CHAPTER 19

Mapping Images to Image-Like Things

The encoder-decoder architecture of the previous chapters denoised images rather well, and deblurred them acceptably. The master recipe looks like: obtain a large number of pairs of (input image, output image); train an encoder-decoder pair to accept those inputs and produce those outputs; now apply to other inputs. This recipe can solve many other very important application problems if you change what the decoder produces. Images can be mapped to image-like things, described somewhat more crisply below. For simplicity, I will refer to this recipe as *image to image mapping*, or as a regression method.

The recipe is straightforward, and best illustrated with the really important example of transforming images into depth maps. A depth map gives a representation of the distance at each pixel from the camera to the object along the corresponding ray. One way to collect such a depth map is to use a specialized camera system (details in Section 2.2). Now collect a large number of example pairs of image-depth map. Rather than train the decoder to produce a denoised image from the encoding, train it to produce a depth map instead.

Procedure: 19.1 *Image to image mapping: Master recipe*

Map an image to an image-like thing by training an encoder-decoder pair to accept images and produce image-like things.

One caution: this chapter is written in terms of encoders and decoders, quite deliberately. This is because some readers may not have read Chapter 22. Many of the encoders that are used to get the very best results in the applications below are rather different in form than those of Chapters **??**. Another caution: For some cases of image to image mapping, details of the camera used to obtain an image can have significant effects. Understanding these effects requires some background in camera geometry, and is put off to Section **??**.

19.1 MECHANISMS AND CONSIDERATIONS

I use the term "image-like things" to refer to maps that are typically aligned to an image. Examples include depth maps, normal maps and albedos. Mapping an image to a cartoon version of the image is within scope, too. A wide range of really useful predictors can be trained with minor variants of the master recipe. Some conditions are required for the recipe to work. What is predicted should be "like" an image – a spatial map of features, though it does not have to be the same size as the input image.

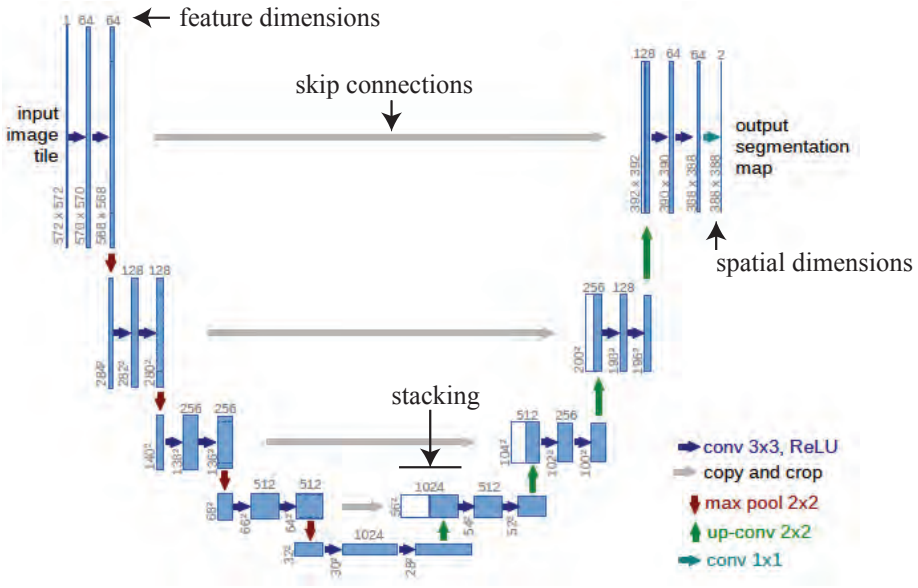


FIGURE 19.1: *The original u-net, reproduced from the original paper. The blue blocks represent feature maps; the heavy arrows are annotated in the legend. "Up-conv 2x2" means upsampling by 2 followed by a 2x2 convolution.* Figure 1 of Ronneberger O, Fischer P, Brox T (2015). "U-Net: Convolutional Networks for Biomedical Image Segmentation", MICCAI 2015

What is predicted can have many feature dimensions. For example, predicting surface normals is successful, and a surface normal field is a map of three dimensional unit vectors. What is predicted needs to be continuous – the methods described here will not do well at predicting, say, an integer at each pixel.

Perhaps the biggest difficulty is that what is predicted should be largely determined once you have an image. This test is often very hard to apply without trying to predict, but some examples will help. One class of problem that doesn't meet the test occurs where there are many different right outputs for a given input. A good example of a problem like this is *colorization*, where one must take a monochrome image and predict a color image. Many different color images correspond to a given monochrome image. The techniques given here tend to work poorly on most colorization problems.

The harder aspect of the test is knowing what can be predicted from an image. It is now known that depth can be predicted very well from a single image in most cases, but up until quite recently it was believed that images had very few little information about depth and were fundamentally ambiguous. Similarly, it is now known that normals can be predicted very well from images as well, but this came as something of a surprise to the community.

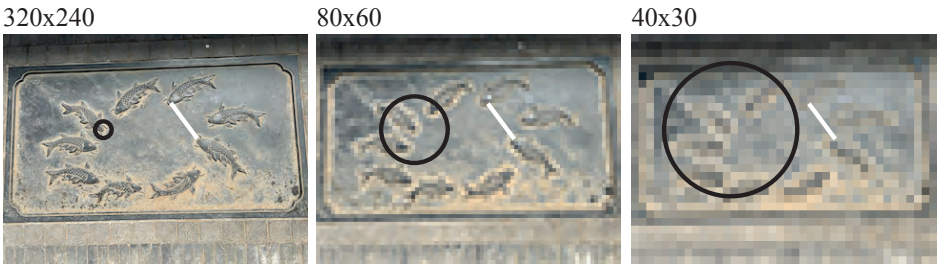


FIGURE 19.2: *A u-net will accept images of the same scene at different resolutions, but the predictions might change because any particular location in the output has a receptive field whose size is measured in pixels. On the* **left**, *an image at 320x240 resolution; in the* **center** *the same image at 80x60 resolution; and* **right**, *that image at 40x30 resolution. The distance between two fish heads in the image is marked by the* **light line**. *The* **dark circle** *represents the receptive field of a location in the output, which is measured in pixels. As a result, the receptive field is rather smaller than the distance between fish heads in the left image, and rather larger in the right image. Ideally, the reported value at this pixel is the same (or about the same) for each image, even though the image values that the function that computes the output value observes very different inputs in these three cases.*

19.1.1 U-Nets

A *u-net* is an encoder-decoder combination, so-called because the encoder makes feature blocks spatially smaller and larger in the feature dimension, then the decoder makes feature blocks spatially larger but smaller in the feature dimension. If you draw this structure in the right way, it looks like a U. The original u-net had a specific architecture (Figure **??**), but the term now encompasses a broad range of networks. Networks that consist of image encoders followed by decoders that produce image-like things are now often called u-nets.

Notice that the u-net of Figure **??** is *fully convolutional*. This means that each layer is agnostic as to the spatial extent of its input, as long as the input is big enough. For example, a 3×3 convolutional layer will accept any input larger than 3×3 and a ReLU layer or an upsampling layer will accept any size of input. A fully connected layer will not. Since each layer has this property, so does the whole network. This is a common property of u-nets.

A network that is fully convolutional has some important advantages for image to image mapping. As long as the input image is big enough in each dimension, the network will produce an output. The minimum size depends on the network (you need to ensure that the smallest feature block is big enough that the convolution will succeed **exercises**).

19.1.2 U-nets, Scale and Resolution

If a u-net is fully convolutional, you can train the network on images of a fixed size, and expect it to produce outputs for images of somewhat different sizes. What is

remarkable is that this output is usually quite close to what you would want. For example, you could train a u-net to produce 320x240 depth maps from 320x240 images. If you then pass this u-net an image at 640x480 resolution, the output is likely quite close to the 640x480 depth map. This form of generalization is quite useful.

Just how far this generalization extends depends in some detail on the application. Think about the responses in the network to two images of the same scene of an interior. One image is $N \times N$ and the other is $kN \times kN$. The actual distance between two items – for example, two chairs which are two metres apart – in the scene is not affected by the change in resolution. However, the number of pixels between these chairs has changed by a factor of k. When the network observes the first image, the receptive field for any feature is very different than when it observes the second image (Figure 19.2). As k gets bigger, the effect is more significant. You should expect the network to be somewhat resistant to problems caused by the change in receptive field. In the example, the training set should be big enough to contain pictures with two chairs two metres apart seen from fairly far away (so they appear close in the image) and seen from fairly nearby (so they appear well separated in the image). Experience teaches that u-net based image to image mappers behave rather well in the presence of this effect unless the change of image scale is very large.

19.1.3 Equivariance

Many image to image maps should have properties that are referred to as *equivariance*. Annoyingly, this term is almost always misused. It applies to the behavior of a function under a group action (there is no particular reason to look this up if you don't know what it is), so one should specify what kind of equivariance is intended. Properly, translation equivariance is the shift-invariance of Section 6.

As a thought experiment, think about an image to image mapping that accepts an input $\mathcal{I}(x, y)$ and produces an output $\mathcal{O}(x, y) = \mathcal{M}(\mathcal{I})$. Assume that each is defined on the entire plane. Then translation equivariance of the mapping requires that if you translate the input, the output translates. In turn, this means that for any t_x, t_y, if

$$\mathcal{I}_2(x, y) = \mathcal{I}_1(x + t_x, y + t_y) \text{ and } \mathcal{O}_1 = \mathcal{M}(\mathcal{I}_1)$$

then

$$\mathcal{M}(\mathcal{I}_2) = \mathcal{O}_1(x + t_x, y + t_y).$$

Notice that with very little work, you can extend this definition to apply to functions of an infinite grid on the plane and discrete translations (I've actually already done this; look up shift invariance in Chapter 5). This property is entirely desirable for an image to image mapping, yet almost always unattainable because you never have an infinite image. If you happened to be in a position to apply a u-net to infinite images $\mathcal{I}_1$ and $\mathcal{I}_2$, you would discover (after a very long wait) that it was translation equivariant. But if you apply a u-net to a finite $\mathcal{I}_1$, you will discover that when you translate the image to get $\mathcal{I}_2$ some pixels of $\mathcal{I}_1$ disappear and new pixels appear. This means the translation isn't a group action and the u-net isn't translation equivariant in any useful sense (there is a method to fix this mathematical difficulty

– you identify the top of the image with the bottom, and the left with the right, to get a torus, and translation acts on this torus – but it doesn't apply to real images).

Another way to think about the nuisances created by working with finite images is that your u-net is pretty much guaranteed to use padding at most stages. If it doesn't, the output will be a lot smaller than the input. But padding means the function that predicts a value in the center of the output is likely quite different from the function that predicts a value close to its boundary. The predictor in the center will mostly depend on image pixel values. The predictor near the edge will depend quite strongly on the padding. Because they're not the same, the overall prediction can't be equivariant.

A somewhat weaker form of translation equivariance is highly desirable for most image to image mappings. Imagine that $\mathcal{I}_1$ and $\mathcal{I}_2$ are finite windows cut out of an infinite image and they overlap. Then it is desirable that $\mathcal{O}_1$ and $\mathcal{O}_2$ agree with one another in the region of overlap. This property does not appear to emerge naturally just as a result of having a large training set **exercises**.

Section 19.1.2 argued that passing the same image at different resolutions to a well-trained u-net mostly produces the same result at different resolutions. While this can't be true for very coarse or very fine resolutions, there is usually a range of resolutions for which it is largely true. You should regard this property as being a weaker form of *scale equivariance*.

Remember this: *Many very useful image-to-image mappings can be implemented with an encoder-decoder pair, usually called a u-net. A fully convolutional u-net will produce a useful output for any input above a minimum size. Not all u-nets are fully convolutional. This depends on how the encoder is built. U-nets are not – and cannot be made – formally translation equivariant. In practice, u-nets display good behavior under scale and translation.*

19.2 APPLICATION: DEPTH FROM A SINGLE IMAGE

The master recipe applies to predicting depth from a single image in a straightforward way. Obtain a large number of (image, depth map) pairs, then train an encoder-decoder pair (perhaps a u-net) to predict the depth from the image. Mostly, the prime obstacle is obtaining enough data, and depth predictors have improved notably as the size of the available datasets has increased. Improvements in encoder structure have also been important (Chapter 22). You should regard single image depth prediction as a technology that "just work" for most purposes.

19.2.1 Representing Depth

What comes out of a depth predictor is very seldom the depth, and for very good reasons. The actual distance in, say, metres (otherwise known as *absolute depth*) can be difficult to predict and is often not particularly useful. To predict actual

depth, you need to know, at least, something about the scale in the camera and the scale in the world. Here is an example. Imagine your dataset contains both images of real rooms and images of dollhouse rooms. The dollhouse furniture is very realistic. The dollhouse pictures are taken with a camera that has an arrangement of lens, imaging device, etc. so that the size of the furniture in dollhouse images and the size of furniture in real world images is the same. Under these circumstances, getting absolute depth might be very difficult indeed. A variety of possibilities will break this ambiguity. For example, there may be no dollhouse pictures; you may know something about the camera setup; and so on. But you will need to know something substantial about the data you are working with to recover absolute depth from pictures reliably. An alternative is *relative depth*, which is depth up to a scale that is unknown and can vary from image to image. Here a dollhouse picture and a real picture might report the same relative depth map. Something else would have to determine the relationship between the maps and the actual depth.

Both absolute and relative depth present another problem. Large distances tend to be less important than small distances. You will bump into nearby objects sooner than you will bump into distant objects, for example. This means that it is usually acceptable to make larger errors in bigger depths (relative or absolute). But a 1% error in a large depth is much larger than a 1% error in a small depth. If you train a model using the squared error in depth, the behavior of the trained model tends to be dictated by the error in large depths. The *inverse depth*, sometimes known as *disparity* (Section 36.3) has a convenient property. Assume the true depth is d, so true disparity is $\delta = 1/d$. If the predictor predicts disparity with some error ϵ, then the corresponding depth is

$$\frac{1}{\delta + \epsilon} \approx d - \epsilon d^2$$

which means that a fixed error in disparity is a bigger error in large depth and a smaller error in small depth.

Depth predictors mostly predict

$$a_d \delta + b_d$$

where a_d and b_d are constants, determined *per image*. At training time, these constants are recovered from the training depth map. Testing a system like this is easy if you have a depth map for the test image (recover the constants, recover the depth from the disparity, then compare). But when you use the system, you may not have a depth map, and you need some procedure to manage the fact that two constants are missing per image. If you use a model from the internet, you should be careful to check what it predicts. Some models estimate these constants then correct the disparity estimate into an absolute depth estimate using these estimates; others will estimate only b_d and report a relative disparity estimate; others simply report $a_d \delta + b_d$ and leave the rest to you.

19.2.2 Losses and Evaluation

Depth prediction losses tend to be a mixture of L_1 and L_2 norms on the prediction error. I am not aware of evidence that an adversarial loss usefully improves depth

predictors. This is likely because depth maps are relatively simple, and the amount of data available makes adversarial losses superfluous.

Desirable properties of a depth predictor are: (a) accuracy and (b) good zero-shot behavior. One key metric for accuracy of depth predictors is *AbsRel*. Compute the depth prediction from whatever the predictor produces for an image. You might need to use the ground truth depth map for that image to do this. For example, if it predicts $a_d\delta + b_d$, use the depth map to extract a_d and b_d and then recover predicted depth $\hat{z}_{ij}$ for the i, j'th pixel. Write z^*_{ij} for the ground truth depth at each pixel. Now evaluate the mean over all pixels and all images of

$$\frac{|\,\hat{z}_{ij} - z^*_{ij}\,|}{z^*_{ij}}.$$

Another metric, usually called δ, measures the fraction of pixels such that

$$\max\left(\frac{\hat{z}_{ij}}{z^*_{ij}}, \frac{z^*_{ij}}{\hat{z}_{ij}}\right) > 1.25.$$

These metrics measure slightly different things. AbsRel measures how bad relative depth is on average; δ measures what fraction of pixels is really bad. If you don't have ground truth depth for the image, neither metric works. If you have disparity, you can use mean-square error in disparity, scaling and translating as required.

Accuracy in normal predictors is usually evaluated using mean and median of the angle between true and predicted normals, and the percentage of normals where that angle is below some set of thresholds (typically 11.25^o, 22.5^o and 30^o).

Good zero-shot behavior means the predictor can produce accurate depth maps for images of a kind not seen in training. It is difficult to measure how different a test image is from training images with any kind of precision, but the usual procedure is to train on on a training split from some datasets, test on a test split of those datasets, then also evaluate on datasets where no item was used in training.

As of writing, depending on the dataset and on the model, for depth predictors you could expect an AbsRel of between 0.06 and 0.25, a δ of between 1.9% and 25%, and a frame rate of between 5 and 90 FPS. For normal predictors, you could expect to see mean angular errors of 15^o to 30^o, depending on model and dataset. Generally, faster models produce worse predictions, and in each case there is a notable component of error that is explained by test dataset alone. Generally, if a slow model produces good or bad numbers on a particular dataset relative to other datasets, so will a fast model. Depth and normal are intertwined, so it is usual for methods to predict both simultaneously. There is some evidence that doing so produces better predictions.

19.2.3 Resources and Considerations

Training your own depth predictor is now not for the faint-hearted. The collection of available datasets is large; each has its own special properties and nasty habits; and evaluation is at a very large scale. Using a published depth predictor is usually fine, but you do need to keep an eye on what it predicts. Confusing $a_d\delta + b_d$ for depth is easy to do and leads to all sorts of mischief.



FIGURE 19.3: *On the* **left** *an example of* foreshortening. *A circle on a world plane looks like an ellipse in an image plane, because the world plane is tilted backwards. Try this by holding your hand in front of your face, then tilting it backward – it gets shorter in the tilted dimension. If you knew that you were viewing a plane covered in circles, the shape of the texture elements in the image would give you a cue to the the orientation of the plane. Some version of this cue seems to be used by people. On the* **right**, *an image of trees at HKU. Notice how the leaves on each tree seem to give you a cue to the shape of its surface. Elementary methods for exploiting this cue are both intricate and obsolescent, but the cue is powerful.* Image credit: *My photograph of trees at HKU.*

Resource pointers: 19.1 *Some strong open source depth predictors*

These include:

- MiDAS `https://github.com/isl-org/MiDaS`;
- Depth-Anything `https://github.com/LiheYoung/Depth-Anything`;
- Depth-Anything V3 `https://depth-anything-3.github.io`;
- Omnidata `https://github.com/EPFL-VILAB/omnidata`.

These links also connect to information about available datasets. In fact, the Omnidata link leads to a pipeline that builds datasets, as well as predictors for depth and other surface properties.



FIGURE 19.4: *Predicting depth from single images is now a reliable technology.* **Top row** *shows images;* **bottom row**, *predicted depth representations. The predictor reports* $a_d\delta + b_d$ *where* δ *is disparity and* a_d, b_d *are* per image *constants. Orange is closer to the eye, purple is further from the eye. Notice: depth maps contain quite fine detail (eg the larger elephant's trunk; where the rider contacts the horse; the grooves between suitcases); small depth gradients are well represented (eg along the cowboy's arm); windows can cause confusion (eg window next to suitcases).* Image credit: *Figure 1 of "Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-shot Cross-dataset Transfer", Ren´ e Ranftl*, Katrin Lasinger*, David Hafner, Konrad Schindler, and Vladlen Koltun.*

Resource pointers: 19.2 *Some strong datasets used for depth prediction*

These include:

- NYUV2, a dataset of scans of indoor scenes; contains images, depth and segmentation information (`https://cs.nyu.edu/~fergus/datasets/nyu_depth_v2.html`).
- ScanNet, a dataset of scans of indoor scenes; contains images, depth and segmentation information (`http://www.scan-net.org`).
- Kitti, a dataset of outdoor scenes mostly to do with cars and driving, with depth and other information (`https://www.cvlibs.net/datasets/kitti/eval_depth.php?benchmark=depth_prediction`).

Mostly, if you need normals, you need to estimate them from the depth maps.

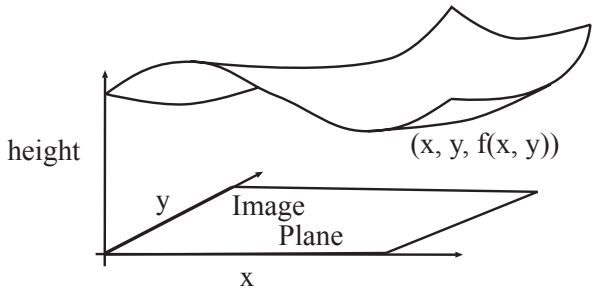


FIGURE 19.5: *Normals can be derived from depth measurements in straightforward ways. At the (x, y) location in the image, the depth is $f(x, y)$, leading to a surface $(x, y, f(x, y))$. Straightforward computations yield the normal of this surface as long as it is differentiable.*

Remember this: *Very accurate depth predictions can be made with little real difficulty at high speed and high accuracy from most images. Predictions can be in a variety of forms, and it is important to be sure what your predictor is producing. Training your own predictor is not for the faint-hearted, but open resources are extremely strong.*

19.3 APPLICATION: NORMAL FROM A SINGLE IMAGE

The master recipe applies to predicting surface normal from a single image, with minor variations. Getting normal data that is accurate at every pixel is mostly impossible, and this needs to be managed. Depth and normal are related, and this can be exploited. Single image normal prediction is another technology that "just works" for most purposes.

19.3.1 Normal from Depth

A *surface normal* or (usually) normal is a unit vector at a point on a surface that is at right angles to the surface. This vector exists if the surface has a *tangent plane* – a plane of tangent vectors. Most surfaces in real life do. If a surface has a sharp crease in it, the tangent plane may be hard to define or meaningless. If a surface wiggles significantly at very small scales – for example, the surface of a natural sponge – it may require care to define a meaningful tangent plane. Mostly, objects in images can be thought of as surfaces which are smooth enough that they have a normal at most points. There is a close relationship between depth and normal, so if you can predict one from an image, you should be able to predict the other.

The relationship is important. Assume that a camera maps the point at



FIGURE 19.6: *An example of an RGB image registered to a depth map. The depth map was obtained with a Kinect device. Bluer points are closer, redder are further away. Notice a scattering of black points, which correspond to locations where the Kinect could not determine depth. These tend to occur at the outlines of objects, but can be elsewhere. They are often a result of shadowing effects or specular reflection at the surface. These effects mean some care is required to get normal estimates, as derivatives may be unreliable.* Image credit: *The images come from the very well known NYUV2 dataset, published at* `https://cs.nyu.edu/~fergus/datasets/nyu_depth_v2.html`.

(x, y, z) in 3D to the point (x, y) in the image (there are other camera models, and everything here can be adapted to them). Further, write $f(x, y)$ for the image depth at the point (x, y) in the camera. Then there is a surface in 3D given by

$$(x, y, f(x, y))$$

illustrated in Figure 19.7 Now consider the vector from $(x, y, f(x, y)$ to $(x+\delta x, y, f(x+\delta x, y)$; as δx gets smaller, this vector will get shorter, but

$$\lim_{\delta x \to 0} \frac{1}{\delta x} (\delta_x, 0, f(x + \delta x, y) - f(x, y))^T = \left(1, 0, \frac{\partial f}{\partial x}\right)^T$$

will be tangent to the surface if the derivative exists. Similarly,

$$\left(0, 1, \frac{\partial f}{\partial y}\right)^T$$

is tangent to the surface. Using the very widely established convention of writing

$$p = \frac{\partial f}{\partial x} \text{ and } q = \frac{\partial f}{\partial y}$$

the normal is

$$\frac{1}{\sqrt{1 + p^2 + q^2}} [-p, -q, 1]^T$$

(although you can flip the sign depending on whether you want the normal pointing from the surface to the camera or from the camera to the surface).



FIGURE 19.7: *Ground truth normals estimated from depth measurements can present difficulties. The* **top row** *shows images from the NYUV2 dataset of Figure 19.7.* **Bottom row** *shows normal estimates derived from the depth, and used as ground truth on occasion. Normal estimates are rendered by mapping R, G and B to components of the normal. Here, a normal pointing to the left in the image is pink, one pointing up in the image is light green, and the normal of a frontal surface is blue. These estimates aren't the same as the true normals (look at the venetian blinds, for example; the blodges on the back wall in the bicycle image or the bedroom image; and the blodges in the bookshelf image). Evidence that the difficulty comes from problem depth predictions includes the tendency of deformations to start at surface boundaries and to be concentrated at shadow effects.* Image credit: *Figure 2 of "Estimating and Exploiting the Aleatoric Uncertainty in Surface Normal Estimation" Gwangbin Bae Ignas Budvytis Roberto Cipolla*

What all this means is that if you have a depth estimate that is good enough that you can form acceptable estimates of the partial derivatives, you can estimate a normal. Estimating the normal from the partial derivatives isn't a particularly good idea, however, because the depth tends to be noisy. It is better to use the approach of Section 14.1.3 (here you would get the nearest neighbors by looking at neighboring image points). Even this approach will produce poor normal estimates at some image locations, because depth measurement devices have problems measuring depth at some locations. These are typically around the outlines of objects in images, where the depth can change very sharply from pixel to pixel (Figure 19.7 and Section 2.2 for more details). Most images in image-depth datasets are affected by these missing points. If you see a depth image that appears not to be, you should suspect that it has been postprocessed in some way.

19.3.2 Depth from Normal

If you have a normal estimate that is good enough you can estimate depth up to a constant of integration. Assume the normal predictor predicts a unit normal at each location $\mathbf{x} = (x, y)^T$. Write $[n_x(\mathbf{x}), n_y(\mathbf{x}), n_z(\mathbf{x})]^T$ for that normal. Now

$$\frac{-n_x(\mathbf{x})}{n_z(\mathbf{x})} = \frac{\partial z(\mathbf{x})}{\partial x} = p(\mathbf{x}) \text{ and } \frac{-n_y(\mathbf{x})}{n_z(\mathbf{x})} = \frac{\partial z(\mathbf{x})}{\partial y} = q(\mathbf{x})$$

so you should be able to recover the depth map up to a missing constant by integration. This requires some care, because the estimates of the normal might not be exactly right. Recall that

$$\frac{\partial p}{\partial y} = \frac{\partial q}{\partial x}$$

a condition often referred to as *integrability*. An algorithm appears below. Better estimates can be constructed, but require more elaborate methods **exercises**.

Procedure: 19.2 *Obtaining depth from normals*

Represent the depth map as a vector $\mathbf{z}$. Write $\mathbf{n_x}$ for a vector containing the x component of each normal (etc.). Write $\mathcal{D}_x$ for a matrix that maps the depth map to its x derivative. It is likely that smoothing the depth before differentiation is a good idea; incorporate this into $\mathcal{D}_x$ **exercises**. Write diag for the operator that maps a vector to a diagonal matrix with that vector on the diagonal. Then you can recover a depth estimate by solving

$$\underset{\mathbf{u}}{\text{argmin}} \left\{ \begin{array}{c} [\text{diag}(\mathbf{n_z})\mathcal{D}_x\mathbf{u} + \mathbf{n_x}]^T [\text{diag}(\mathbf{n_z})\mathcal{D}_x\mathbf{u} + \mathbf{n_x}] + \\ [\text{diag}(\mathbf{n_z})\mathcal{D}_y\mathbf{u} + \mathbf{n_y}]^T [\text{diag}(\mathbf{n_z})\mathcal{D}_y\mathbf{u} + \mathbf{n_y}] \end{array} \right\}$$

subject to a constraint on $\mathbf{u}$ that supplies the constant of integration (for example, $\mathbf{1}^T\mathbf{u} = 0$)

What comes out of a normal predictor is usually not the normal. Write $(n_x, n_y, n_z)^T$ for the components of the unit normal. Most normal predictors report $(1/2)(n_x + 1, n_y + 1, n_z + 1)^T$, which has the advantage of being in the range $[0, 1]$. If you don't know this, using the normal predictor can be difficult.

Normal predictors require care, because the ground truth predicted from depth images can present problems (Figure 19.7). Pushing the normal predictor to reduce prediction error at locations where the ground truth isn't reliable can cause training problems. However, there are manageable procedures to estimate where the ground truth is right (Figure 19.8). Predictions can be notably improved by incorporating such an estimate. Write κ_i for an estimate of predicted normal uncertainty at pixel i, where κ_i is larger when the predictor is more certain. Write μ_i for the predicted value of the normal at i and $\mathbf{n}_i$ for the ground truth normal at i. Then

$$\mathcal{L}(\kappa_i, \mu_i, \mathbf{n}_i) = -\log\left(1 + \kappa_i^2\right) + \log\left(1 + \exp\left(-\kappa_i \pi\right)\right) + \kappa_i \text{acos} \mu_i \mathbf{n}_i$$

is a loss that encourages the network to reveal when the prediction is uncertain but also encourages the prediction to be like the ground truth.

This loss is worth understanding in some detail. Think about the unet looking at some local region of an image to predict a normal. The unet must be computing some feature representing the region, then predicting a normal from that feature. Regions that look similar will tend to have similar features and will tend to get

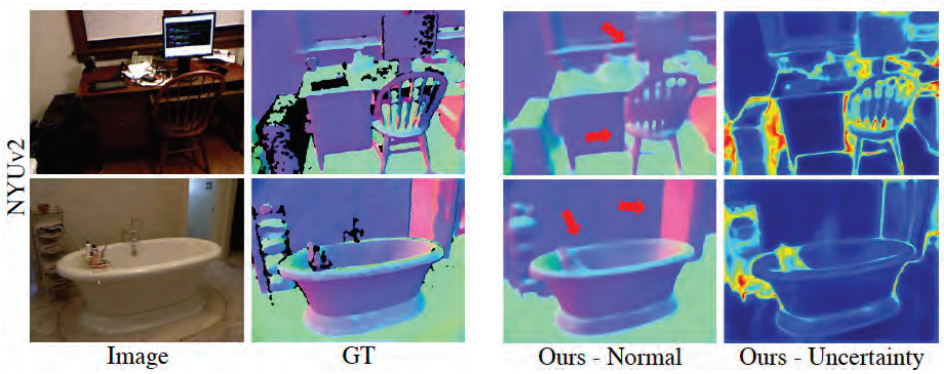


FIGURE 19.8: *If uncertainty in estimating the ground truth is taken into account, single image normal predictions can be very strong.* **Left** *shows image and ground truth normal estimates; note the points where normal can't be estimated (black).* **Right** *shows estimated normals and estimates of uncertainty (yellow/red colors are more uncertain; red arrows point to fine details recovered particularly well).* Image credit: *Part of Figure 2 of "Estimating and Exploiting the Aleatoric Uncertainty in Surface Normal Estimation" Gwangbin Bae Ignas Budvytis Roberto Cipolla*

similar normal predictions. If all the training data patches that look similar have a very similar normal, then the unet can obtain a small loss by predicting that normal and a large κ value. The larger the κ, the smaller the first two terms in the loss. A large κ is only practical if all the training patches that look similar to that region have a very similar normal. Otherwise, the $\text{acos}\mu_i\mathbf{n}_i$ term will be large because at different training items $\mathbf{n}_i$ will be different. Similarly, if the training data items that are "similar" to that region have quite different normals, it is safer to predict a small κ.

19.3.3 Why it Works

It isn't obvious why one should be able to predict depth or normal from a single image. For most of the history of computer vision, it was believed that images were intrinsically ambiguous because they had been projected from 3D to 2D. As a result, the image did not reveal (a) how far away surfaces were and (b) what their orientation was. Current evidence strongly supports the idea that depth and normal are relatively easily predicted from a single image – most images are not, in fact, all that ambiguous. This likely follows from a number of interrelated points.

- The depth and normal at a pixel is usually rather similar to the depth and normal at neighboring pixels. Further, large changes in depth and normal are usually signalled by edges.
- Depth maps aren't all that complicated. Scenes tend to have fairly stylized depth maps, so, for example, rooms are boxes with other boxes in them, and so on. Further, many depth regions are roughly either inclined planes, cylinders or spheres.

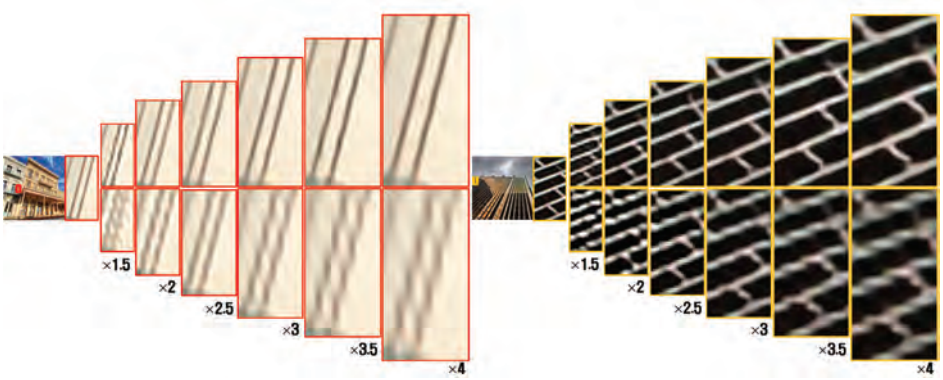


FIGURE 19.9: *Learned superresolution methods can perform extremely well. This figure shows the effect of superresolution methods by zooming in on patches from two example images (tiny rectangles in* **red** *and* **yellow** *superimposed on the image). Two methods are shown. The method producing the patches on the* **top** *row has been trained to produce a residual estimate for a wide range of different upsampling amounts. The method producing the patches on the* **bottom** *row has been trained to produce a residual estimate for* $2\times$ *upsampling. Methods are impressively succesful, but training with a range of upsampling amounts makes a real difference to the results.* Image credit: *Figure 5 of "Accurate Image Super-Resolution Using Very Deep Convolutional Networks", Kim et al, 2015. The patches in the top row come from that method; the patches in the bottom row come from the method of "Learning a deep convolutional network for image super-resolution", Dong et al, 2014.*

- Surface textures and shading offer cues to the orientation of a surface and so to the normal. Shading cues are discussed in Section 36.3. Texture cues are illustrated in Figure 19.3. The early vision literature is rife with elementary methods to exploit these cues, which weren't all that successful.

Remember this: *Very accurate depth and normal predictions can be made with little real difficulty at high speed and high accuracy from most images. Predictions can be in a variety of forms, and it is important to be sure what your predictor is producing. Training your own predictor is not for the faint-hearted, but open resources are extremely strong.*

19.4 FURTHER APPLICATIONS

19.4.1 Image Superresolution

Image superresolution is the task of taking an image at some resolution and producing a higher resolution version of the image. Ideally, the method does not care

how much bigger you make the image (though there is likely some performance limit). The natural baseline method is simply upsampling and interpolating (recall Figure 3.2). A version of the master recipe applies as well. It is straightforward to find pairs that look like (image at low resolution, image at high resolution). There is no need to predict much of the information in the high resolution image, because elementary interpolation methods give a reasonable estimate. Careful framing of the problem makes it possible to train a method that is agnostic about the degree of upsampling.

Write $\mathcal{L}$ for the low resolution image, interp($\mathcal{L}$) for an interpolation method that upsamples the low resolution image by the same amount in each direction, and $\mathcal{H}$ for the target high resolution image. Now train a predictor – which will be a u-net – that accepts interp($\mathcal{L}$) and produces an estimate $\hat{\mathcal{R}}$ of the residual $\mathcal{H} - \text{interp}(\mathcal{L})$. The full high resolution image is then

$$\hat{\mathcal{R}} + \text{interp}(\mathcal{L}).$$

$\hat{\mathcal{R}}$ should consist of mostly zeros or very small values, because interpolation is quite good. Notice that this means that as long as the u-net can predict acceptably, the method does not need to know *by how much* you are upsampling. Instead, it takes your interpolated upsampling and predicts small corrections to that. In turn, this suggests the method should be trained using data that is upsampled by different amounts.

Evaluation is by PSNR on standard datasets. Competitive methods will get PSNRs in the high 20s or early 30s for $4\times$ upsampling, depending on the evaluation dataset.

Resource pointers: 19.3 *Resources for superresolution*

There is now a large range of upsampling methods and a large range of evaluation datasets:

- `https://github.com/ChaofWang/Awesome-Super-Resolution` offers a list of papers with links to code.
- A useful review appears in [].

19.4.2 Defogging

Fog or haze has significant effects on images. These are relatively easily explained with a simple physical model (Figure 19.11). In fog, light does not travel directly from a source to a surface to the eye. Instead, some fraction of light travelling along a given ray is *scattered* out of the direction of that ray (perhaps it hits a tiny drop of water suspended in the air, or some dust). Further, some amount of light is scattered into the direction of the ray, because it was scattered out of some other ray. In fog, scattering effects are largely independent of angle or of wavelength, so what the image looks like depends the expected distance that light will travel before being scattered. If the fog is heavy, this distance is short, and you may not

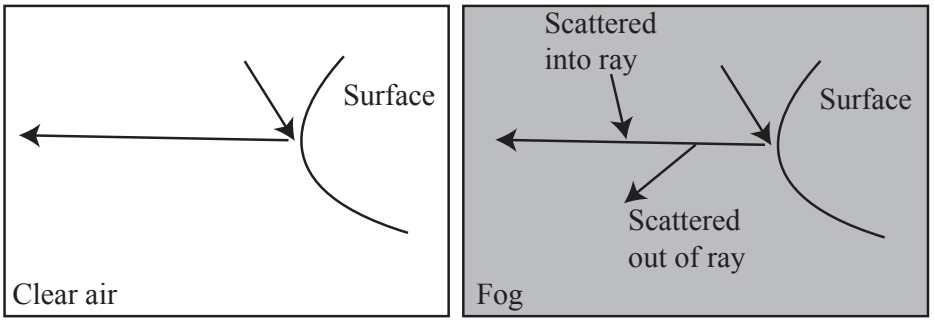


FIGURE 19.10: *In clear air, light travels along the ray from a surface toward the sensor directly, and without any effects (***left***). In fog or haze, light leaving the surface along the ray to the sensor can be scattered out of that ray (perhaps by encountering a fine water droplet), as in the* **right**. *Some of the light arriving at the sensor will consist of light that was travelling in another direction, but was scattered into the ray pointing at the sensor. If the surface is very far from the sensor, almost all light will come from the fog and so the pixel will be fog colored. If the surface is close to the sensor, it will look as if there was no fog, but a more distant surface will have lower contrast and its color will shift towards the color of fog*

be able to resolve much in the image because almost all light arriving at a pixel is light scattered into the viewing direction. This will tend to be the same from pixel to pixel.

The light scattered into a ray will depend on the length of the ray. The longer the ray, the more likely light scatters into that ray. If the fog is very light, you may see no effect at all except over very long scales (of the order of kilometers). The color of very distant objects will shift slightly towards the color of the light source, because the inscattered light will tend to come from the light source. If you know the color of the light source or can estimate it, you can use this effect as a depth cue (Figure 36.3; the cue is known to artists as *aerial perspective*). When the fog is light, objects that are near in the image will be largely unaffected but more distant objects may be quite strongly obscured. A simple physical model of this effect is straightforward. Write $p_i(\mathbf{x})$ for the value of the i'th color channel at location $\mathbf{x}$, $c_i(\mathbf{x})$ for the color you would see at location $\mathbf{x}$ if there was no fog, l_i for the color of the light source in the i'th color channel, s_i for a scattering constant (one for each color channel) and $d(\mathbf{x})$ for the depth at $\mathbf{x}$. Then

$$p_i(\mathbf{x}) = e^{[-s_i d(\mathbf{x})]} c_i(\mathbf{x}) + (1 - e^{[-s_i d(\mathbf{x})]}) l_i$$

is a fair physical model. This model omits a blurring effect, which is easily observed. Distant objects will tend to be somewhat blurred as well, and nearby objects will tend to be sharper (Figure 36.3). There are a number of elementary defogging methods that exploit this model or variants quite successfully.

Defogging by image to image transformations follows the master recipe in a natural way, and is notably successful. One minor nuisance is that it is hard to

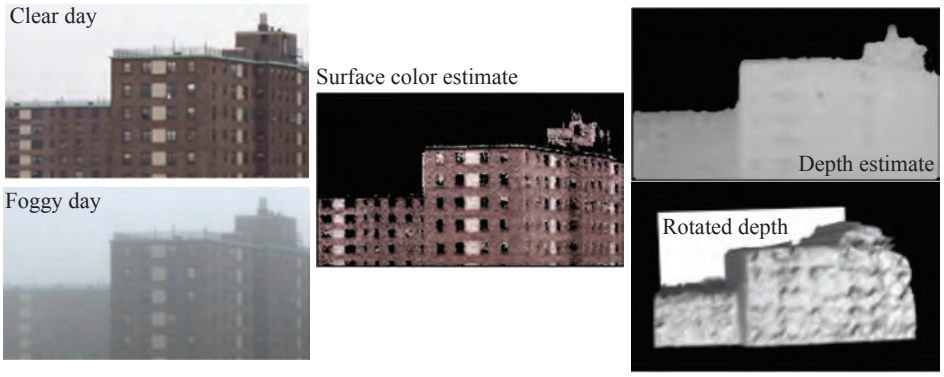


FIGURE 19.11: *Fog offers a depth cue if you know the color of surfaces in the absence of fog or if you see them in two different levels of fog. On the* **left**, *views of a set of distant surfaces on a foggy and a (fairly) clear day. This is enough to reconstruct the color of surfaces (***center***). This color image, together with the foggy image, yields the depth on the* **right**. Image credit *Parts of Figure 9 of "Vision in bad weather", Nayar and Narasimhan, 1999.*

obtain pairs of (image of scene, image of foggy scene). A fog machine can help with this problem, but isn't a scalable approach. It is usual and successful to simulate foggy images from non-foggy images by applying the physical model above.

Evaluation is by PSNR, but mostly on simulated fog.

Resource pointers: 19.4 *Resources for defogging*

There are lists of defogging and dehazing papers at:

- `https://github.com/younggguncho/awesome-dehazing?tab=readme-ov-file`.
- `https://github.com/Xiaofeng-life/AwesomeDehazing`

There are datasets of pairs (image of scene, image of foggy scene) available at:

- `https://data.vision.ee.ethz.ch/cvl/ntire19/ /dense-haze/`
- `https://data.mendeley.com/datasets/jjpcj7fy6t/1`
- `https://github.com/tanvirnwu/HazeSpace2M_ACMMN_2024?tab=readme-ov-file`
- `https://github.com/fiwy0527/AAAI25_SGDN`

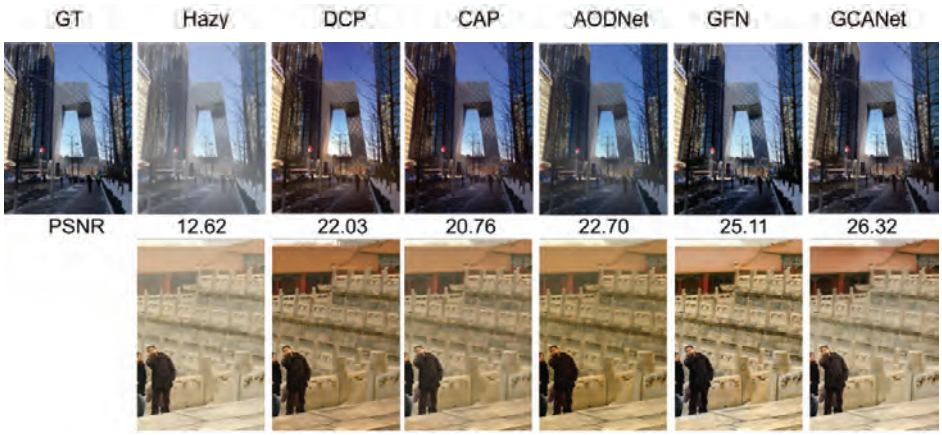


FIGURE 19.12: *Image to image methods defog images satisfactorily. The* **top** *row shows results for a number of methods applied a simulated foggy image. The original image is on the* **far left***; the simulated foggy image appears* **next***. Results from a number of different methods follow. Below each is the reconstruction PSNR, which can be computed because there is both a ground truth and an estimated image. The* **bottom** *row shows results for a real foggy image. Here PSNR is not available, because the true defogged version is not known.* Image credit: *Part of Figure 3 of "Gated Context Aggregation Network for Image Dehazing and Deraining" Chen et al 2018. The different methods are cited in detail in that paper.*

Remember this: *There are many applications for the master recipe for image-to-image mapping. Image superresolution produces an upsampled version of an image that is better than pure interpolation using a learned procedure, and works well over a large range of upsampling scales. Defogging removes fog from images, and typically requires simulated training data, but works well.*

19.5 YOU SHOULD

19.5.1 remember these facts:

U-nets are well-behaved methods for image-to-image mapping. . . . 364
Accurate, fast depth predictions can be made at high speed from most images. . . . 369
Accurate, fast depth and normal predictions can be made at high speed from most images. . . . 374
Superresolution and defogging are useful applications of image-to-image mapping. . . . 378

19.5.2 remember these procedures:

Image to image mapping: Master recipe . . . 360
Obtaining depth from normals . . . 372

19.5.3 exploit these resources:

Some strong open source depth predictors . . . 367
Some strong datasets used for depth prediction . . . 368
Resources for superresolution . . . 375
Resources for defogging . . . 377

19.5.4 be able to:

- Make an informed guess as to whether a particular image-to-image application will work.
- Explain the master recipe.
- Use a single image depth predictor.
- Use a single image normal predictor.
- Use an image superresolution method.
- Use an image defogging method.
- Spot other likely applications of image-to-image mapping.

EXERCISES

QUICK CHECKS

19.1. Section 19.1.1 says: "The minimum size depends on the network (you need to ensure that the smallest feature block is big enough that the convolution will succeed)." How do you determine the minimum size from the network?

19.2. A surface is given by $(x, y, f(x, y))$. Show that

$$\mathbf{n} = \frac{1}{\sqrt{1 + p^2 + q^2}} [-p, -q, 1]^T$$

is (a) normal to the surface and (b) a unit vector. Is $-\mathbf{n}$ also a unit normal?

19.3. Could you represent a sphere by $(x, y, f(x, y))$ for some f?

19.4. A surface is given by $(x(s,t), y(s,t), z(s,t))$ where s and t are parameters. What expression gives the normal?

19.5. Section 19.3.2 says "It is likely that smoothing the depth before differentiation is a good idea; incorporate this into $\mathcal{D}_x$." How would you do this?

19.6. Section 19.4.1 has "$\hat{\mathcal{R}}$ should consist of mostly zeros or very small values, because interpolation is quite good." Why is this true?

19.7. Section 19.4.2 has: "This model omits a blurring effect, which is easily observed. Distant objects will tend to be somewhat blurred as well, and nearby objects will tend to be sharper." What causes this effect? (a qualitative explanation is enough, and is easy).

PROGRAMMING EXERCISES

19.8. This exercise compares normals from single image normal predictors with depths from single image depth predictors.

- **(a)** Obtain an open-source single image depth predictor. For a set of images, obtain predictions of depth at least up to scale (check that your predictions are depth, rather than $a/d + b$, or considerable frustration may lie ahead). Use these depth predictions to obtain unit normal predictions, using the methods of Section 19.3.1. By eye, how good are the normals? how can you improve them (most likely is using a smoothed predictor of the depth gradient)?
- **(b)** Obtain an open-source single image normal predictor. For a set of images, obtain predictions of normal. Use these normal predictions to obtain a depth prediction, using the methods of Section 19.3.2. By eye, how good is the depth? how can you improve it?
- **(c)** Does any of this suggest a scheme for obtaining improved depth and normal predictions? Does this scheme work?

19.9. Obtain the wonderful dataset of images of the same scene under different lightings from Lukas Murmann, Michael Gharbi, Miika Aittala, and Fredo Durand at `https://projects.csail.mit.edu/illumination/`). You will use these images to investigate the sensitivity of a predictor (I say depth here, but others are interesting as well) to changes in illumination. They are arranged as 25 lightings each of over 1000 scenes. The lighting has been carefully arranged so that each of the 25 lightings is the same across all scenes (which takes considerable care – read the paper).

- **(a)** Obtain an open-source single image depth predictor. For each lighting of at least 10 scenes, compute the depth map. You will see that these are different. Why is this annoying?

(b) Now obtain a pooled estimate of the true scene depth from all the lightings of each scene. Be careful. If your depth predictor reports $a/d + b$, the constants may be different from image to image, which makes this tricky. Furthermore, if the light has caused the camera to saturate, the depth prediction tends to be unreliable, so there will be depth outliers.

(c) Construct a measure of the sensitivity of the depth predictor to lighting. You can do this by comparing the differences between depths for different images of the same scene to the pooled estimate of the true scene depth.

(d) Use this measure to compare two distinct depth predictors. Which is more sensitive to lighting? why?

(e) Can you improve the sensitivity of a predictor by pre-processing the input image?