust optical flow. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 17602–17611, 2022.

[26] Deqing Sun, Xiaodong Yang, Ming-Yu Liu, and Jan Kautz. Pwc-net: Cnns for optical flow using pyramid, warping, and cost volume. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8934–8943, 2018.

[27] Deqing Sun, Xiaodong Yang, Ming-Yu Liu, and Jan Kautz. Models matter, so does training: An empirical study of cnns for optical flow estimation. *IEEE transactions on pattern analysis and machine intelligence*, 42(6):1408–1423, 2019.

[28] Zachary Teed and Jia Deng. Raft: Recurrent all-pairs field transforms for optical flow. In *European conference on computer vision*, pages 402–419. Springer, 2020.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[30] Jianyuan Wang, Yiran Zhong, Yuchao Dai, Kaihao Zhang, Pan Ji, and Hongdong Li. Displacement-invariant matching cost learning for accurate optical flow estimation. *Advances in Neural Information Processing Systems*, 33, 2020.

[31] Haofei Xu, Jing Zhang, Jianfei Cai, Hamid Rezatofighi, and Dacheng Tao. Gmflow: Learning optical flow via global matching. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8121–8130, 2022.

[32] Gengshan Yang and Deva Ramanan. Volumetric correspondence networks for optical flow. *Advances in neural information processing systems*, 32, 2019.

[33] Zhichao Yin, Trevor Darrell, and Fisher Yu. Hierarchical discrete distribution decomposition for match density estimation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6044–6053, 2019.

[34] Feihu Zhang, Oliver J Woodford, Victor Adrian Prisacariu, and Philip HS Torr. Separable flow: Learning motion cost volumes for optical flow estimation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10807–10817, 2021.

[35] Shengyu Zhao, Yilun Sheng, Yue Dong, Eric I Chang, Yan Xu, et al. Maskflownet: Asymmetric feature matching with learnable occlusion mask. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6278–6287, 2020.

[36] Shiyu Zhao, Long Zhao, Zhixing Zhang, Enyu Zhou, and Dimitris Metaxas. Global matching with overlapping attention for optical flow estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 17592–17601, 2022.

3 Supplementary Material

3.1 Traditional Optical Flow

Traditional models for optical flow were mostly based on the famous optical flow constraint $I_x u + I_y v + I_t = 0$. Here I are partial derivatives and u and v are the directional x and y components of the flow vectors. Theoretically, solving the optical flow equation gives us the optical flow. Unfortunately, the equation only holds when the brightness of a pixel is consistent over time. In a perfect world, this is the case when the scene is stationary and the camera moves. Unfortunately in practise, that is not generally the case. Moreover, camera related image artifacts can also invalidate the optical flow constraint. The reason for that is that pixel brightness can be influenced by such camera-related artifacts. Another problem is that there is no reasoning involved when solving the constraint directly. When a pixel is in image_1 but not in image_2, as might be the case for occlusions or objects that move out of the image frame, the optical flow constraint no longer holds. In practise, one can encounter many problems like non-stationary scenes, lighting changes, camera artifacts, need for reasoning, and possibly more. Due to these problems, more recent research has largely focused on approaches that resolve these issues.

3.2 Learning-Based Optical Flow

Due to the aforementioned issues with solving optical flow by solving the optical flow constraint, other approaches have been widely researched. Inspired by breakthroughs in computer vision that use learned mathematical models, many optical flow researchers have looked at similar solutions. The strength of learned models for computer vision in general comes from the fact that image interpretation can be fully learned. In traditional image processing approaches, there often exists so called manual feature extraction. That is, to manually define rules that extract useful information (or 'features') from images, such that these features can be interpreted by a computer. Unfortunately, manual feature extraction is a difficult task, often resulting in sub-optimal features. With learning-based computer vision, this is different. There, an image is taken directly to be the input of a learnable model. In other words, the image becomes the features directly. This way, it is left to the learnable model to figure out useful features by itself. Now if unrelated image artifacts from cameras are considered, this problem solves itself in the learnable setting. By allowing a model to learn its own features, computer vision models can learn to ignore certain unrelated image artifacts or lighting changes if ignoring them results in a better overall performance.

While simply using a learned model for feature extraction and still using the optical flow constraint could theoretically work, it does still have the strong drawback that the model cannot perform reasoning. Reasoning might be needed for pixels that exist in image_1, but not in image_2. In an attempt to provide this 'reasoning' ability to the model, one can define

a more general learnable model. In such case, no set of predefined constraints is provided. Instead, an even more general approach is taken. Rather than solving a fixed optical flow constraint, learned based models are simply allowed to learn any useful constraints themselves. In other words, optical flow is formulated as an end-to-end learnable problem. In an end-to-end learnable optical flow setting, a learnable model is defined, which has inputs image_1 and image_2. An optical flow map $flow$ is the set to be the output. Now, similarly to humans, these learnable models learn by trial and error. For a given set of inputs a prediction is made for the $flow$. If it is wrong, the model is slightly corrected, after which the process is repeated. The set of images that is used for this is called the training data. Since the model needs to be learned what is correct, the actual correct output for the training data is needed, such that the model can be corrected if it is wrong. This is called a supervised learning setting. Unfortunately, this requires that there exists correct $flow$ for all (image_1, $image_2$) pairs. This correct flow is called the ground truth $flow_{gt}$.

In many computer vision settings, these ground truth labels are obtained by manually providing them for a given set of data. However, for optical flow this is very difficult. For example, if a model would have the objective to learn if an image contains a cat or not, providing labels for these images would be as easy as labelling them 'cat' or 'no cat'. On the contrary, labels for optical flow $flow_{gt}$ effectively have per-pixel labels. In essence, every individual pixel has its own 'correct answer'. To train an end-to-end learnable model for optical flow in a supervised way, every set of input images needs its own $flow_{gt}$. This in turn requires labelling each individual pixel. As this is extremely costly in terms of human labour, building large-scale datasets of images image_1 and image_2 with corresponding labels $flow_{gt}$ remains a challenge. Fortunately, several approaches to this do exist, as will be described next.

3.3 Datasets

As designing optical flow datasets requires per-pixel labels for the ground truth optical flow map $flow_{gt}$, obtaining large-scale datasets is often difficult. Ideally, one would use real-world data for image_1 and image_2. The reason for this is that optical flow models are often eventually used in practise on real-world images as well. In any learning-based model, there exists the problem of a generalization gap. That is, if the training data differs a lot from the data that will be used at a later time in practise, its performance will degrade. This is because the model will learn to use specific features in the training data to correctly predict optical flow. If the features when used in practise are different from the training data, the performance will consequently be worse. Unfortunately, using real-world image_1 and image_2 is difficult, as taking these pictures is quite simple, but labelling each individual pixel is infeasible. Alternatively, approaches exist where image_1 and image_2 are artificially generated. If the positioning of the objects, and hence the displacement between image_1 and image_2 is done according to some random but known mathematical transformation, this information can be used to directly calculate the per-pixel flow, which can be used as the ground truth labels. This idea is the main inspiration behind many widely used optical flow datasets, as will be discussed next.

3.3.1 FlyingChairs

FlyingChairs was proposed by Dosovitskiy et al. [2015], and uses a collection of real-world images to synthesize an artificial optical flow dataset. As backgrounds, normal real-world images are used. Then, between image_1 and image_2 the background is randomly warped. Due to warping by following a set of defined formulas, the displacement for background pixels can be calculated, and hence the flow displacement vector is known for every background pixel. In the foreground, multiple semi-transparent images of 'flying chairs' are placed. Similarly to the background, the foreground chairs are warped randomly in image_2 relative to their spatial position in image_1. Again, due to knowing the warp, one can calculate the displacement for each pixel, and use this as the flow value. This way, all pixel displacement vectors are known, and so these displacement vectors can be used as the labels for optical flow, without the need for manually annotating any pixels.

3.3.2 FlyingThings3D

FlyingThings3D was proposed by Mayer et al. [2016], and is based on the same idea as FlyingChairs. However, rather than only using chairs as foreground objects, they expand their object collection to include a wide variety of other object types as well. Moreover, rather than using flat 2-dimensional images of objects and warping those as done by FlyingChairs, the FlyingThings3D dataset uses 3-dimensional object models to not only warp the image, but actually create different viewpoints of the foreground objects. Logically, this is closer to the real world, but this also provides more of a challenge for the optical flow model. The reason for this is that from a slightly different angle, an object could look different. Due to that, the model has to learn how certain objects look from different angles, rather than just look for an area in the image that looks similar as would suffice for FlyingChairs.

3.3.3 Sintel

The Sintel dataset Butler et al. [2012] takes a different approach and does not rely on any real-world images. Instead, the work proposes the use of a rendered world. This is similar to current day 'computer-generated imagery' (CGI), as used in animated movies. In this CGI world, static scenes as well as dynamic scenes with characters in action are animated. Internally, any CGI movie frame is calculated using a set of underlying equations. Sintel proposes to exploit this, and use it to calculate flow. For a given pixel in image_1, the corresponding position on the object at that pixel location can be calculated. Then, in the subsequent frame image_2, this object can be looked up, and the same position can be used to figure out where the pixel is being rendered there. By obtaining the displacement vector like this, there is again no need for any manual label annotation. For Sintel there exist two datasets: Sintel Clean and Sintel Final. Of these, Sintel Clean is the simplest variant, as it does not contain difficult camera-related artifacts. In contrast, Sintel Final is their second dataset, which consists of the same images but with atmospheric effects, depth of field blur, and motion blur. These types of effects are often what real-world cameras produce as well. Due to these artifacts, Sintel Final image pairs provide a larger challenge for optical flow models.

3.3.4 KITTI

Rather than artificial image construction using real-world images like FlyingThings3D or FlyingChairs, or CGI methods like Sintel, the KITTI Menze and Geiger [2015] dataset takes a different approach. KITTI uses a set of advanced sensors on a driving car to find correct flow labels for the surroundings. One of the main difficulties with optical flow in general is that it only has access to image_1 and image_2 as inputs. On itself, images have no notion of explicit depth. KITTI uses stereo view cameras together with a laser scanner to obtain a lot of information about the scene. They obtain a whole 3-dimensional point cloud, not just a colored image. By fitting geometrically correct models to moving 3D point clouds, optical flow maps can be calculated. One of the biggest advantages of the KITTI dataset is the use of actual real-world image data taken by cameras. Unfortunately, their method does not predict an optical flow vector for each pixel as the provided optical flow ground truths $flow_{gt}$ are sparse. The meaning of sparse is the opposite of dense, in dense there is a label for every pixel location.

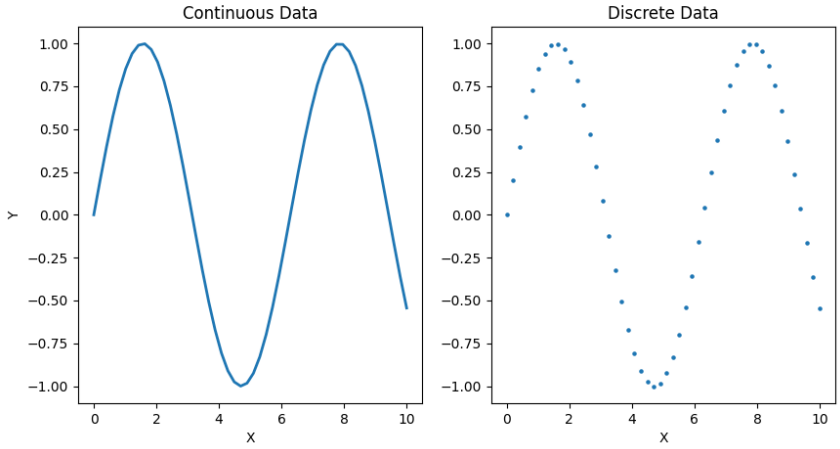

Figure 3.1: **Left:** continuous data, every X value has a corresponding Y. **Right:** discrete data, Y is only defined for specific values of X.

3.4 Prior Knowledge

Dot-Product

The dot-product is the inner-product in Euclidean space. The dot product is an operation that can be performed on two vectors as following. For a vector $v = (v_0, v_1, ..., v_{n-1})$ and another vector $u = (u_0, u_1, ..., u_{n-1})$ the dot product is defined as $\langle u, v \rangle = (v_0 u_0 + v_1 u_1 + ... + v_{n-1} u_{n-1})$. Generally, such a dot product is used as a similarity metric between two vectors.

Manhattan L_1 and Euclidean L_2

When calculating distance, there exist different approaches. For example, in Euclidean space the distance between two points is defined by $d(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$. The norm of a vector (the length) in Euclidean space, is referred to as the L_2 norm. An alternative to Euclidean space is the Manhattan space. There, a distance is calculated as $d(a, b) = |a_x - b_x| + |a_y - b_y|$. In other words, the distance is simply defined by the sum of the absolute differences between the coordinates. The norm of a vector (the length) in Manhattan space is called the L_1 norm.

Discrete and Continuous

In this thesis both discrete and continuous data are considered. A visual example to illustrate the difference is shown in Figure 3.1. To explain the difference, consider the mathematical formula $y = sin(x)$. In this representation, there exists a y value for every possible real-valued x. Effectively, this allows for mapping infinitely many input values x to a corresponding output. In other words, there exists an output value for the whole continuous

input domain. In contrast, discrete representations are not defined everywhere, but only for specific values of x. An example hereof would be to consider an image of some real-world phenomena. Our computers generally use a discrete representation to store image data. The reason for this is that individual pixels are used, which gives the image a corresponding resolution. Now consider that for a discrete image representation of pixels, one would want to know the value (or color) of a pixel. This is well defined, and is as simple as using the discrete representation to lookup the value that corresponds to the pixel coordinate. However, things get difficult when a value is needed at a location that is not exactly on a pixel coordinate. While all sorts of interpolation or guessing techniques can be used to reason what the most likely value is, it is simply not explicitly defined in the discrete representation. But, if pixel values are considered to be a collection of real-world measurements, simply doing an extra observation on the real world for this specific location would provide the actual correct value. Here, the spatial position in the real world is a continuous spectrum, while an image is simply a discrete representation thereof.

Data Distributions

Data can be seen as a collection of measurements of some real-world phenomenon. For example, consider a small collection of images of cats U. Each of these images provides a view of some real-world phenomenon that contains a cat. Now consider how many of such possible cat images could exist. In essence, their could exist infinitely many cat images (assuming some continuous image representation is used), each time with a small change somewhere in the image compared to another cat image. This collection of all possible cat images would be called the distribution of all cat images D. Effectively, the small collection of cat images U, comes from the data distribution D. This is important for learning-based models, as one usually only has access to U. However, unseen data that will be processed by the model when it is used in practise, will come from D, and not necessarily from U. Therefore, it is important to consider that the objective is generally to fit a learning-based model to some distribution D, not to some collection of data U. However, describing D exactly is difficult, and hence learning is often done on U instead, but it is important to know the difference.

Linear or Convex Combination

To calculate a linear combination of a sequence of elements each element is given a weight after which everything is summed. An example hereof can be given for a sequence of inputs as (x_0, x_1, x_2). Then, the linear combination is given as $w_0x_0 + w_1x_1 + w_2x_2$. Note that this definition is not solely limited to scalar values, as is the case when (x_0, x_1, x_2) are just numbers. If (x_0, x_1, x_2) are vectors, the same operation can be applied, as vectors allow for both summing and multiplication. In general, a linear combination can be taken of any sequence of elements as long as the elements can be multiplied with a scalar weight, as well as be summed. A specific version of a linear combination is a *convex* combination. In essence, a convex combination is highly similar to a linear combination, but adds an extra restriction to the weights $(w_0, w_1, ..., w_{n-1})$. This restriction is namely that the weights must be positive and sum to 1. In other words, the weights must form a discrete probability distribution. Here, discrete refers to it being a series of values, rather than be a smooth line for a continuous input domain, and a probability distribution requires all values to be positive and sum to 1. While the difference between a linear and convex combination

might not seem that important, it should be noted that convex combinations are a lot more restrictive. If no negative weights can be used and the weights have to sum to 1, note that no weights larger than 1 can exist. Effectively, the complexity of the linear combinations that can be made is strongly reduced. This is important, because in learning-based models using a less complex function can actually be advantageous. For example, if it is known that the optimal solution is likely to be a convex combination rather than a linear one, setting this restriction can help to reduce the search space, which could make it easier for the model to converge to a solution faster. The search space is the continuous space of all learnable model parameters.

3.5 Deep Learning

3.5.1 Learning-based Models

In general, the so called 'learning-based models' refer to all mathematical models that map a series of inputs to a series of outputs, using a learnable set of parameters. Effectively, a function as simple as $y = ax$ where a is a learnable parameter, x the input, and y the output, could already be seen as a learnable model. However, it is not really a 'powerful' model. Here, powerful expresses the complexity of the function that can be learned. In many learning problems, such a simple function does not suffice as the underlying patterns in the data are a more complex than what can be captured by the function.

A popular approach to learning-based models is deep learning. In deep learning, the general idea is to stack together many smaller learnable models. The main choice for these smaller models is often a relatively simple linear model. However, stacking together many linear models, still makes the final model simply a linear mapping as well. In an attempt to resolve this, deep learning uses non-linear activation functions after most linear layers. This section will describe the high-level idea of deep learning, their common building blocks, and why deep learning is considered a powerful learnable model for many current day problems.

3.5.2 Objective

In deep learning, there usually are no manually set constraints or pre-defined heuristics. Instead, all underlying logic is left to be figured out by a learnable model. This on itself is quite a powerful idea, as deep learning models can learn to make predictions from data that might not be completely understood by humans. An example of this is optical flow. There, humans are able to manually annotate flow, yet are unable to provide a fixed set of equations to be solved that solves optical flow from images in a general case. When a deep learning model is created, the learnable parameters are randomly initialized. At that point, they likely do not capture any useful patterns in the data. Then, the model is trained. Training a deep learning model is based on the idea of learning by trial and error. To calculate what the error is for a given output, a so called loss function is defined. This loss function can be used to back-propagate the gradient, which in turn can be used by an optimizer to update the learnable model parameters.

Supervised, Unsupervised, and Self-Supervised

In general most deep neural networks are trained by trial and error. However, there exist many differences in how the error is calculated. Perhaps the simplest case is a supervised setting. In a supervised setting, a dataset is created where each sample has corresponding labels, the ground truths. To calculate the error, the predicted value is compared against the ground truth using the loss function. However, deep learning models generally need thousands of images at minimum for training, preferably even a lot more. Labelling all these images can quickly become costly, especially when labelling the data is a difficult task. Therefore, alternative approaches like unsupervised learning are also widely researched. In unsupervised learning, a large collection of training data is provided, but without any labels or ground truths. In such case, one has to define a loss function that does not require a ground truth. The important question that remains is how one can define an 'error' if the correct answer is unknown. A popular variant of unsupervised learning is self-supervised learning. The idea with self-supervised learning is that the labels are extracted from the data in an automated way. For example, if a learned model has the objective to colorize an image based on a gray-scale image, such a dataset could easily be built by simply taking many colored pictures and convert them to gray-scale. In such a case, a model could have the learning objective to reverse this operation: make colored pictures from grayscale, where all ground truth labels are known.

Several similar ideas exist, of which an interesting one is auto-encoders. The idea of an auto-encoder is to define a learnable model that has the objective to output the input, as $f(x) = x$. So, ideally, for a given input image, the predicted output image is the same. Now, in the design of the learnable model, one can ensure that there exists a so called bottleneck, which is a layer that has a significantly smaller representation than the size of the input image. If with this bottleneck the network is able to the predict the output from the input, this bottleneck representation must be a smaller encoded representation of the image. Effectively, without providing any explicit encoding techniques, such a network can learn to compress data into a smaller representation. The part of such a model before the bottleneck layer is called the encoder, while the part after the bottleneck is called the decoder. In video compression such an approach can be used where one system encodes the video using the encoder, sends the compressed data over the internet, and the receiving party uses the decoder (which was sent to it in advance) to reconstruct the video again.

Loss Function

Now consider the supervised case, as this is what is used by the optical flow models that are discussed in this thesis. When an input sample is processed by an untrained model, the output is calculated using the random initial learnable parameters, which does likely not result in the correct output. To correct the model, the error needs to be provided, or in other words, how much the model was wrong. To obtain the error, the correct output is required; the ground truth. Fortunately, this is available for the supervised learning setting.

To calculate the error, many approaches exist. For a given model prediction $\hat{y}$ and a ground truth value y, the L_1 error would be defined as $error_{L_1} = |\hat{y} - y|$. Note that this prediction is not necessarily a single scalar value, and may be a vector of multiple scalar values, each corresponding to an output of the deep learning model. In optical flow, each pixel would have its own $\hat{y}$ with corresponding y. However, to calculate the error, a single scalar value is needed. An easy way to do this is to use the mean of the individual errors. While L_1 error

is often used, L_2 can be used in a similar way and is defined as $error_{L_2} = (\hat{y} - y)^2$. When taking the mean of these errors, this is called the mean squared error, or MSE for short. In general, the error function is often referred to as the 'loss function'. The idea behind this is that the outcome of that function is to be minimized.

Choosing the right loss function might not always be trivial, as each loss function tends to have certain specific properties. For example, L_2 loss squares the error, which gives high penalty for model predictions that make very wrong predictions. While this might sound good in theory, practise often shows that datasets are not perfect. For example, datasets can have wrong labels by accident, or have labels that are correct, but cannot be learned from the data. In such case, if one were to give high error to these samples, unwanted extreme parameter updates follow, which could hinder model training. Another example of this is when there exist outliers in the data. An outlier is a sample in a dataset that is very different from the rest of the data. If it is desired to force the model to learn the correct predictions for these samples, L_2 loss would likely be the best choice, as it strongly penalizes large error. In contrast, L_1 does not give this high penalty for very wrong samples, as it simply does not use the square of the error. This makes L_1 a more suitable loss function if it is known that the data contains wrong samples or outliers which ideally should be ignored. In such case, L_1 loss could train the model more easily. Overall, many more loss functions exists, some of which are for instance hybrid variants of L_1 and L_2 loss that tend to take benefit of both approaches. In general, there is no single best loss function.

For optical flow models, L_1 loss is generally adopted. A reason for this is when an object in image_1 moves outside of the image, and is therefore not in image_2. In such case, as many datasets are artificially generated, the computation that calculated the flows during dataset construction did know where it moved the object, and thus provides labels for these pixels in the ground truth flow map. However, it would be impossible to tell exactly where an object moved in such case, given solely image_1 and image_2. This is what was referred to as samples that have correct labels, but simply cannot be learned from the data.

Evaluation

To evaluate how good a trained model is, an evaluation metric can be used. On itself, the loss function can already be used for this. However, the loss function is defined such that it can be minimized by the model. Most importantly, it must be differentiable. But, it is not always the case that the loss function gives a good human-interpretable impression of how well the model performs. For example, an evaluation metric could simply be the accuracy; so how many of the samples it predicted correctly. However, accuracy on itself is not defined by a continuously differentiable function, as it simply counts the amount of correct samples. In such case, accuracy can be calculated to give an easy human-interpretable evaluation metric, but it cannot be used by the model to train. A similar case with a different reason for an alternative evaluation function exists for optical flow, as is discussed next.

In general, optical flow models are evaluated using the average end-point-error (EPE) metric. This metric is calculated as the average of the errors of all individual pixels. That is, for every pixel in image_1, a flow vector $\hat{f}$ is predicted, which is compared against the correct flow vector f as error $= \text{dist}(\hat{f} - f)$ where dist refers to the distance in Euclidean space, or in other words, the L_2 norm. The final model is then evaluated by taking the mean of the errors for all pixels from all images. Interestingly, this L_2 loss is actually nicely differentiable, but many optical flow models minimize L_1 loss instead. The reason for this is with regards

to impossible samples. Remember that the L_2 loss strongly penalizes very wrong samples. A problem with optical flow is that in artificially labelled datasets, objects can move behind something else, or move out of the image frame completely. In such case, it is impossible to predict the correct flow vectors given solely image_1 and image_2. Using the L_2 loss, these impossible examples get high errors, which causes large updates to the model parameters which in turn hinders model convergence. Instead, L_1 loss does not use extra high penalty for very wrong samples, which helps the model converge by essentially partly ignoring these impossible samples. Interestingly, when minimizing the L_1 loss, the eventual end-point-error, which is calculated using the L_2 norm, is generally lower than when minimizing the L_2 loss directly, due to more stable convergence.

Gradient Descent

For a given set of inputs the output $\hat{y}$ can now be obtained, from which together with y and a loss function, the loss value can be calculated. Next, the objective is to minimize this loss value. A way of doing this is using gradient descent. In gradient descent, the main idea is that for every parameter one can describe its influence on the loss value. Mathematically, this is referred to as the gradient $\frac{\partial L}{\partial p}$ for loss function L with respect to parameter p. This gradient of to the loss function with respect to parameter p indicates how a change in the parameter p influences the loss. To minimize the loss, p needs to be updated such that it decreases the loss value, which can be done using the gradient. Namely, the parameter p can be updated by taking a step in the opposite direction of the gradient, or simply take a step in the direction of the negative gradient. If the positive gradient is used to update p, the loss function will increase, which is not as desired. Note that the negative gradient is just a direction; it points in what direction to move a parameter to minimize the loss. To use this to update p, this can be done using gradient descent as $p' = (p - \alpha(\frac{\partial L(x;y;\theta)}{\partial p}))$, where p' is the updated parameter value, p the original parameter value, and α the learning rate, or the step size. The learning rate is effectively how far the parameter moves in the direction of the negative gradient. Moreover, $\frac{\partial L(x;y;\theta)}{\partial p}$ is the gradient of the loss function L with respect to the parameter p that was calculated on input x, ground truth y, and parameters θ. Note that p is part of θ.

On a high level, gradient descent can be seen as walking over a hilly landscape and simply always following the direction of steepest descent. Note that this approach does not guarantee that the globally lowest point is reached. For example, consider the parameter space as given in Figure 3.2. Here, for some starting values of p, it is likely that gradient descent converges to the local minimum, not the global minimum. However, this plot is given for a single parameter p. Note that some models can have millions of parameters. For number of parameters n_p this plot would become a $(n_p + 1)$ dimensional plot, so the loss landscape is very complex and cannot be visualized as done here for a single parameter. In general, several techniques exist to avoid local minima, one of which is simply choosing an appropriate learning rate. Further techniques are discussed in the Section 3.5.2.

Backpropagation

For gradient descent, the gradient of the loss function L with respect to a parameter p is needed: $\frac{\partial L}{\partial p}$. However, obtaining this for deep complex mathematical models might not

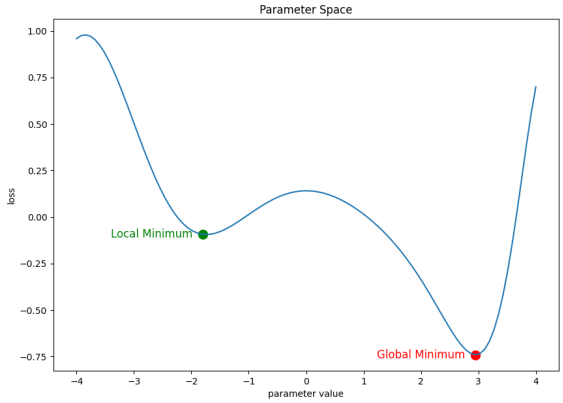

Figure 3.2: Loss landscape for a given parameter p with corresponding loss value L.

always be trivial. Fortunately, there exists a systematic and efficient approach that can be used on any learnable model to obtain the gradient of the loss with respect to any parameter p, as long as each individual component of the learnable model is differentiable, as well as the loss function. Note that individual network components are also called layers. This approach is called backpropagation.

At the output of a learnable model, the gradient of the loss L can be calculated with respect to the outputs of the network; $\frac{\partial L}{\partial y}$. Now consider that a deep neural network consists of many individual layers. Each layer has an input x and an output y. The output of the last layer y was predicted as the network output. The chain rule can be used to obtain the gradient of the loss with respect to the input of the last layer x, rather than with respect to the output y. This can be done as $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}$. Fortunately, $\frac{\partial L}{\partial y}$ is known as this was calculated using the network output and the loss function. Next, $\frac{\partial y}{\partial x}$ differs for each type of layer but can be calculated as long as the layer is differentiable. So, the important thing here is that if the gradient of the loss with respect to the output of a layer $\frac{\partial L}{\partial y}$ is known, it can be used to calculate the gradient of the loss with respect to the input of the layer $\frac{\partial L}{\partial x}$. Now an important observation for backpropagation can be made. That is, the input of a given layer was the output of the previous layer. Effectively, using this chain-rule trick each layer's $\frac{\partial L}{\partial y}$ can be used to obtain $\frac{\partial L}{\partial x}$, which is in fact $\frac{\partial L}{\partial y}$ of the previous layer. This way, the gradient can 'propagate' all the way back trough a network no matter how deep the network is, as long as all layers are differentiable.

Now that at each layer the gradient of the loss with respect to the output $\frac{\partial L}{\partial y}$ is known, the chain rule can be used again to obtain the gradient of the loss with respect to any learnable parameters. This is of great interest, as this allows for the use of gradient descent to update the parameters. For gradient descent $\frac{\partial L}{\partial p}$ is needed, where p is a learnable parameter. Using the chain rule it can be obtained that $\frac{\partial L}{\partial p} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial p}$. Here, $\frac{\partial L}{\partial y}$ is the gradient of the loss with respect to the output of the layer, and is known by propagating the gradient backwards through all layers. Then, $\frac{\partial y}{\partial p}$ is the gradient of the output with respect to the given parameter

that is to be updated. Fortunately, this can be calculated if the layer is differentiable, as it is known how y was computed from input x using parameter p.

In essence, backpropagation can be seen as using the chain rule to rewrite the gradient of the loss with respect to the output, to the gradient of the loss with respect to the input, combined with the observation that each layers input was the output of its previous layer. Within each layer, this gradient of the loss with respect to the output of that layer can be used to calculate the gradient of the loss with respect to any given parameter p within that layer. Using this 'local' gradient, gradient descent can be used to update the parameter values.

Optimizers

While gradient descent with back propagation is theoretically possible, it is not often used in practise on complex models like deep neural networks. This is mostly due to optimization problems. The first problem with gradient descent is that in order to calculate the loss, all the data samples from the dataset are used. For large datasets and complex models, this becomes an incredibly costly operation. An alternative to this is stochastic gradient descent. There, for each step a subset of the whole dataset of samples is used to update the model parameters. Due to not using all samples via random sub-selection, the gradient becomes more a bit more noisy, or random, and hence the name stochastic. Stochastic gradient descent is currently widely used in optimizing deep learning models, but often not in its original form. The reason for this is slow convergence.

Now reconsider the analogy of gradient descent being visualized as a ball rolling down the slope of a hill. In real world, the ball will reach a lowest point rather quickly, due to acceleration. However, the phenomenon of acceleration is not part of the original stochastic gradient descent. There, the direction of the rolling ball is recalculated each time, and a step is taken in that direction. In other words, multiple steps in the same direction do not cause the next steps in the same direction to be larger. In stochastic gradient descent, this idea can also be implemented. There, it is called momentum, which is mainly for providing the effect of acceleration, which in turn speeds up convergence. Moreover, it has two other important properties. First, it can solve an important problem of stochastic gradient descent. The problem with stochastic gradient descent compared to normal gradient descent is that by only using a small subset of the data each time, the gradient can become noisy. By using momentum, a single gradient that is wrong and points in a completely different direction, only slightly changes the actual direction of the rolling ball. This avoids the optimization from directly taking a step in the wrong direction when the gradient is wrong only once in a while. The second important effect of momentum is avoiding local minima. Reconsider the ball rolling of the slope of a hill. If there is slight increase at some point, a ball without acceleration would immediately get stuck (assuming a small learning rate). On the contrary, a ball with acceleration might have enough momentum to move over the small increase, and thereafter continue to roll further down the slope. In terms of optimization, this makes an important difference, as getting stuck in local minima is not desired, but finding the global minimum is.

Many more types of optimizers exists, but for deep learning models these are almost always based on the principle of stochastic gradient descent (SGD) with momentum. An example of such an alternative optimizer is the widely used Adam optimizer Kingma and Ba [2014]. While SGD uses a fixed learning rate, or step size, for all parameters, Adam is a so called adaptive optimization algorithm. For each of the individual parameters, the second and

first order moments of the gradients are used to provide momentum and to scale the update step. As a result, Adam generally converges a lot faster. However, please note that there is no single best optimizer. Each problem and model can have its own properties, and finding an optimal optimizer for a problem usually requires a trial and error approach.

Train/Validation/Test Split

In case of evaluating any trained learnable model, one has to consider what the model is evaluated on. For example, if a dataset contains 1,000 samples and all of these samples are used for training, it becomes difficult to evaluate the model. Reason for this is that when using the same data again, this does not indicate how well the model performs on unseen data, which is the eventual goal. Therefore, a part of the dataset is usually set aside, which is called the validation data. The model should never be able to see any validation data during training. So, the training data is used for training, and the validation data can be used to evaluate the model performance on unseen data. This practise is widely known and generally used. This approach can be taken one step further. The reason for this is that sometimes multiple models are being trained on the training data, and the validation data is used to compare which model performs best on unseen data. However, when multiple models are used and a single best model is chosen, this is not necessarily the best model in general. After all, this model was chosen because it performed best on the specific data of the validation set, not because it performed best on any unseen data in general. All one could conclude from this is that that specific model performed best to the unseen data of the validation dataset, not to any unseen data in general. So, to have a measure of real-world performance, using the validation data will not suffice. For that reason, sometimes the validation set is split again and a test set is kept apart as well. In such case, training data is still used for training, and validation data is used to compare different models against each other. Then, a final model is chosen based on the performance on the validation data, after which this model is evaluated on the test data for a measure of real-world performance. For the test data, it is important to only evaluate on it once as the very last test. Going back and tuning based on test data performance may never be done.

3.5.3 Deep Neural Network Layers

Linear Model

The idea behind a learnable model is to consider the model as a large flexible mathematical function. This learnable function learns to map one or more inputs, to one or more outputs. If n is denoted as the amount of inputs, and m as the amount of outputs, such a function can be written as $f(x_0, x_1, ..., x_{n-1}) \Rightarrow (y_0, y_1, ...y_{m-1})$.

First, consider the scenario where a single input is mapped to a single output, and let the input format be a vector of shape $v = \{v_0, v_1, ..., v_{n-1}\}$. Then, to turn this into an output value, a possible way to do this is to take a linear combination of the input values. That is, weigh each element by a corresponding weight and sum that. This linear combination can be written as $l_0 = (w_0 v_0 + w_1 v_1 + ... + w_{n-1} v_{n-1})$ where the values from w are scalar values. Note that this is the definition of the dot product $\langle w, v \rangle$. However, this dot product only outputs a single value, where it could be desired to map the values from the input to multiple output values. This can be done by simply taking k linear combinations, each

time with different weights w. This way, the size of the output can be chosen freely by setting k. This type of linear combination is the building block of most machine learning models, as they form the simplest operation to map a series of input scalars to a chosen number of output scalars. Note that this operation of taking many dot products for each w can efficiently be implemented using a matrix multiplication of form $y = xW$. However, a standard linear combination always goes through the origin of the coordinate system. To give a bit more complexity to the model a bias term can be added, which makes the formula $y = xW + b$.

Non-Linear Activation

While on itself a linear function can map any size input to any size output, it should be noted that the transformation is linear. Due to that, it is not able to solving any problems that require a non-linear mapping function. A possible way to make simple linear models non-linear, and thus make them able to model more complex functions, is to apply a non-linear activation function. An activation function is a function that is applied to the output of another function, for example a linear model. An example of an activation function is the *ReLU* which is defined as $y = max(0, x)$. where x is the input and y is the output, where max is the element-wise function that takes the highest value. In essence, the *ReLU* function simply sets all values below 0 to 0. Note that the activation function itself has no learnable parameters. The main idea of this approach is that by stacking many linear layers, the resulting model is still linear. Adding these non-linear activation functions between these linear layers allows the model to learn non-linear functions. This is nice because it keeps the building blocks nicely simple and linear, yet makes the overall network a complex function.

Another example of a non-linear activation function is the *sigmoid* function. The *sigmoid* function takes unbounded inputs and maps it to range $(0, 1)$. Similarly, *tanh* can be used as an activation function as well, but maps to the range $(-1, +1)$ instead. Note that for *ReLU*, the range of the output was $[0, \infty)$. Three common activation functions are shown in Figure 3.3. Note that each activation function can have different properties, which can make them ideal for different types of problems.

Neural Networks

With the linear model and non-linear activation functions described, a simple neural network can now be constructed. A simple neural network could simply consist of many stacked linear layers, each with a non-linear activation function applied to their output. Importantly, the universal approximation theorem states that for a sufficiently complex neural network it holds that it can learn to model any mathematical function. This makes neural networks a universal approximator. Note that this theorem only holds when non-linear activation functions are used. This theorem is one of the main reasons why neural networks are so widely adopted for a large range of tasks.

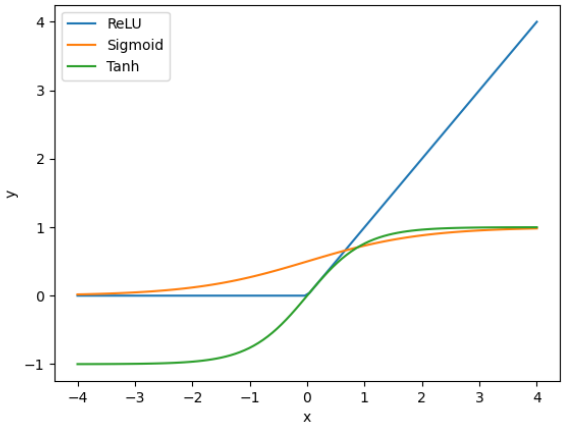

Figure 3.3: The *ReLU*, *sigmoid* and *tanh* functions plotted for the input domain $(-4, +4)$.

3.5.4 Vanishing Gradients and Normalization

A difficulty with neural networks is their optimization. That is, to find which parameters work best for a given problem. While optimization approaches on their own are important factors on the final performance, it should be noted that the design of the learnable model is also of great importance. In back propagation, the gradient of the loss function with respect to each parameter is calculated. So, this information is propagated backwards all the way from the output back to each parameter. This includes backward passes through the activation functions. This can lead to two well-known problems; vanishing gradients and exploding gradients.

Consider the *tanh* activation function from Figure 3.3, and consider an input for this activation function to be $x = 4$. The resulting output value is then close to 1, as can be seen in the graph. However, possibly this value was wrong, and should instead have been -1. In such case, ideally the gradient would indicate that the value of $x = 4$ has to be decreased such that the resulting output gets closer to -1. This is what the gradient is used for in gradient descent. However, there is a problem here, as can be seen in the Figure 3.4 which contains a plot of the gradients of the activation functions. The problem is that the gradient is close to 0 for the input value $x = 4$. So, while ideally the gradient would have provided our optimization with the knowledge that x has to be decreased to get the output towards -1, the gradient now provides no useful information. In neural networks, many layers are often stacked, many of which are non-linear activation functions. If several of these activation functions provide gradient 0 in their backward pass, multiplying all these gradients together as done by backpropagation will result in a gradient near 0 when it reaches the parameter that is to be updated. As can be seen in Figure 3.3, all activation functions have areas where the gradient is near 0. The other problem, exploding gradients, works in a similar way but happens when large gradients are multiplied together, and hence the gradient has 'exploded' to an unusable high value once it reaches the parameter that needs to be updated.

In an attempt to solve both problems, multiple normalization approaches have been widely adopted, such as BatchNormalization Ioffe and Szegedy [2015], LayerNormalization Ba et al.

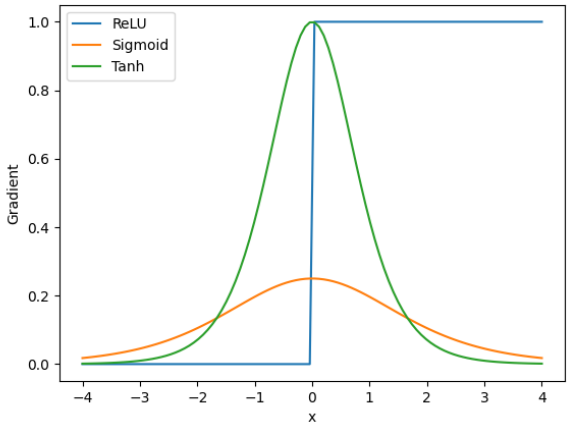

Figure 3.4: The *ReLU*, *sigmoid* and *tanh* gradients plotted for the input domain $(-4, +4)$.

[2016], InstanceNormalization Ulyanov et al. [2016], or GroupNormalization Wu and He [2018]. While each of these approaches normalizes in a slightly different way, the general idea is that the activation values are kept around 0 on average. For example, by subtracting the mean and dividing by the standard deviation, the average of the input is positioned at 0 and the input has unit variance. This would generally provide useful gradients as can be seen in the gradient plot Figure 3.4. So, using normalization approaches, vanishing as well as exploding gradients are less likely to occur. Moreover, the choice of the activation function might also help with this, as the gradient of *ReLU* is either 0 or 1. That means, that as long as the activation is positive, the gradient is 1 and hence the magnitude of the gradient stays unchanged when multiplied together.

3.5.5 Skip Connections

In an alternative approach to solve the vanishing gradient problem, skip connections, or residual blocks He et al. [2016] were proposed. The main idea behind these type of connections is that rather than having each layer feed into the next layer in a sequential manner, the connection also is able to pass around a layer. A way of doing this is to make the individual network components predict the residual instead. That is, to see each layer output as an update to the input, rather than treat an output as a the new output value directly. Mathematically seen a normal layer could be written as $y = f(x)$ where $f(x)$ is the layer with x as input and y as output. For residual connections, this would become $y = x + f(x)$. By doing this, the output value y is computed from two terms, both x and $f(x)$. If the gradient in the backward pass through $f(x)$ vanishes, the x term could still provide a useful gradient. Effectively, $f(x)$ computes the residual that together with the input forms the output value, so $f(x)$ no longer becomes the output value directly. Note that different techniques are not exclusive, so residual blocks may also employ normalization techniques such as BatchNormalization Ioffe and Szegedy [2015].

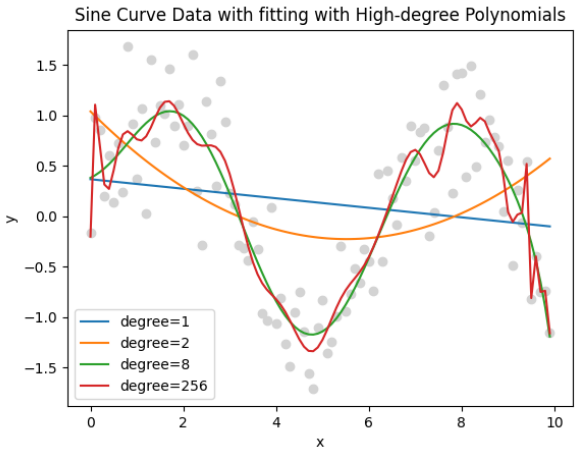

Figure 3.5: Data from a *sin* function with random noise, where different order polynomials are fitted to the data. Note that for degree 1 and 2 a clear underfit is occurring, for degree 8 a decent fit seems to occur, and for degree 256 a clear overfit is visible.

3.5.6 Regularization

A simple linear model when combined with non-linear activation functions, can be used to build a neural network. Given that it is sufficiently complex, this neural network is a universal function approximator. However, it is important to note that the complexity of a neural network is an important factor for its performance. When a neural network is not complex enough, the problem of underfitting can occur. Underfitting means that the data is not sufficiently captured by the model, and hence during training the performance cannot increase any further. A visual example is provided in Figure 3.5. There, discrete data points are plotted from a sinus function with added random Gaussian noise. To fit a mathematical formula to this line, a c-degree polynomial function is used. So, $c = 1$ refers to a simple line. As can be seen in Figure 3.5, this line (blue) does not fit to the data well, no matter where and how it is placed. This is referred to as underfitting. Considering the second degree polynomial (orange), the function is still underfitting to the data. In both cases, the function simply cannot represent a pattern that is complex enough to learn a good fit to the data points. For $c = 8$, the data seems to be captured well. However, if the complexity is further increased to for example $c = 256$, a classical example of overfitting is visible. Surely, this function of $c = 256$ has better performance than $c = 8$ when evaluated on the gray data points. However, $c = 256$ likely performs worse than $c = 8$ when evaluated on unseen data. Here, unseen data refers to other points coming from the same sinus with Gaussian noise distribution where the gray points were also sampled from. In practise, all that usually matters is how well a model performs to unseen data, as training data performance is no longer relevant after training, so overfitted functions should generally be avoided.

So, overfitting is the problem where a neural network learns the data too well. During training, a seemingly perfect prediction is given for a large portion of the data. However, when new data is provided to the model, the performance is a lot worse. An example of overfitting is when a model simply learns to remember all the exact data points that it sees during training. In such a case it does not recognize any general patterns. Logically,

a sample a test-time is something unseen, and an overfitted model will perform worse in such case. Both underfitting and overfitting are problems that one generally wants to avoid. However, this can be difficult for neural networks, as given enough complexity they become universal function approximators. However, there is not a single simple constant that can be tweaked to change the complexity, as was the case with the c-degree polynomial. In general, it is difficult to quantify the complexity of a neural network. However, it is still highly desired to avoid the overfitting and underfitting phenomena. In an attempt to do so, many techniques have been proposed. Usually, this is simply based on trial and error. As such, if the model is underfitting one of two things is usually done. Either the model is made more complex by adding more layers, which makes the network 'deeper', or the representation size of certain layers is increased, which makes the network 'wider'.

Avoiding overfitting is the other general goal in the design of any learnable model. One of the simplest techniques to avoid overfitting is called early stopping. Early stopping means that the performance of the validation data is monitored while training. If the performance on the validation-data starts to decrease while the performance on the training data is still increasing, an overfit is happening. Simply stopping training upon this phenomena is a possible way to ensure the model does not overfit any further. This idea is based on the assumption that easier more general solutions are found first, while models only learn to overfit to specific unrelated patterns at a later stage. Another approach would be to make the network less complex, so to reduce the number of learnable parameters by making the network less deep or less wide. Retraining while monitoring the performance on both the training data and validation data is usually the only way to tell whether an overfit or underfit is occurring.

Another way to avoid overfitting is through regularization. Regularization is a technique that encourages the network to find the easier solutions first, and only use difficult solutions when they strongly benefit the performance of the model. One such approach is L_1 or L_2 regularization. Here, a penalty is added that penalizes parameters with high magnitude. So, the model is encouraged to find solutions where the weight values, or model parameters, are small. The general idea here is that small overall weights find general patterns, and that weights that have much larger values than others find highly specific patterns. So, by penalizing large weights, solutions are less likely to focus on highly specific patterns in the input data. This forces the network to find more general patterns using generally smaller weights, which in turn decreases the chance of overfitting. Note that the network is still able to use large weights, but that doing so will only occur when it provides great benefit to the overall performance. Setting the strength of this regularization constant (or, how strong the penalty is) is usually again done by trial and error. However, similar to setting the degree of the polynomial function, regularization usually allows for tweaking a single constant that describes the model complexity.

Most optimizers like SGD have an easily usable built-in way of enabling this type of regularization called 'weight decay'. For the popular optimizer Adam Kingma and Ba [2014], a fixed 'weight decay' variant AdamW Loshchilov and Hutter [2017] is commonly used, as the original formulas were found to be incorrect with regards to the use of weight decay.

3.5.7 Weight Initialization

Parameters, or often referred to as weights, are the trainable components of learnable layers. However, when a model is created these parameters have to be given some initial value.

While it might not seem important, the initialization of these parameters has large influence on many factors. Some of which are vanishing gradients, exploding gradients, or other convergence issues. For example, consider that the weights are initialized to very large values. Then, the gradients will become large too, which in turn can cause exploding gradients in the backward pass. On the contrary, consider that the weights are initialized such that all activation functions have activations in areas with saturated gradient, with gradient values near 0. In such case, the network will never converge, as parameters are not updated. Overall, many different works have investigated different weight initialization techniques. Some weight initialization techniques even consider the type of used non-linear activation functions, and adapt their initialization based on that. In general, some form of random initialization is used. Examples hereof are Gaussian/uniform initialization, Xavier initialization, and He initialization, but many others exist. Each of these initialization approaches has different design reasons and can be useful in different scenarios. Like many other things in deep learning, finding a weight initialization scheme that works can sometimes require a bit of trial and error.

3.5.8 Convolutions

Regularization can be done in many different ways, another one of which is parameter sharing. This can be especially helpful with images. If an $H \times W$ image with 3 RGB channels is used as the input to a neural network, a possible way of doing this is to just consider every pixel and every color channel as a separate feature. Unfortunately, this scales badly as the amount of input features would be $H \times W \times 3$. For a 256×256 image, this results into $196,608$ input features. When used as the input to a simple linear layer with equally sized output, the number of parameters without bias would be $38,654,705,664$ where the problem regarding complexity and feasibility quickly becomes apparent. Such an approach is generally referred to as a fully-connected approach, where every input feature can have influence on every output feature. Reason for this complexity is that processing an image this way does not consider any so called 'spatial inductive bias'. That is, the model has to learn all possible different spatial orientations, for example for translations and different scales. To clarify this further, consider that humans would generally recognize something when they have learned it, unrelated to the spatial positioning of the object in their field of view. This is what is meant by translation invariance. Ideally, this same property would be made part of neural networks as well. Note that this is not yet the case in a fully-connected approach. If an object is at the left top of the image, it is processed by a completely different set of parameters then when it is placed at the right bottom of the image.

A possible way to effectively 'hard-code' this spatial inductive bias into neural networks is using convolutions. The idea of a convolution is that one slides a window over an input image and treats the values in a window at each position as an individual set of input features. Then, all these sets of features, which are considerably smaller than before due to the window size being many times smaller than the whole image, are processed using a linear layer. The window at every spatial position shares the same linear layer with the same parameters. This is an important component for introducing 'translation invariance' into neural networks for images. So, the same parameters are used to detect certain local patterns, irrelevant of the location of this pattern within an image. One of the important effects of this is that it strongly reduces the required number of parameters, from several billions to several thousands in most cases. This is important, as having too many parameters makes the network vulnerable to overfitting. Taking this 'parameter sharing' approach in

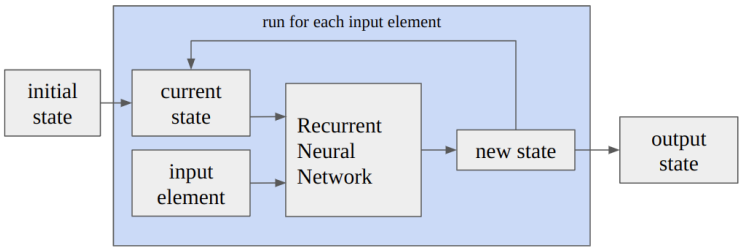

Figure 3.6: General design of a recurrent neural network, where a state is updated sequentially based on the next element in some input sequence. At the end, an encoded single fixed-size vector representation of the input sequence is obtained.

the form of convolutions can help to avoid this, effectively making the use of convolutions a regularization technique. Another thing to consider is the nice computational advantages of such an approach, as effectively every window position can be computed in parallel, making current-day GPU-based convolution implementations fast and efficient for processing images.

3.5.9 Recurrent Neural Networks

A widely known problem for many mathematical models is their fixed input format. For images, this is often avoided by resizing the input image to a fixed size. Something to consider is how one would do this when the input is not always the same size, for example with a sentence consisting of words. A sentence can be a couple words long, but also be a lot longer. If a fixed input format has to be defined, this becomes difficult. If the input sentence is shorter than the predefined size, the additional positions can simply be left empty, but this is computationally inefficient. On the other hand, if the input sentence is longer than the predefined size, part of the sentence will simply not fit in the input, and has to be left out, which can lose important parts of the full input sentence. Simply said, both approaches are not ideal.

In an attempt to resolve this problem, a neural network with variable-size input was proposed. The idea here is that the network handles a single input at the time. By repeatedly taking in a single element, this step can be repeated as often as there are inputs. This makes its use for sentences great, as input elements can be added one-by-one until the end of the sentence is reached. The trainable parameters for this model are the same for every step, and hence the number of parameters is not dependent on the input length. Most importantly, it is possible to add as many inputs as needed. The name of such a network is called a recurrent neural network. The main design for it is that such a model takes two inputs; the ‘new input‘ and the ‘previous state‘. The state can be seen as a temporary storage, where information can be stored for later. For the first input element, there is no previous state yet. This could simply be set to all 0 values, but other initialization approaches exist. The output for the model is simply the ‘next state‘. Effectively, the state is a piece of memory that is updated by a neural network for every input element in a sequential manner. Then, after processing all elements from the input sequence, this state, or memory, is considered to be an encoded representation of the full input sequence. This way, a fixed-size representation

of a variable-length input sequence is obtained. For this reason, recurrent neural network were traditionally widely adopted for modelling sequential inputs like sentences.

Unfortunately, there exist some problems with traditional recurrent neural networks. The main problem is with regards to optimization. In the backward pass, gradients of the loss with respect to each of the weights are calculated. However, for networks that are too deep, gradients usually vanish, even when techniques like normalization are used. Another common problem is that it is difficult to model long-term dependencies using recurrent neural networks. To clarify this further, consider that the last element of a sequence is strongly dependent on the first element. In such case, the ability to model it correctly is determined by the models ability to model long-term dependencies. That is, to know what the first input element was, when it processes the last input element. In case of recurrent neural networks, this is theoretically possible. However, every step provides a new state, and thus completely updates the memory. If the neural network does not know in advance what input elements will follow, it is difficult to decide which parts of the state, or memory, have to be kept and which parts can be overwritten.

Several works have been proposed to resolve both these issues by carefully designing the components of recurrent neural networks. One of the first works to do so was the Long Short-Term Memory network, or LSTM for short. By carefully designing so called 'gates' as part of the architecture, the model has the ability to carefully add information to, or and remove information from the state at each step. In a similar way, gated recurrent units (GRU) Cho et al. [2014] were proposed. The advantages of the GRU is that it is easier and faster to train due to its fewer gates and thus less parameters. Furthermore, several approaches exist for processing sequences using recurrent neural networks. For example, bidirectional LSTM's exist that model the sequence in two directions, both from start to end and from end to start. While such approaches can generally improve performance, there currently exist possibly better approaches that do not rely on recurrent neural networks, as will be discussed next.

3.5.10 Attention

While the idea behind recurrent neural networks is well thought through, it does have some shortcomings. One major shortcoming is the fixed-size limited representation of the state, or the memory. This state is a fixed-size vector that has to hold all the information present in the given input sequence. The problem is that both very complex and long sequences as well as very short and simple sequences share the same model, and hence use the same fixed-size state, or memory. This makes it difficult to choose the size of this fixed-size internal state.

An alternative approach to this sequence-format input is attention Vaswani et al. [2017]. In attention, the important idea is to keep the variable-length representation at each point throughout the model. Before, with recurrent neural networks, LSTM's or GRU's, the size of the internal state vector had to be manually chosen to be a certain fixed size, which would need to encode the whole sequence. In attention, the idea is that a state vector is assigned to each element in the input sequence and that this representation is kept throughout. For sentences, each word in the input would get its own state vector. Then, the attention operator is used to update each individual state vector with respect to the other elements in the sequence. Computationally this is of great advantage as each word can be processed in parallel, while for recurrent neural networks this had to happen sequential. Moreover, instead of having a single globally shared state vector, each word has its own state vector. In

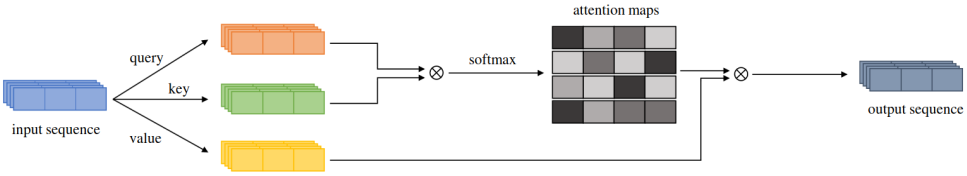

Figure 3.7: Schematic overview of attention where the input consists of a sequence of 4 elements. Each element has a 3-dimensional representation. This 3-dimensional representation is mapped to a 3-dimensional query, key and value vector. Using the query and key the attention maps are constructed. There are 4 attention maps with 4 attention values as there are 4 elements in the input sequence. Using the attention maps, the output for each element is written as a linear, or convex combination of the *value* vectors of the other elements, including itself.

mathematical terms, a recurrent neural network has a single state vector v, while in attention there exists n state vectors; one for every input element as $(v_0, v,1..., v_{n-1})$.

To use attention in such way, a method has to be defined that can update a single element's state vector with respect to a variable length sequence that contains the other elements' state vectors. For this, Vaswani et al. [2017] proposes to use a series of functions; the key, value, and query. These functions are not contextual, so they do not look at any context. Instead, these functions only get as input the state vector of each individual element. From the state vector of an element, a learnable linear layer produces a key, value and query vector. The main idea is similar to how 'search' is done on for example the web. First, the query is used to compare against all other keys. So, for each element in the input sequence the query vector is taken, which is compared against the key vector of all other elements in the input sequence. Note that this requires s^2 comparisons for a sequence length of s. Doing such a comparison can be done in many different ways, but Vaswani et al. [2017] proposes to use the scaled dot-product. The dot-product is used to describe the relation between two vectors, which is scaled down by a fixed constant to avoid large values which can hinder optimization. Once the dot product for a query with every other key has been calculated, this gives s similarities for this specific query, corresponding to one specific input element from the sequence. Then, the these values are turned into a discrete probability distribution, for which the softmax function is used. The softmax function is a function that takes unbounded values as input, and outputs a probability distribution. An example hereof is shown in Figure 3.8. This discrete probability distribution is obtained for each element in the input, and is called the 'attention map'. An attention map corresponds to a single element of the input sequence, and describes the relation with all other elements of the input sequence in the form of dividing its 100% attention over all the sequence elements. Note that the attention map is of size s, and also includes the similarity with itself. A general overview of attention on sequences is provided in Figure 3.7.

Now, reconsider the original goal; updating an element's state vector with regards to the other elements of the sequence. This is where the value vector comes in. For each element an attention map has now been calculated. This attention map describes how much a given element depends on other elements in the sequence, and its sums to 100%. As the last step, the attention map is used to construct new vector values, by using the value vectors. As the attention map is a discrete probability distribution and sums to 1, it can easily be used as the weighted average, which can again be implemented using a dot-product. The new value for a given element is calculated by taking the dot-product of the attention map with

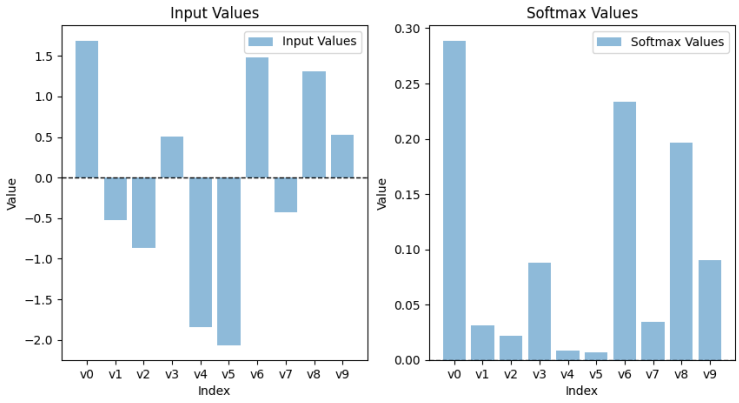

Figure 3.8: An example use of the $softmax$ function. **Left:** random input values that are unbounded as they are sampled from a normal distribution. **Right:** the unbounded values mapped by $softmax$. They are all positive and sum to 1, so they form a discrete probability distribution.

each value of all elements of the input sequence. Effectively, the new value for an element becomes the weighted average of all the other elements value vectors, where the weight is determined via the similarity of the key and query function of the elements, as encoded in the form of an attention map.

3.5.11 Transformers

While attention in general can be implemented on any sequence of elements, it does have some problems. One of which is that attention uses a per-pixel mapping function to obtain the key, query and value vector. These functions have no notion of context, so they cannot incorporate any information about other elements in the input sequence. Hence, if the input features are not contextual to begin with, the query, key and value vectors cannot become contextual either.

In an attempt to resolve this problem, Transformers were proposed Vaswani et al. [2017]. Transformers consist solely of attention layers, followed by an element-wise non-linear block. Many of these Transformer blocks can be stacked upon each other to obtain deep and powerful sequence-processing models. Skip-connections as well as LayerNormalization Ba et al. [2016] are used to make training such a network possible, which allows training of very deep Transformer models. For reference, almost all current-day language models, including the famous ChatGPT (Generative Pre-Trained Transformers) model, use Transformers in their architecture.

Spatial Position Information

Transformers and attention on its own have no notion of relative or absolute spatial position. From the perspective of attention, the sequence of inputs is a set, not a list, so it contains no explicit order. In an attempt to resolve this, many approaches exist. Early works mostly

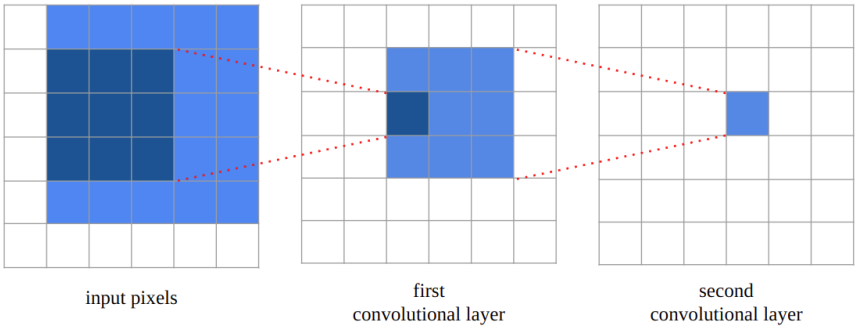

Figure 3.9: Visual overview of the receptive field of a pixel in a second convolutional layer, with respect to earlier layers. Note that 3×3 convolutions are used, and the receptive field in the output of the second convolutional layer is 5×5.

focused on providing the absolute position of an element within a sequence, as the position of a word in a sentence can have an important impact upon its meaning. A possible way of doing this is to concatenate the absolute position of an element in the sequence to the state of the element. Note that without any positional information, a sentence is just a bag of words with no order, which cannot be interpreted easily and possibly have many different meanings. More recently, relative positional information is often used. If one can describe how two elements are positioned relative to each other, this can already be enough and there is no need to know the absolute position.

3.5.12 Receptive Field

While Transformers were initially designed to take sequences as input, they do solve an important problem of convolutions. This problem is namely its ability to model long-range dependencies. Using attention, and hence Transformers, the relation between the first and last word in a sentence can be modelled just as easy as two subsequent words. In convolutions as used for images, this is not the case. As convolutions work by sliding a small window over the image, objects that are far away from each other in an image are never together in a window. However, something to consider is the receptive field of a convolutional neural network. That is, the effective field of view of a given pixel in the convolutional layer's output. To explain this, consider an example where a 3×3 kernel is slid over an image, and thus each new pixel value is based on the 3×3 pixels that were in the window for the given pixel. However, when another convolutional layer is put on top of that, a pixels at the border of the second convolution window already had a receptive field of the 3×3 on the input. Effectively, two stacked convolutions of size 3×3 has a receptive field of 5×5 with respect to the input image. A visual example hereof is shown in Figure 3.9. In theory, stacking many convolutional layers could provide a receptive field that is as large as the whole input image. Unfortunately, this does not often work in practise. Even though theoretically it is possible, simple optimization of convolutional neural networks is known for having issues with modelling long-range dependencies, as the neural network tries to find local features first.

3.5.13 Vision Transformers

Due to the difficulties with modelling long-range dependencies using convolutions, recent works have looked at different ways of using Transformers for image data. The difficulty with that is that Transformers expect sequences as input. While every pixel in an image could be considered as an element of a sequence, this induces a complexity related problem. Remember that to build the attention maps, s^2 comparisons had to be done for sequence length s. So, if every pixel becomes an element of the sequence this results in sequence length $s = (h \times w)$ for an image of size $h \times w$. Due to s^2 comparisons, this quickly becomes infeasible. Moreover, the state vector of each input element in Transformers is usually large, at least several times larger than the 3-dimensional RGB pixel color. Therefore, keeping $(h \times w)$ large state vectors in memory is infeasible and will not work.

To make this idea work, Vision Transformers (ViT) Dosovitskiy et al. [2020] were proposed. The high-level idea is to first split the input image into non-overlapping patches of size 16×16. These individual patches are taken as the input elements in a sequence. Now, each patch has its own state vector with a dimensionality of $16^2 \times 3 = 768$. The 3 comes from the number of color channels in RGB image data. Note that this approach reduces the number of elements in the input sequence by a factor $16^2 = 256$. With this reduced number of input elements, the use of Transformers becomes feasible. Unfortunately, there exist some problems with this approach, one of which is the lack of a strong inductive bias. In the section on convolutions, it was explained why simply using linear layers on raw image data is generally not a good idea. One of the reasons for this was that convolutional neural networks naturally have properties like translation invariance, where the location of an object within the image does not matter, as parameters are shared among locations. This problem strongly relates to ViT networks. While convolutional layers perform local feature extraction and build context by stacking convolutional layers, ViT model global features throughout the network. This makes it more difficult for ViT to quickly learn useful features. As such, ViT models often require enormous datasets and incredibly long training, making them unsuitable for many applications. However, when the training time and the required dataset size is not a problem, ViT can outperform convolutional neural networks on many tasks.

3.5.14 Hierarchical Transformers

While the global modelling capability of Transformers is something that is often considered a good thing, further research has focused on the use of Transformers while maintaining the idea of first extracting local features. One such idea is SwinFormer Liu et al. [2021]. In their work, the idea of local attention is used. In local attention, attention is used within a sliding window in a similar manner to convolutions. This way, the computational complexity is strongly reduced, as there no longer exists the need to compute the similarity with every element of the sequence, but instead only with all elements within the window. Additionally, the network can only learn local features in the first layers, bringing back the hierarchical properties of convolutional neural networks. Similarly, Neighborhood Attention Transformers (NAT) Hassani et al. [2022] has been proposed. By stacking multiple NAT layers the receptive field can be increased, effectively providing it with the advantages of attention while also taking advantage of the hierarchical structure of convolutional neural networks. These advantages include faster training, a reduced number of parameters, the need for less data, and the ability to model global dependencies more easily.

3.6 Deep Learning for Optical Flow

3.6.1 Correlation Volumes

The problem of optical flow can be formulated as a brute force problem. That is, for every pixel in image_1, compare it to every possible position in image_2. However, as image_2 has infinitely many possible positions, doing a brute-force search is not possible. Interestingly, image_2 is already a discrete representation, as it consists of pixels. Inspired by this concept, a brute-force approach can be used to compare every pixel in image_1 with every pixel in image_2. Unfortunately, this is computationally infeasible as this would require h^2w^2 comparisons. In correlation volumes, the same idea is used, but instead on image patches rather than individual pixels. While that does not change anything about the complexity of the calculation itself, it does strongly reduce the number of calculations that need to be done. Rather than doing h^2w^2 comparisons, this is reduced to $(h/p)^2(w/p)^2$ comparisons, where p is the patch size. Comparing two patches is usually done using an operator like a dot-product between the feature vectors of the patches, which gives a single scalar value that describes the correlation between the two patches. The resulting correlation volume is then 4-dimensional of size $((h/p),(w/p),(h/p),(w/p))$. In other words, this correlation volume describes the correlation of every patch in image_1, with every patch in image_2.

3.6.2 Feature Extraction

So far the use of feature vectors for patches was mentioned, but not how they are obtained. When looking for a similarly looking patch, doing this based on the patch content directly is generally a bad idea. The reason for this is that such a representation is not contextual. For example, between image_1 and image_2 lighting might change and a patch might change color completely, even though it still contains the same object. Looking for a comparison solely based on pixel colors will not work in such case. Something else to consider is that something that is in the middle of a patch in image_1, could be right in between two patches in image_2. Doing an exact comparison of pixel values between these patches will likely not result in high positive correlation as is needed. In an attempt to resolve these issues, features vectors from patches are usually generated using some sort of a convolutional neural network. This network has a larger receptive field, so it can model some notion of the context. As this convolutional neural network is learnable, it is left to the network itself to learn useful features such that the dot-product as used for the correlation volume provides useful information to the final objective. Note that this can be used to build patches of size $(p \times p)$ by taking images as input with size $(h \times w)$ and outputting size $((h/p) \times (w/p))$ feature maps. In these feature maps, every cell corresponds to a $(p \times p)$ patch of the input image.

3.6.3 Optical Flow as Optimization

One of the most important recent improvements in optical flow came with RAFT Teed and Deng [2020]. As a result, many recent works all build upon this architecture. Once the correlation volume is obtained, converting it into optical flow is not straightforward. The reason for this is that a certain patch may have multiple matches. For example, if the same object can be found in the image multiple times at multiple locations, there can exist multiple patches with high correlation in the correlation volume. Then, logical reasoning is required

to convert the correlation volume into optical flow. To do so, RAFT formulates optical flow as an optimization problem. For every pixel, the initial displacement is set to $(0, 0)$. Then, multiple optimization steps are taken. At each step, a neural network is provided with information of the surroundings with respect to the current location. Then, the neural network is given the objective to provide a Δ_{flow} with respect to its current location. Doing this repeatedly, effectively makes it similar to gradient descent. However, where for gradient descent the gradient is calculated mathematically, here a neural network has the objective to provide a useful gradient, Δ_{flow}. Doing this step-wise through many layers makes it difficult for the gradient to propagate backwards. As a solution, a convolutional gated recurrent unit is used. This is effectively similar to a normal gated recurrent unit, but with convolutions instead of plain linear layers. This allows it to model a sequence of images, rather than a sequence of elements.

3.6.4 Upsampling

Interpolation

Due to the computational cost of the correlation volumes, almost all recent works output flow at a resolution that is 8 times smaller than the input images image_1 and image_2. This is done by setting the patch size to be 8×8. While this works well in many cases, it does have some drawbacks. One of which is the need for upsampling as the final step. While upsampling in general is not new, the specific properties of optical flow make it difficult to use traditional methods. For instance, if some object moves in direction $-a$, and the background moves in direction $+a$, consider what happens when using interpolation. Using interpolation, new values can be created with values from $[-a, +a]$ (inclusive). However, for optical flow we know that all values $(-a, +a)$ (exclusive) are wrong for sure. For a pixel between these two objects, it either is part of the one object and moves into direction $-a$, or is part of the other object that moves in direction $+a$. Anything else is definitely wrong. For simplicity, we do not consider the cases where values from $(-a, +a)$ (exclusive) are actually the right answer, as this can happen due to it being the actual direction of motion as a result of rotation, but this is not a result interpolation in any way. Unfortunately, this insight makes practically all upsampling techniques that are based on interpolation unsuitable. Alternatively, nearest neighbor upsampling exists. However, when upsampling by a multiple of factor 2, nearest neighbor upsampling will simply make all sub-pixels the color of the parent low-resolution pixel, and hence will not align flow edges with object edges. Therefore, flow edges will become strongly jagged.

Convex Upsampling

The interpolation problem on optical flow becomes worse for larger upsampling factors. In early works on optical flow an upsampling factor of 2 was often needed, making the traditional upsampling approaches using interpolation quite common. However, when the upsampling factor is as large as 8, there are lots of edge values that are all wrong due to interpolation. To solve this, convex upsampling was proposed by RAFT Teed and Deng [2020]. Effectively, their idea is highly similar to interpolation methods, but rather than to interpolate based on the distance to the neighbor pixels, the weights for which pixel value to adopt are predicted as part of the model output. While traditional upsampling methods

usually do not have any learnable parameters, it should be noted that the convex upsampler does have learnable parameters that are used to predict these weights.

Bibliography

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

Butler, D. J., Wulff, J., Stanley, G. B., and Black, M. J. (2012). A naturalistic open source movie for optical flow evaluation. In *Computer Vision–ECCV 2012: 12th European Conference on Computer Vision, Florence, Italy, October 7-13, 2012, Proceedings, Part VI 12*, pages 611–625. Springer.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. (2020). An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*.

Dosovitskiy, A., Fischer, P., Ilg, E., Hausser, P., Hazirbas, C., Golkov, V., Van Der Smagt, P., Cremers, D., and Brox, T. (2015). Flownet: Learning optical flow with convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2758–2766.

Hassani, A., Walton, S., Li, J., Li, S., and Shi, H. (2022). Neighborhood attention transformer.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., Lin, S., and Guo, B. (2021). Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 10012–10022.

Loshchilov, I. and Hutter, F. (2017). Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Mayer, N., Ilg, E., Hausser, P., Fischer, P., Cremers, D., Dosovitskiy, A., and Brox, T. (2016). A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4040–4048.

Menze, M. and Geiger, A. (2015). Object scene flow for autonomous vehicles. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3061–3070.

Teed, Z. and Deng, J. (2020). Raft: Recurrent all-pairs field transforms for optical flow. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part II 16*, pages 402–419. Springer.

Ulyanov, D., Vedaldi, A., and Lempitsky, V. (2016). Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

Wu, Y. and He, K. (2018). Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19.