Deep Networks for Image-to-Image Prediction

Carlo Tomasi

March 14, 2022

1 Introduction

Early applications of deep networks in computer vision focused on object recognition. Recognition networks take an image as input and output a category, or a score for each of a set of predefined categories. As a consequence, recognition networks have the structure of a funnel: They are large at the input and small at the output, since the number of categories is always much smaller than the number of pixels in the input image.

Some more recent uses of deep networks address the estimation of image-like quantities instead. For instance, an *image motion estimation* network takes two frames as input and outputs a motion field. In principle, the motion field has two numbers (the two components of image motion) for each pixel in the input frame pair.[1]

As a consequence, the output of a motion estimation network is also *image-like*, in the sense that the quantities it estimates are in the form of an image with two values per pixel. Information of this type is sometimes called *retinotopic*, in that the output quantities are associated to individual points of the input (the "retina").

Another example of an image-to-image estimation problem is *image segmentation*, which assigns a category to each pixel of the input. For instance, the input could be a satellite image, and the pixel categories could be building, vegetation, road, water, and so forth. This is still a retinotopic output, and the value at each pixel is either a label or a one-hot encoding of it.

Both motion analysis and segmentation are image-to-image estimation tasks, even if the format or number of input and output images is different in the two cases. However, segmentation is a classification task, while motion estimation is a regression task. As we know, the difference between regression and classification, which can be quite substantive for many machine learning algorithm, is rather minor for deep networks from an architectural standpoint, and the main difference concerns loss functions.

For pedagogical continuity with previous notes, the next Section outlines a prototypical architecture used for image motion estimation. This architecture looks like an hourglass rather than a funnel: It starts large, it funnels down to a relatively narrow bottleneck, and broadens out again through a reverse funnel. The bottleneck is meant to force the network to compute abstract representations of the input. This so-called *encoder-decoder* architecture would by itself be unable to maintain a retinotopic representation, since the bottleneck effectively obliterates resolution. *Skip*

[1] In practice, because of both storage and computation time limitations, the outputs of image-to-image neural networks are often at a somewhat lower resolution than the inputs, perhaps by a factor of 2 or 4. This restriction is likely to be lifted as hardware becomes faster and less expensive.

connections are then added that connect layers in the decoder funnel to layers in the encoder at the same resolution.

Since the second funnel goes from small to large feature maps, it requires ways to *increase* resolution, which is often done through an operation called *up-convolution*. Section 2 describes this operation in some detail.

Section 3 then briefly overviews encoder-decoder architecture for image segmentation. The architectural considerations are quite similar to those for motion estimation, so this Section is brief. However, it presents an opportunity to discuss the issue of *class imbalance* that arises when the distribution of class labels in a training set is uneven. This issue arises also in classification problems other than segmentation.

2 Networks for Image Motion Estimation

The input of an image motion[2] estimation network is typically a pair of consecutive frames out of a video sequence. While common, this input format is also very restrictive, as it focuses on a very small period of time, in which the distinction between image changes caused by motion and those caused by image noise, compression, or other artifacts is often small. Estimating image motion from more than two frames, or even in a recurrent fashion from video of unbounded length, is a promising direction for research, and is not discussed in this note.

Loss Training an image motion estimation network aims at reducing a risk function that measures the average *End-Point Error* (EPE), that is, the discrepancy between the true motion field $\mathbf{v}(\mathbf{x})$ and the motion field $\mathbf{u}(\mathbf{x})$ computed by the network. To this end, each training sample is a pair of image frames $f(\mathbf{x})$, $g(\mathbf{x})$ out of a video sequence and the true motion field $\mathbf{v}(\mathbf{x})$ between them, for which the following relationship holds under ideal circumstances[3]

$$g(\mathbf{x} + \mathbf{v}(\mathbf{x})) = f(\mathbf{x}) \; . \tag{1}$$

This equation is the finite (as opposed to differential) version of the Brightness Change Constraint Equation: The same point in the world looks the same in the two frames.

The EPE is typically defined as the average Euclidean distance between $\mathbf{u}$ and $\mathbf{v}$ over the (discrete) image domain Ω:

$$\frac{1}{|\Omega|} \sum_{\mathbf{x} \in \Omega} \|\mathbf{u}(\mathbf{x}) - \mathbf{v}(\mathbf{x})\|^2 \; . \tag{2}$$

Training Data The cost of annotating data sets for image motion estimation is high, because annotation involves specifying the true motion field at every pixel for each pair of frames in the training set. This is a fundamental difficulty, and not just an issue of labor expense, because the aperture problem makes it hard to even know what the true motion field $\mathbf{v}$ is at a particular pixel.

A promising method for addressing this difficulty is to use Computer Graphics (CG) to generate synthetic video sequences. CG can generate very realistic scenes and motions, and the perceptual differences between animated graphics and real-world video is rapidly shrinking. If anything, CG

[2] As usual, the literature uses the terms "motion field" and "optical flow" interchangeably.

[3] We assume again, for pedagogical simplicity, that the images are black-and-white.

images are often of higher quality than real-world video, but then it is not too hard to make images look *worse* (that is, more realistic) by image processing methods!

A key advantage of CG data is that the true image motion can be inferred from knowledge of the 3D models, motion models, and camera models used in the rendering process. Several artificial data sets with ground-truth image motion annotation are available, including the so-called *Sintel* [2] and *Flying Chairs* [3] datasets.

One aspect that makes annotating for image motion *easier* than for, say, image recognition is that for the latter task each image is a training sample, while for image motion estimation virtually every *pixel* (or at least every small image neighborhood) is a training sample. Therefore, annotating a single pair of images amounts to providing hundreds of thousands or even millions of training samples.

An intriguing possibility explored in recent literature [9] is to use *unsupervised* methods to train a deep network for image motion estimation. These methods still require video data, but they require no annotation, and are explored in Appendix 3.

2.1 Supervised Image Motion Estimation

Standard classification tasks in computer vision differ from the regression task of image motion estimation also in the format and size of the output: In motion estimation, the required output motion field has ideally the same resolution as the input image (one motion vector per pixel, rather than one or a few labels per image). This factor has far-reaching consequences on the architecture of the networks used for motion field estimation, as discussed next.

The output of a classification network is a class label, and the number of possible labels is typically much smaller than—and otherwise unrelated to—the number of image pixels. More importantly, the choice of label often depends on large portions of the input image, if not on all of it. As a consequence, typical classification networks have smaller and smaller layers as one moves away from the input. Units in each layer have larger and larger receptive fields, and subsequent layers boil down image information into more and more concise representations. The last layers in the network have all but "forgotten" the detailed topology of the image, and instead encode abstract aspects of the image that are related to the distinctions between different class labels.

Thus, a classification network looks somewhat like a funnel, wide and retinotopic at the input and narrow and abstract at the output. This narrowing, sometimes called a *contraction*, is achieved by pooling (max pooling or average pooling) or strided convolutions.

In a motion-estimation network, on the other hand, the output has about the same height and width as the input, although it may have a different number of channels: two channels for the output and either two (two black-and-white frames) or six (two color frames) channels for the input. As a consequence, the network cannot be a funnel.

However, the abstraction performed by narrowing layers is still useful in motion-estimation networks, because the wider receptive fields in units in deeper layers may be able to see the correct motions better than a narrower field would. For instance, the detail of a tree's canopy in the red square in Figure 1 contains a largely repetitive texture. At testing time, the network may accordingly be unable to determine which of the several plausible displacements is the correct one. When matching two images of the larger detail in the yellow square in the same Figure, on the other hand, a smaller number of displacements may be consistent with the brightness pattern contained in the two images. In other words, the aperture problem is less of a problem when the aperture is larger. Coarser-resolution images may lead to coarse motion fields with high bias

Figure 1: A small (red) and larger (yellow) detail of a tree's canopy.

(because of coarseness) and low variance (because of the comparatively large amount of data used), and images at finer resolutions may be able to refine these fields locally and improve resolution without increasing variance too much.

One way to resolve the tension between the need for abstraction and the need to preserve resolution is to concatenate two neural networks: The first network is the *contracting* stage, or *encoder*, and the second, called the *expanding* stage, or *decoder*, progressively increases resolution back up to input resolution. An example architecture of this type is shown in Figure 2, and is known as the *FlowNet* architecture [3].[4]

The expansion could be achieved by bilinear interpolation, but this would be a fixed expansion, that is, its parameters would not be learned. Instead, the goal is to make the expansion flexible and let the network optimize its parameters during learning. This can be achieved by an operation called *up-convolution*, described next.

Up-Convolution The operation of up-convolution is most easily understood for signals in one dimension, and all the concepts involved extend immediately to multiple dimensions. Consider the strided convolution of signal $f(x)$ with a kernel $k(x)$ that has p elements:

$$g(y) = \sum_{x=0}^{p-1} k(x) f(sy - x) \; . \tag{3}$$

[4]The FlowNet paper also describes a more complex architecture where two separate deep networks constrained to have the same parameters ("siamese" networks) process the two input frames independently before their outputs are merged through a correlation network into a so-called *loss volume*, and are then processed together with a third network. Experiments show that the added complexity does not improve performance significantly.

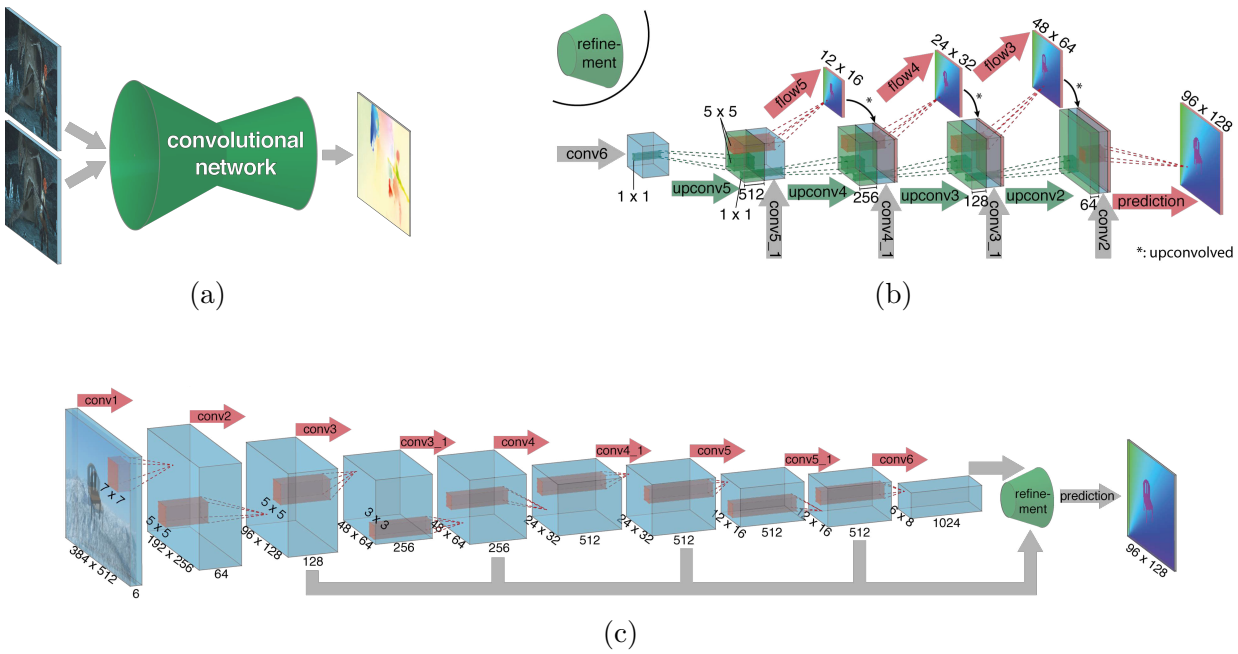

Figure 2: (a) The overall shape of the architecture of FlowNet. The expanding stage shown in detail in (b) is represented by the green truncated cone in (c), which shows the contracting stage in detail. Figures are from [3].

In this expression, s is the stride of the convolution, a positive integer. Since the convolution is linear, it can be written in matrix form as

$$\mathbf{g} = K\mathbf{f} \tag{4}$$

where $\mathbf{f}$ and $\mathbf{g}$ are column vectors that collect all the values of f and g, respectively. If f has m elements and g has n, then K is $n \times m$.

As an example, consider a stride-2 convolution in the 'same' format, with $m = 12$. Then, the output of the stride-1 convolution would have length $n = m$. Since m is even and the stride is $s = 2$ instead, the output g has length $n = m/2 = 6$, and K is a 6×12 matrix. As a result, this convolution decreases resolution by half.

Let the kernel k have length $p = 5$, and let its elements be $[a, b, c, d, e]$. Then,

$$K = \begin{bmatrix} c & b & a & & & & & & & & & \\ e & d & c & b & a & & & & & & & \\ & & e & d & c & b & a & & & & & \\ & & & & e & d & c & b & a & & & \\ & & & & & & e & d & c & b & a & \\ & & & & & & & & e & d & c & b \end{bmatrix} .$$

The corresponding *up-convolution* is defined to be the convolution with matrix representation

$$\boldsymbol{\varphi} = K^T \mathbf{g} \, . \tag{5}$$

A symbol other than $\mathbf{f}$ is used on the left-hand side of this expression, because K^T is not the inverse of K, so one does not get $\mathbf{f}$ back with this product.

This discussion could stop here: An up-convolution is exactly a transformation that can be written in the form of equation 5, just as a (strided) convolution is a transformation that can be written in the form of equation 4. Note that the transformation 5 takes an input with n values (6 in the example) and produces an output with m values (12 in the example). In other words, while a strided convolution (with stride $s > 1$) reduces resolution, an up-convolution (with stride $s > 1$) increases it.

However, neither expression is computationally efficient. For illustration, we picked an example with a small value of n. In practice, n, the number of pixels in one dimension of the input image, will be much bigger than p, the number of kernel coefficients in that dimension. Thus, in practice, the matrix K (or K^T) will contain mostly zeros. While convolution implemented through equation 3 requires $O(pn/s)$ operations, expression 4 requires $O(n^3/s)$, a much bigger number. Because of this, we now describe a way to write up-convolution more efficiently as well, with an expression that has the flavor of equation 3.

To understand the structure of up-convolution, let us rewrite the matrix K^T as a table and mark each column with the entry of $\mathbf{g}$ that it multiplies in the matrix product $K^T\mathbf{g}$:

g_0	g_1	g_2	g_3	g_4	g_5
c	e				
b	d				
a	c	e			
	b	d			
	a	c	e		
		b	d		
		a	c	e	
			b	d	
			a	c	e
				b	d
				a	c
					b

The structure of this table becomes more immediately apparent if zeros are inserted after each sample of $\mathbf{g}$. If the stride is s, insert $s-1$ zeros. Thus, replace $\mathbf{g}$ with the vector $\boldsymbol{\gamma}$ whose entry at position y is

$$\gamma(y) = \begin{cases} g\left(\frac{y}{s}\right) & \text{if } y \stackrel{s}{=} 0 \\ 0 & \text{otherwise} \end{cases} \qquad \text{for} \quad 0 \le y \le sn \; . \tag{6}$$

Here and elsewhere, $a \stackrel{c}{=} b$ means that a and b are equal modulo c, so that $y \stackrel{s}{=} 0$ means that y is divisible by s. The transformation from $\mathbf{g}$ to $\boldsymbol{\gamma}$ is called a *dilution* by a factor s. After dilution,

the table has sn columns rather than n (12 rather than 6 in the example) and is square:

γ_0	γ_1	γ_2	γ_3	γ_4	γ_5	γ_6	γ_7	γ_8	γ_9	γ_{10}	γ_{11}
g_0	0	g_1	0	g_2	0	g_3	0	g_4	0	g_5	0
c		e									
b		d									
a		c		e							
		b		d							
		a		c		e					
				b		d					
				a		c		e			
						b		d			
						a		c		e	
								b		d	
								a		c	
										b	

Since entries in the empty columns of this table multiply zeros in $\boldsymbol{\gamma}$, we can put anything we like in them without changing the product. In particular, we can fill the table as follows:

γ_0	γ_1	γ_2	γ_3	γ_4	γ_5	γ_6	γ_7	γ_8	γ_9	γ_{10}	γ_{11}
g_0	0	g_1	0	g_2	0	g_3	0	g_4	0	g_5	0
c	d	e									
b	c	d	e								
a	b	c	d	e							
	a	b	c	d	e						
		a	b	c	d	e					
			a	b	c	d	e				
				a	b	c	d	e			
					a	b	c	d	e		
						a	b	c	d	e	
							a	b	c	d	e
								a	b	c	d
									a	b	c

This is the matrix for a 'same'-format correlation with $k(y)$, that is, a convolution with the reverse of k,

$$\kappa(y) \stackrel{\mathsf{def}}{=} k(p-1-y) \; ,$$

and can be written as follows:

$$\phi(x) = \sum_{y=0}^{p-1} \kappa(y)\gamma(x-y) \; . \tag{7}$$

In summary:

> *The up-convolution with stride s of signal $g(y)$ with kernel $\kappa(y)$ is the convolution of the s-diluted version of $g(y)$ with $\kappa(y)$.*

Since the matrix corresponding to up-convolution with stride s is the transpose of a convolution matrix, up-convolution is sometimes also called *transposed convolution.*[5]

What matters in the definition of up-convolution is its format, not the values in κ: In a neural network, the coefficients of κ are learned, and all we know is that there is *some* stride-s convolution kernel (namely, $k(y) = \kappa(p - 1 - y)$) from which the up-convolution could be derived as described above.

Of course, the description of up-convolution as a convolution of a diluted signal is merely conceptual. Computationally, it would be wasteful to first dilute the signal and then perform all the multiplications, including the ones by zeros. The efficient way to compute up-convolution is obtained by replacing the definition 6 of dilution into the expression 7 for convolution to obtain

$$\phi(x) = \sum_{y \overset{s}{=} x,\ y=0}^{p-1} \kappa(y)\ g\left(\frac{x-y}{s}\right) .$$

With this implementation, each sample of ϕ requires approximately p/s multiplications, since the condition $y \overset{s}{=} x$ retains one every s terms in the summation, namely, those for which the argument of g is an integer.

Since sampling and dilution are separable operations (that is, they can be applied dimension-wise to a multidimensional signal), the discussion above generalizes immediately to images, or even to signals with higher-dimensional domains, and to vector-valued signals. The dilution factors in different dimensions can be different (although they rarely are in the literature).

Skip Links Up-convolution increases resolution, but it does so only formally, rather than substantively, as there is no way to recreate the retinotopic information that was lost in the encoder through convolutions with stride greater than 1. Because of this, motion field results obtained with a network whose expanding stage has only up-convolution layers tend to be coarse: While the output has as many motion field vectors as each of the input frames has pixels, the motion field map is blurred, with nearby vectors being more similar to each other than in the ground truth, and with poorly localized discontinuities.

To address this issue, so-called *skip links* are added to the network. These links are represented by gray arrows in Figure 2. Each link copies the activation map at the output of a convolutional layer in the contracting stage to the output of the layer in the expanding stage that has the same resolution. This activation is concatenated in the channel dimension to the expanding layer's activation, as shown in Figure 2 (b). In this way, the substantively low-resolution information output by an expanding layer is aggregated with the substantively high-resolution information from the corresponding contracting layer, and the (substantive) resolution of the output motion field map is significantly improved.

Performance As shown in the original paper [3], FlowNet does slightly better (6-8 pixels of End-Point Error, EPE) than the classical (that is, pre-deep-learning) method by Brox *et al.* [1] (7-9 EPE) on standard benchmark sets. FlowNet does not do quite as well as other image motion estimation methods based on neural networks, such as EpicFlow [10] or DeepFlow [12], which achieve EPEs around 4-5 pixels, or even more recent networks, which do almost twice as well [6, 11]. However,

[5] Up-convolution is sometimes also called "deconvolution." This is a misnomer, however, because this term denotes something else altogether in signal processing.

the latter approaches are rather complex compared to FlowNet, and it seems pedagogically more useful to examine the potential of the deep-learning approach in a simple form.

3 Networks for Image Segmentation

Encoder-decoder networks are the workhorses of image segmentation as well. The architecture is very similar, save for the obvious differences at the input and output: One image instead of two at the input, and a *label score map* at the output.

A label score map is a stack of arrays with the same spatial resolution (number of rows and columns) as the input image and one channel per label. If the input image has h rows ("height") and w columns ("width"), and there are K labels, then the label score map is a $h \times w \times K$ array $p(r, c, k)$ and the K values for each r and c are the outputs of a soft-max function. Therefore,

$$p(r, c, k) > 0 \quad \text{and} \quad \sum_{k=0}^{K-1} p(r, c, k) = 1 \ .$$

The map is a bit simpler when $K = 2$: In that case, we would have $p(r, c, 0) = 1 - p(r, c, 1)$, so only $p(r, c, 1)$ is typically output. Class 1 is the class of interest (for instance, "building" in a satellite image) and class 0 stands for "everything else." The score $p(r, c, 1)$ is then for a positive detection, and the array of all scores is often called a *heat map*.

Loss Since segmentation is a (pixel-wise) classification problem, the cross-entropy loss is appropriate at every pixel, and losses over the entire image are averaged to return a per-image loss.

In some segmentation problems, the distribution across classes can be very uneven. For instance, a building segmentation system deployed on sparsely populated areas may have very few building pixels. As a consequence of this *class imbalance*, the trivial predictor that classifies every pixel as "non-building" would achieve very low risk and very high accuracy even with the 0-1 loss measure. Because of this, the training algorithm is likely to return a predictor that is similar to the trivial one. In terms of back-propagation, the feedback signal provided by the few positive examples of buildings is too weak for good training, and there are too many local minima in the risk function that correspond to near-trivial predictors.

The *focal loss* was introduced to address this problem [5]. Instead of the straight cross-entropy loss,

$$\ell_{\text{xe}}(y, \mathbf{p}) = -\log p_y$$

the focal loss is

$$\ell_{\text{f}}(y, \mathbf{p}) = \alpha_y (1 - p_y)^\gamma \ell_{\text{xe}}(y, \mathbf{p}) \ .$$

In this expression, y is the true label, $\alpha_0, \ldots, \alpha_{K-1}$ are positive coefficients that add up to 1, and γ is a real number greater than 1.

The weighting terms α_y and $(1 - p_y)^\gamma$ have separate functions. Specifically, α_k is set to be inversely proportional to class frequency: The smaller class k is, the greater the corresponding α_k. If there are n_k pixels in class k, set $\tilde{\alpha}_k = 1/n_k$ and then normalize to unit sum:

$$\alpha_k = \frac{\tilde{\alpha}_k}{\sum_{j=0}^{K-1} \tilde{\alpha}_j} \ .$$

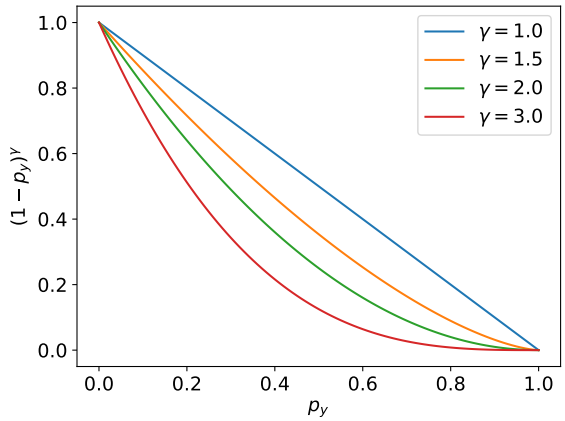

Figure 3: The function $(1 - p_y)^\gamma$ for a few values of $\gamma \geq 1$.

The effect of these coefficients is to boost the loss for more sparsely represented classes.

Since $0 < p_y < 1$ and $\gamma > 1$, the function $(1 - p_y)^\gamma$ is decreasing and convex in p_y, as shown in Figure 3. The effect of this term is then to emphasize training samples with small scores p_y over those with larger scores. Large-score samples are those the classifier finds easy to classify: They are correctly classified and well away from the decision boundary. As a result, the focal loss effectively ignores most of the uncontroversial, correctly classified samples and focuses on the hard ones. These are samples that are either incorrectly classified or classified correctly but with a small margin.[6]

A trivial classifier would misclassify all samples in sparsely represented classes, while many samples in the more populated classes are likely to be uncontroversial. Because of this, use of the focal loss keeps the training algorithm well away from trivial classifiers, and the ill effects of class imbalance are thereby reduced.

Class-imbalance is not specific to segmentation, and the focal loss is often used in other classification problems as well.

Training Data Labeling images for segmentation is easier than labeling image pairs for motion estimation. For instance, satellite images can be labeled with some user interface that allows dragging rectangles over buildings or, for more complex shapes, draw polygons around them. Images labeled via crowd-sourcing are therefore typically used to train image segmentation systems.

Performance One of the earlier systems for image segmentation to use deep learning methods is by Noh *et al.* [8] and uses a now-classic encoder-decoder architecture. Many improvements have been proposed since, in large part to improve on the resolution of the results, which tended to be somewhat blurry in early systems. A survey examines progress in the area in the years 2015-2020 [7].

[6]The *margin* of a sample is some measure of its distance from the decision boundary.

Appendix: Unsupervised Image Motion Estimation

The key difficulty in learning to achieve a small value for the End-Point Error (EPE) defined in expression 2 is to determine the true motion field $\mathbf{v}$ at every pixel $\mathbf{x}$ of each training image pair. In other words, annotation is expensive. An intriguing alternative explored recently [9] is to use equation 1 as a starting point instead. Specifically, one could define a *reprojection error*

$$g(\mathbf{x} + \mathbf{u}(\mathbf{x})) - f(\mathbf{x})$$

that measures the discrepancy between the value image f takes at pixel $\mathbf{x}$ and the prediction $g(\mathbf{x}+\mathbf{u}(\mathbf{x}))$ that could be made of that value assuming that constancy of appearance holds between the two frames. Note that $\mathbf{u}$, the computed motion field, was replaced for $\mathbf{v}$ in this difference: We are interested in how good our guess of the motion field is, not how good the true motion field is.

While the EPE in expression 2 is (the norm of) the difference between two motion field vectors, the reprojection error is the difference between the *color* (or brightness level) of f at $\mathbf{x}$ and the color of g at the point that $\mathbf{x}$ moves to in the time between frame f and frame g. Thus, the reprojection error provides some information on how good the computed motion field $\mathbf{u}$ at $\mathbf{x}$ is.

The square of the reprojection error is essentially the same as the color loss ℓ_c used in variational methods for image motion estimation [1]. If constancy of appearance holds, and if image noise is modest, then guessing the correct motion field at $\mathbf{x}$, that is, letting

$$\mathbf{u}(\mathbf{x}) = \mathbf{v}(\mathbf{x}) \ ,$$

would make the reprojection error small, and one could use this loss (averaged over the whole image domain) for *training* a deep network.

However, the reprojection error provides rather indirect information about the quality of $\mathbf{u}$. In particular, because of the aperture problem, there is often a whole family of motion fields that yield the same reprojection error. Thus, while a good motion field estimate typically yields a small loss, the converse is not true: A small loss could be achieved with very wrong estimates of the motion field, as long as its normal component is correct. In addition, violations of constancy of appearance make the reprojection nonzero even with perfect $\mathbf{u}$ and noiseless images.

The solution is then to augment the loss with appropriate regularization terms. Let us then use for training the loss

$$L(\mathbf{u}) \stackrel{\text{def}}{=} \sum_{\mathbf{x}\in\Omega} \ell(\mathbf{x}, \mathbf{u}(\mathbf{x}), D(\mathbf{u}(\mathbf{x})))$$

where

$$\ell(\mathbf{x}, \mathbf{u}, D) \stackrel{\text{def}}{=} \ell_c(\mathbf{x}, \mathbf{u}) + \lambda_g \ell_g(\mathbf{u}) + \lambda_s \ell_s(D)$$

and where

$$\begin{aligned}
\ell_c(\mathbf{x}, \mathbf{u}) &\stackrel{\text{def}}{=} \psi(\|g(\mathbf{x} + \mathbf{u}) - f(\mathbf{x})\|^2) \\
\ell_g(\mathbf{x}, \mathbf{u}) &\stackrel{\text{def}}{=} \psi\left(\left\|\frac{\partial g(\mathbf{x} + \mathbf{u})}{\partial \mathbf{x}^T} - \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}^T}\right\|^2\right) \\
\ell_s(D) &\stackrel{\text{def}}{=} \psi\left(\|D\|^2\right) \ .
\end{aligned}$$

Recall that $\mathbf{x}$ is image position, $\mathbf{u}$ is the computed motion field, f and g are the two frames, D is the Jacobian matrix of $\mathbf{u}$, and

$$\psi(s^2) \stackrel{\text{def}}{=} \sqrt{s^2 + \epsilon^2}$$

is the Charbonnier loss measure. The nonnegative regularization parameters λ_g, λ_s are either given or chosen by cross-validation.

The loss ℓ just defined is the same used in the variational approach, except for the absence of the term ℓ_μ that measures the mismatch between the motion field $\mathbf{u}$ and the values of motion field measured by a separate method at a sparse set of image locations. We saw in Section 2.1 that neural networks have other means to handle large motions.

The idea of unsupervised image motion estimation [9] is to build a deep neural network ϕ that takes two image frames f and g and an image location $\mathbf{x}$ and computes an estimate $\mathbf{u}(\mathbf{x})$ of the image motion between the two frames at $\mathbf{u}$:

$$\mathbf{u}(\mathbf{x}) = \phi(\mathbf{x}, f, g) \ .$$

The parameters of ϕ are learned by using back-propagation to minimize the average loss $L(\mathbf{u})$ over the training set. The only training set needed for this is a set of frame pairs (f, g), and these are very easy to collect. No annotation is needed.

Three important questions arise at this point:

- What is an appropriate architecture for ϕ?
- Can a differentiable function be found that, given an image g and a motion field $\mathbf{u}$, computes the new image
$$\gamma(\mathbf{x}) \stackrel{\mathsf{def}}{=} g(\mathbf{x} + \mathbf{u}(\mathbf{x})) \ ? \tag{8}$$
This function must be differentiable because it is part of a deep network, which needs to be trained by back-propagation.
- Does the loss ℓ constrain the solution well enough, so that the network ϕ trained in this unsupervised way yields good estimates of the motion field $\mathbf{u}$?

The last question above will be answered empirically: Train a network on a training set (without annotations), and test it on a test set (for which annotations are of course necessary, in order to measure performance).

The answer to the first question can be found in Section 2.1: FlowNet was proven adequate to compute a motion field from a pair of video frames. In computing the motion field values $\mathbf{u}(\mathbf{x})$, FlowNet acts as a *localization network*, in that it computes the locations where g needs to be sampled to produce γ. The computation of the grid points points

$$\mathbf{w} \stackrel{\mathsf{def}}{=} \mathbf{x} + \mathbf{u}(\mathbf{x}) \ , \tag{9}$$

whose coordinates are real-valued, is called *grid generation*.[7]

The answer to the second question above is rather straightforward: We saw earlier in the course that bilinear interpolation can be used to *sample* an image g at a grid of points $\mathbf{w}$. The transformation that computes γ from g in the definition 8 is therefore a simple application of bilinear interpolation, and its implementation for every $\mathbf{x} \in \Omega$ is often called a *sampling network*. To recall, if $\mathbf{w} = (w_1, w_2)$ and

$$\omega_1 = \lfloor w_1 \rfloor \quad \text{and} \quad \omega_2 = \lfloor w_2 \rfloor \tag{10}$$

[7]It seems excessive to assign two different names to computing $\mathbf{u}$ and then $\mathbf{w}$. This is done in order to establish a correspondence with the components of so-called Spatial Transformer Networks [4], which are sometimes more complex than they are here.

are the integer parts of w_1 and w_2, and

$$\delta_1 = \max(0, w_1 - \omega_1) \quad \text{and} \quad \delta_2 = \max(0, w_2 - \omega_2)$$

are their fractional parts, then

$$\begin{aligned} \gamma(\mathbf{x}) = g(\mathbf{w}) &= g(\omega_1, \omega_2)(1-\delta_1)(1-\delta_2) \\ &+ g(\omega_1+1, \omega_2)\delta_1(1-\delta_2) \\ &+ g(\omega_1, \omega_2+1)(1-\delta_1)\delta_2 \\ &+ g(\omega_1+1, \omega_2+1)\delta_1\delta_2 \end{aligned}$$

where $\gamma(\mathbf{x})$ is allowed to be real-valued as well. All the functions in this definition are differentiable, except for the floor functions in equations 10, which are discontinuous. However, the functions δ_i for $i = 1, 2$ are continuous and piecewise linear in w_i with discontinuous derivative at $w_i = \omega_i$. Each of these functions is an affine function of w_i followed by a ReLU function. As a consequence, the discontinuous derivative can be handled with the same techniques used to handle the ReLU in back-propagation.[8]

Spatial Transformer Networks Thus, to compute $g(\mathbf{x}+\mathbf{u}(\mathbf{x}))$ one needs a localization network (FlowNet, to compute the motion field $\mathbf{u}$), followed by a grid generator (to compute the grid of sampling points $\mathbf{w}$), followed by a sampling network (bilinear interpolation applied to every point of the sampling grid to compute γ). This cascade is called a *Spatial Transformer Network* (STN). STNs were initially introduced in the context of image recognition [4], in order to make object representations invariant to certain geometric transformations, but as we see they have found use in image motion estimation as well.

A key property of STNs is that they are made of sub-differentiable functions[9], and can therefore be trained by back-propagation, either by themselves or as part of a bigger network. Estimating image motion may be viewed as the problem of learning the parameters of the localization network of an STN, which is in turn one component of a motion field computation network. The architecture of the full network, called a *Dense Spatial Transform Flow* (DSTFlow) network [9] is shown in Figure 4.

Performance and Research Questions The endpoint errors for the motion-field estimates produced by a DSTFlow network were shown to be about twice what they are for FlowNet [9] when trained on the same set of video frame pairs. However, training a DSTFlow network requires no data annotation, while training a FlowNet needs a full ground-truth motion field for each training sample!

Thus, the notion of unsupervised training of image-motion estimation networks seems to be rather appealing. Since training data is much easier to gather for DSTFlow, it is worth experimenting to see if training such a network on a much bigger dataset produces significantly better results. This has not been attempted at the time of this writing. Working with data sets that are several orders of magnitude bigger than the current ones may still require carefully curated video data, to make sure that frame pairs used for training are related by simple image motion,

[8] Specifically, one computes the *sub-gradient* rather than the gradient. This is a minor technical point, and is beyond the scope of these notes.

[9] Functions with a sub-gradient everywhere.

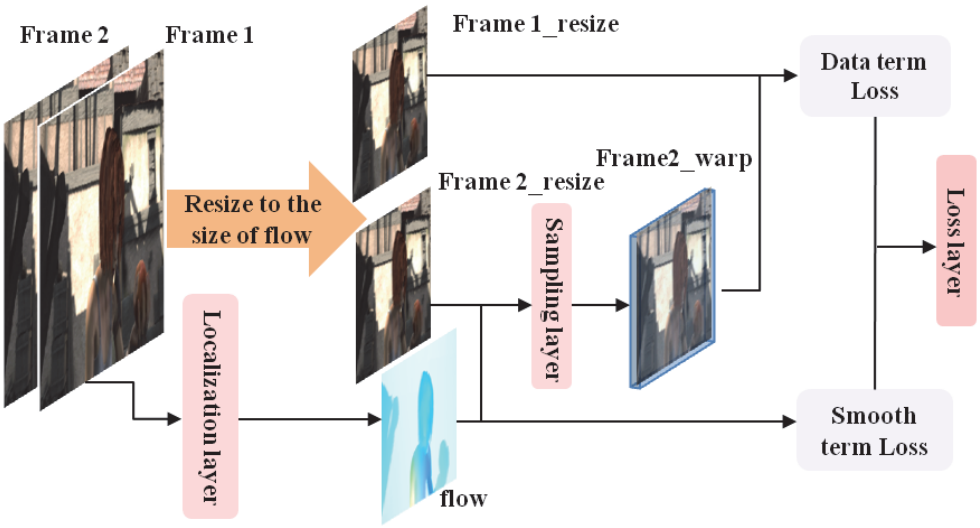

Figure 4: The architecture of a DSTFlow network. The localization "layer" is a FlowNet. The Data term Loss is $\ell_c(\mathbf{x}, \mathbf{u}) + \lambda_g \ell_g(\mathbf{u})$ and the Smooth term Loss is $\lambda_s \ell_s(D)$. All the parameters of the DSTFlow network are learned end-to-end by back-propagation. There are no learnable parameters in the sampling layer. Figure is from [9].

as opposed to scene transitions, dramatic lighting changes, or other un-modeled changes. It is also possible that improving performance enough to match that of supervised methods may call for new training methods that are more tolerant of the more indirect supervisory signal provided by the reprojection error, when compared with the more direct endpoint error that can be computed from ground-truth motion fields.

References

[1] T. Brox and J. Malik. Large displacement optical flow. In *IEEE International Conference on Computer Vision and Pattern Recognition*, pages 41–48, 2009.

[2] D. J. Butler, J. Wulff, G. B. Stanley, and M. J. Black. A naturalistic open source movie for optical flow evaluation. In *European Conference on Computer Vision*, Part IV, LNCS 7577, pages 611–625. Springer-Verlag, 2012.

[3] A. Dosovitskiy, P. Fischer, E. Ilg, P. Hausser, C. Hazirbas, V. Golkov, P. van der Smagt, D. Cremers, and T. Brox. FlowNet: Learning optical flow with convolutional networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2758–2766, 2015.

[4] M. Jaderberg, K. Simonyan, and A. Zisserman. Spatial transformer networks. In *Advances in Neural Information Processing Systems*, pages 2017–2025, 2015.

[5] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[6] P. Liu, M. Lyu, I. King, and J. Xu. Selflow: Self-supervised learning of optical flow. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4571–4580, 2019.

[7] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey. *arXiv preprint arXiv:2001.05566*, 2020.

[8] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation. In *Proceedings of the IEEE international conference on computer vision*, pages 1520–1528, 2015.

[9] Z. Ren, J. Yan, B. Ni, B. Liu, X. Yang, and H. Zha. Unsupervised deep learning for optical flow estimation. In *AAAI*, pages 1495–1501, 2017.

[10] J. Revaud, P. Weinzaepfel, Z. Harchaoui, and C. Schmid. EpicFlow: Edge-preserving interpolation of correspondences for optical flow. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1164–1172, 2015.

[11] Z. Teed and J. Deng. RAFT: Recurrent all-pairs field transforms for optical flow. In *European Conference on Computer Vision*, pages 402–419. Springer, 2020.

[12] P. Weinzaepfel, J. Revaud, Z. Harchaoui, and C. Schmid. DeepFlow: Large displacement optical flow with deep matching. In *IEEE International Conference onComputer Vision*, pages 1385–1392, 2013.