CHAPTER 21

Image to Image Mapping using Classification Methods

Choose a *taxonomy* (a collection of labels), say "sky", "road", "pedestrian", "sidewalk", "car", "truck", and "unknown". Place a label at every pixel in an image, resulting in a map of the types of thing or stuff appearing in the image. This map is a *semantic segmentation* of that image. The labels in this example might be useful for the kind of outdoor images that autonomous vehicles observe, for example (Figure 21.1). You could predict a semantic segmentation for new images using a version of the master recipe. You must classify into many classes; and you must classify each pixel.

21.1 MULTI-CLASS CLASSIFICATION

There are some mild differences between multi-class classification and two-class classification, mostly in the details of loss and evaluation. It might be a good idea to re-read Section 20.1 (for general points and evaluation) and Section 20.2.3 at this point.

21.1.1 Predicting Multiple Categories with a Linear Predictor

Imagine you wish to classify something into one of k classes. You have a set of examples where the classes are known for each example. You can encode this information by associating a *one hot vector* with each training example. This vector has k components, one per class. The component corresponding to the class is one, and all others are zero. Write $\mathbf{y}_i$ for that vector for the i'th training example.

Your classifier must now produce a k dimensional vector for an example. This is quite commonly called a *score* Write $\mathbf{x}$ for the example and $\mathbf{s}$ for the vector of scores produced by the classifier. Then a linear classifier predicts

$$\mathbf{s} = \mathcal{A}\mathbf{x} + \mathbf{b}.$$

Stack the entries of $\mathcal{A}$ and $\mathbf{b}$ into a vector θ of all the parameters of the classifier. Notice there is a small disparity between a two class classifier as described in Section 20.2 and this description of a multiclass classifier. This minor nuisance is explored in the exercises **exercises**.

21.1.2 Multi-class Logistic Regression

Interpret the score as a probability using

$$P(\text{item is class } u|\mathbf{x}, \theta) = \frac{\exp[s_u]}{\sum_w \exp[s_u]}.$$

There are two natural ways to turn this expression into a loss that end up with the same expression (recall Section 20.2.3).

The loss could be the negative log-likelihood of the data $\mathbf{y}$ under the model. Write $\mathbf{q}$ for the vector whose u'th component is

$$q_u = \frac{\exp[s_u]}{\sum_w \exp[s_w]}.$$

Then the negative log-likelihood for the example is

$$C(\theta; \mathbf{x}, \mathbf{y}) = -\log\left[\mathbf{y}^T\mathbf{q}\right]$$

and averaging over the whole dataset yields

$$\mathcal{L}_{lr} = \frac{1}{N} \sum_{i \in \text{examples}} C(\theta; \mathbf{x}_i, \mathbf{y}_i).$$

Evaluating this expression might strike you as a bit of a performance but notice there is only one non-zero element in $\mathbf{y}_i$, which makes things somewhat simpler. In practice, an API will do this for you very efficiently.

21.1.3 Multi-class Cross Entropy

Alternatively, the loss could be the cross-entropy of Section 20.2.3. Recall the cross-entropy between a discrete distribution p and another discrete distribution on the same space q is

$$H_x(p, q) = -\mathbb{E}[p][\log q] = -\sum_u p_u \log q_u.$$

Regard $\mathbf{y}$ as a discrete distribution, and $\mathbf{p}$ as a discrete distribution on the same space. Now write

$$\mathcal{L}_{xe} = \frac{-1}{N} \sum_{i \in \mathcal{S}} \log\left[\mathbf{y}_i^T \mathbf{q}_i(\theta)\right].$$

Most APIs call the loss I have derived in two ways the cross-entropy loss. The advantage of the cross-entropy derivation is that it tells you what to do when your training data is uncertain **exercises**.

21.2 SEMANTIC SEGMENTATION

Choose a *taxonomy* (a collection of labels), say "sky", "road", "pedestrian", "sidewalk", "car", "truck", and "unknown". Place a label at every pixel in an image, resulting in a map of the types of thing or stuff appearing in the image. This map is a *semantic segmentation* of that image. The labels in this example might be useful for the kind of outdoor images that autonomous vehicles observe, for example (Figure 36.3). In that figure, notice that many different cars all have the same label. This might be fine for some applications – you don't need to know which car you should not bump into – but may not be good enough for others.

The distinction between *things* and *stuff* is important. A thing is an object one can count ("car", "truck", and so on – *count nouns* in linguist jargon). Stuff



FIGURE 21.1: *Semantic segmentation places a label taken from some taxonomy at each pixel in an image. On the* **left**, *three example images from a well-known dataset for semantic segmentation (KITTI). On the* **right**, *semantic segmentations of each image from the ground truth. The color at each location corresponds to the label (deep purple for roads, blue for cars, bright red for people, and so on).*

can't meaningfully be counted ("sky", "road", "grass", and so on – *mass nouns*). By convention, "unknown" is stuff. Because things can be counted, there can be multiple *instances* of a thing class, so many distinct cars on the road. Distinguishing between instances can be useful. For example, if you want to predict where a car will be in the near future, it helps to distinguish between individual cars because some will move and others won't. Marking different instances of each thing class label is known as *instance segmentation*. Some pixels might not be marked, because they are stuff pixels. Finally, *panoptic segmentation* marks every pixel. Here stuff pixels should get stuff labels and different instances of things should get different instances of thing labels (so, for example, "car-1", "car-2"). The terminology is not exactly established, so at least some references to the panoptic segmentation task appear not to label different instances with different labels. This section concentrates on semantic segmentation, so each pixel should receive a label; labels could be thing or stuff labels; but there is no need to distinguish between distinct instances of the same category.

Semantic segmentation largely follows a quite familiar recipe. If there are k classes, construct a u-net that maps an $M \times N$ image to a $k \times M \times N$ data block. At each pixel location, there is a k dimensional vector. Interpret the components of this vector as scores, one for each class, rather as in Section 36.3. Train this u-net using a dataset that consists of pairs (image, labelled image), where at each pixel location in the labelled image there is a k-dimensional one-hot vector identifying the class for that pixel. Construct a loss by averaging the multi-class cross-entropy loss evaluated at each pixel over all the pixels in the dataset.

This description is a jumping off point for a wide range of variants. There is room for a great deal of innovation in the architecture of the encoder and decoder

used to produce the results. You might predict coarse scale semantic segmentations first, then upsample and refine. You might weight the terms in the cross entropy loss to reflect the fact that some classes have more pixels than others. Label maps have quite strong spatial structure – you tend not to get one isolated "grass" pixel in an enveloping field of "sky", for example – and many variants explore procedures to impose this structure on predictions. Typical numbers of classes as of writing range from 13 to 40, and the recipe gets difficult to use in the form I have given when the number of classes is very large. One difficulty is the data block is very big. Another is that, when there are very many classes, there are typically few examples of many classes. Many variants are built around managing these problems.

21.2.1 Evaluation

At a high level, semantic segmentation is evaluated by scoring whether pixel labels are correctly predicted. Obtain a test set consisting of pairs (images, label maps). Choose a class c. For that class, there are two interesting sets: $\mathcal{G}_c$, the pixels that are labelled with c and $\mathcal{P}_c$, the pixels where the label c is predicted.

Definition: 21.1 *IoU or intersection over union for a class*

The IoU is given by

$$\mathrm{IoU}_c = \mathrm{IoU}_c(\mathcal{G}_c, \mathcal{P}_c) = \frac{\#(\mathcal{G}_c \cap \mathcal{P}_c)}{\#(\mathcal{G}_c \cup \mathcal{P}_c)}.$$

Definition: 21.2 *The Jaccard index for a class*

The Jaccard index of a system for a class is the same as the IoU for that class.

Write TP for the number of true positive pixels (ie $\#(\mathcal{G}_c \cap \mathcal{P}_c)$); FP for the number of pixels where c is predicted, but the true label is different (false positives); and FN for the number of pixels where c is not predicted, but the true label is c (false negatives). You can show that

$$\mathrm{IoU}_c(\mathcal{G}_c, \mathcal{P}_c) = \frac{TP}{TP + FP + FN}.$$

Annoyingly, the Jaccard index is not the same as the F-measure. Summarize performance by averaging the IoU per class over classes. The result is referred to as the *IoU*.



FIGURE 21.2: *The three images in the MS-CoCo dataset that contain an aeroplane, a bird and a person (the bird in the image on the* **left** *is the airline's logo). Note how objects are delineated with polygons and the relatively rich context in which the objects occur.*

Definition: 21.3 *IoU of a semantic segmenter*

The IoU is

$$\frac{1}{C}\sum_{c\in\mathcal{C}} IoU_c$$

In some datasets, labels are themselves distinguished between *category labels* (like "vehicle") and *class labels* (like "car"). For these datasets, it is quite usual to compute an IoU over all labels, an IoU over category labels, and an IoU over class labels. Comparison between these IoUs can offer some insight into the behavior of the method.

As Figure 21.1 shows, some cars in the image are large and some are small. Experience teaches that the IoU measure tends to favor methods that do well on large cars over methods that do well on small cars. To control this effect, introduce per instance weights that are large for small instances and small for large instances. The segmenter does not need to produce instance annotations for this strategy to work. It is enough for the ground truth to contain them. At a pixel that lies on a particular car in a particular image, use the ground truth annotations to compute N_{tc}, the number of pixels that lie on that car. Write $\overline{N}_c$ for the mean number of pixels on a car in the dataset, and write $w_{tc} = \overline{N}_c/N_{tc}$ for the weight (so pixels that lie on small cars get big weights). Any pixel on that car now contributes w_{tc} (rather than 1) to the count of true positives and false negatives, yielding iTP and iFN which are counted over all cars. In this case, predictions that are right or wrong and are on smalll cars are weighted higher than predictions that are on large cars. The weighted score iIoU for the class is given by

$$\text{iIoU} = \frac{iTP}{iTP + FP + iFN}$$

and the average of this score over classes is the evaluation metric.

21.2.2 Datasets

There is a rich collection of datasets for semantic segmentation.

- The Pascal VOC 2012 dataset can be found in many locations (the original source is `http://host.robots.ox.ac.uk/pascal/VOC/voc2012/`, but I have had little success with this server). The dataset originates with a challenge workshop held in 2012. It contains 1464 labelled training images and another 1449 validation images. Test images were held back for evaluation. There are 20 object classes, and a "background" class. Object classes cover a broad range (the "potted plant" has been challenging for years). The dataset is sufficiently well established as a reference dataset that it is available in PyTorch (look at `VOCSegmentation`).

- The Kitti semantic segmentation dataset can be found at `https://www.cvlibs.net/datasets/kitti/eval_semseg.php?benchmark=semantics2015`. There are 200 annotated training images and 200 validation images, from a benchmark launched in 2012. There are 30 classes, to do with the stuff and things an autonomous vehicle is likely to encounter. At the URL, there is a very extensive set of evaluations of different methods applied to this dataset. The dataset is sufficiently well established as a reference dataset that it is available in PyTorch (look at `Kitti`).

- The Cityscapes dataset can be found at `https://www.cityscapes-dataset.com`. There are 5, 000 images labelled carefully, and some 20, 000 frames that are weakly annotated. There are 30 classes, which are the same as the Kitti classes. At the URL, there is a very extensive set of evaluations of different methods applied to this dataset. The dataset is sufficiently well established as a reference dataset that it is available in PyTorch (look at `Cityscapes`).

- The MS-CoCo (common objects in context) dataset can be found at `https://cocodataset.org/#home`. There are some 123,287 images showing 886,284 instances of objects in context. Images are extremely varied (Figure 21.2). Objects are delineated by polygons. The dataset is used for a very wide variety of challenges.

21.2.3 Variants: Voxel Completion

21.3 PREDICTING EDGES

You should think of edge prediction as producing the result of a classifier – this is an edge point or this is not an edge point – at each pixel location in the input image. At every location in the image, the u-net will predict a score of the extent to which that location is an edge point, which can be decoded by, for example, thresholding. However, the problem has some special features. There are few edge points compared to non-edge points, and it can be difficult to be sure what the true edge points for an image are.

There is usable training data. The Berkeley Segmentation Dataset (500 image version at `https://github.com/BIDS/BSDS500`) contains a set of images that have been manually segmented. Segment boundaries can be taken as edges. Each image has been segmented by multiple annotators, who are not required to agree, so that the dataset contains some information about the difference in opinion between

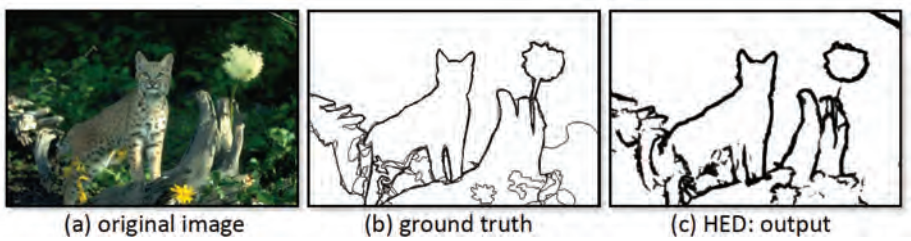


FIGURE 21.3: *Image to image methods can find edges satisfactorily. On the* **left**, *an image from the BSDS500 dataset. Multiple human segmentations of this image appear in the* **center** *(look closely: people largely agree with one another, but in some places you will see many boundaries close to one another). On the* **right**, *edge predictions made by an image to image mapping.* Image credit: *Part of Figure 1 of "Holistically-Nested Edge Detection" Xie and Tu 2015*

human annotators. There will be several distinct annotations of the same image, which may not agree.

21.3.1 Class Imbalance in Edge Prediction

Each training data item consists of an image and a set of annotations, one per pixel location. Write $(\mathcal{I}_i, y_i(\mathbf{x}))$ for the i'th data item, where $\mathcal{I}_i$ has $N_p(i)$ pixels. Notice that for $i \neq j$, $\mathcal{I}_i$ and $\mathcal{I}_j$ might be the same image (images have been annotated multiple times). Use the convention that

$$y_i(\mathbf{x}, \mathcal{I}) = \begin{cases} 1 & \text{if } \mathbf{x} \text{ is an edge point} \\ -1 & \text{otherwise} \end{cases}$$

The unet takes an image $\mathcal{I}$ and uses parameters θ to predict an "edgeness" score at each location. Write $s(\mathbf{x}, \mathcal{I}; \theta)$ for the score predicted by the unet at location $\mathbf{x}$. As in Section 20.2, interpret $s(\mathbf{x}, \mathcal{I}; \theta)$ as

$$s(\mathbf{x}, \mathcal{I}; \theta) = \log \frac{P(\mathbf{x} \text{ is edge} | \mathcal{I}, \theta)}{P(\mathbf{x} \text{ is not edge} | \mathcal{I}, \theta)}$$

and proceed to a cross-entropy loss. *This loss does not work well*, because there are very many more non-edge points than there are edge points, so the classes are *unbalanced*. A predictor can obtain a very good score by simply reporting no edges anywhere.

One way to deal with this effect is to introduce a weight to balance the classes. Write T for the total number of pixels in the annotated images (so if there are 3 annotations for an $M \times N$ training image, you count 3 M N pixels) and E for the total number of edge points in the annotated images. Then $\beta = E/T$ is the total fraction of edge points in the annotated training images. Then the reweighted loss

is for the value at location $\mathbf{x}$ in image $\mathcal{I}_i$ is

$$\begin{aligned}\mathcal{C}(\theta;\beta,\mathcal{I}_i,y_i(\mathbf{x}),\mathbf{x}) &= -(1-\beta)\left(\frac{1+y_i(\mathbf{x};\mathcal{I}_i)}{2}\right)\left[s(\mathbf{x};\mathcal{I}_i,\theta)+\log\left(1+\exp s(\mathbf{x};\mathcal{I}_i,\theta)\right)\right] \\ &\quad -\beta\left(\frac{1-y_i(\mathbf{x};\mathcal{I}_i)}{2}\right)\left[\log\left(1+\exp s(\mathbf{x};\mathcal{I}_i,\theta)\right)\right]\end{aligned}$$

(notice you want a high weight on the positives when edges are rare, which this achieves).

The overall loss is then

$$\mathcal{L}(\theta)=\frac{1}{N}\sum_{i\in\text{training set}}\left[\frac{1}{N_p(i)}\sum_{\mathbf{x}\in\text{locations}}\left[\mathcal{C}(\theta;\beta,\mathcal{I}_i,y_i(\mathbf{x}),\mathbf{x})\right]\right].$$

Train the u-net using this loss. Now recovering an edge map from an image is straightforward. Pass the image into the u-net which yields the score (negative log probability) that a pixel is an edge point at every pixel. Test that score against a threshold. If it is low enough (equivalently, if the probability the point is an edge point is high enough), it is an edge point.

21.3.2 Evaluating Predicted Edge Points

There is an evaluation procedure for edge detectors that compares predictions against human predictions. At a high level, the test is whether human edge points are the same as predicted edge points. The *F-measure* measures this property. Write TP (*true positive*) for the number of points where the method predicts an edge and so do humans; FP (*false positive*) for the number of points where the method predicts an edge and humans do not; TN (*true negative*) for the number of points where neither method nor human predict an edge; and FN (*false negative*) for the points where the method does not predict an edge but humans do.

Definition: 21.4 *Recall*

Recall is the fraction of predicted edge points that are also marked by humans as edge points. The recall of a system is given by

$$R=\frac{TP}{TP+FP}.$$

Definition: 21.5 *Precision*

Precision is the fraction of actual edge points (those marked by humans) that the method finds. The precision of a system is given by

$$P=\frac{TP}{TP+FN}.$$

You cannot properly evaluate a predictor using only one of these numbers. For example, you can get excellent recall by building an extremely cautious predictor that has no false positives, and very few true positives (though the precision will be low). Similarly, an enthusiastic predictor might label almost everything as positive, and so get a very good precision (but low recall). One can summarize these two numbers with an F1 measure, which is the harmonic mean of recall and precision.

Definition: 21.6 *The F-measure*

A system with precision P and recall R has F-measure

$$F = \frac{2}{\frac{1}{R} + \frac{1}{P}}.$$

With some effort, you can show that

$$F = \frac{2}{\frac{1}{R} + \frac{1}{P}} = \frac{2TP}{2TP + FP + FN}.$$

Notice that to get a large value of this number, both R and P must be large.

In practice, evaluation requires care, and it is important to have a consistent protocol, because humans are not consistent with one another. This means that a very good predicted edge point may be very close to, but not on top of, any human prediction. Such a point should be counted as a true positive. There is a standard protocol in place, with code available at `https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/resources.html`.

You can construct the edge map of an image at any scale, which offers a training opportunity. The decoder produces a sequence of data blocks with different spatial dimensions. Each of these blocks could be decoded using a simple linear map to a predicted edge map at those spatial dimensions. It turns out that doing so can help training significantly, because you can compel the intermediate layers to contain strong edge information ([]).

21.4 FINDING INTEREST POINTS

Learning to find and describe interest points is more interesting than edge detection, because it is much harder to produce training data by just annotating images. But interest points have quite distinctive properties. They should be covariant under transformations, and they should match well. These properties can be levered into training data that leads to very good learned interest point detectors. I describe one construction which captures the main points here, but there are a number of others.

21.4.1 Learned Interest Point Detectors: Setup

Interest points tend to be scattered, and don't form curves in images, meaning there is no reason to expect an interest point at every pixel. Tile an image with non-overlapping 8×8 tiles, and assume that there is only one interest point in each tile

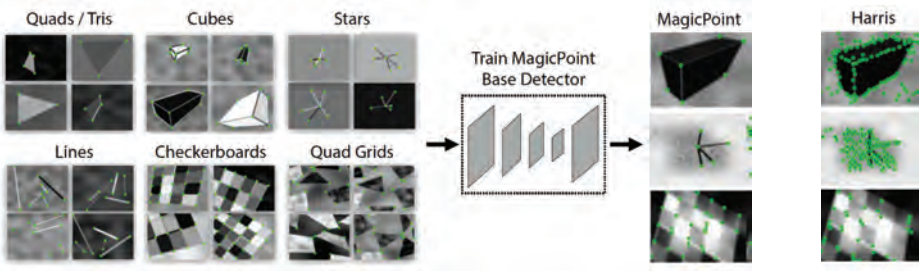


FIGURE 21.4: *Image to image methods can locate interest points very well, and are now unequivocally better than corner detectors. This figure shows results from the MagicPoint detector. On the* **left**, *examples from a collection of synthetic image datasets, where the location of the interest points is known. This data is used to train a detector as in the text, which predicts locations shown as green circles on the* **right**. *Notice the comparison with the Harris corner detector of Section 8.2.1. MagicPoint detections are less frequent, and more strongly associated with actual corners.* Image credit: *Part of Figure 4 of "SuperPoint: Self-Supervised Interest Point Detection and Description DeTone* et al.*, 2018.*

in an image (the 8 here isn't magical, but is the choice of a very effective method called Superpoint []). Then you could build a u-net that accepted an $8H \times 8W$ image (or a $3 \times 8HW$ block for a color image) and predicted a $65 \times H \times W$ block. Interpret the values in the 65 dimensional vector in a location as a score vector.

This score vector is the negative log probability for each of 65 cases, of which $64 = 8 \times 8$ cover the possibility that the interest point is at the corresponding pixel in the 8×8 block attached to that location, and the last covers the possibility that there is no interest point at all. Imagine you have a collection of pairs (images, interest point locations), where the interest point locations are a list.

You can turn this data into training data as follows. For each tile, if there is more than one interest point in the tile, choose one of those at random. Now every tile has either one interest point or none, so you know which of the 65 cases occurs at that tile. You could now train the image to image network to predict the interest points using the multiple class cross entropy loss of Section 21.1.3. The problem with this scheme is that getting a collection of pairs (real image, interest point locations) might be hard.

21.4.2 Creating Interest Point Data

However, you can *create* data with interest points in known locations. For example, place a dark triangle on a light background, and keep track of where the corners are. Notice that any reasonable warp of that image also has interest points at locations predicted by the warps, but has no others. As a result, very large quantities of synthetic data are relatively straightforward to produce. This strategy for producing an interest point detector is extremely strong (Figure 21.4). The results are comparable to the output of a corner detector, because the method produces a location that is well behaved under image warps, but not a neighborhood, orientation or

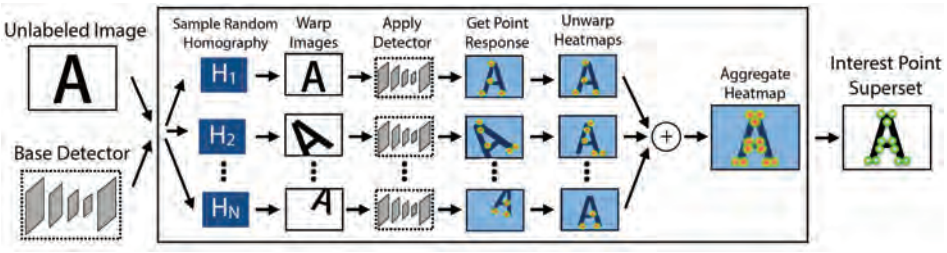


FIGURE 21.5: *The covariance properties that an interest point detector should have can be used to produce a fine-tuning loss that can be applied to unlabelled images. A transformed version of the image should have interest points that transform in a known way. The loss takes the image, applies various random transformations, forms heatmaps from the network output, then applies the appropriate inverse transformation to the heat maps and averages the results. This average represents a good target heatmap for the original image.* Image credit: *Part of Figure 4 of "SuperPoint: Self-Supervised Interest Point Detection and Description DeTone* et al., *2018.*

description.

A detector trained on synthetic images can be fine-tuned on unlabelled data. Interpret the $65 \times M \times N$ block produced by the u-net as a probability distribution. Write $\mathbf{b}_{ij}$ for the 65 dimensional vector at location i, j. Recall that this vector encodes the probability that an interest point is located in each of the 64 locations in an 8×8 image tile, together with the probability that there is not one. Form the 64 dimensional vector $\mathbf{p}_{ij}$ whose u'th element is

$$p_{uij} = \frac{\exp b_{uij}}{\sum_w \exp b_{wij}}$$

where u ranges over the locations of the interest points. This means there are only 64 values of u and the 65'th represents the possibillity there is no interest point. The components of $\mathbf{p}_{ij}$ *do not* sum to one. Now rearrange the components of $\mathbf{p}_{ij}$ so they form an 8×8 tile, putting the components in the location they represent. This process turns the input $M \times N$ image into a $1 \times (8M) \times (8N)$ block where the value at a pixel is non-negative and larger if the location has an interest point. Call this block a *heatmap* (it is "warm" where there are likely interest points and "cool" otherwise). Now take an unlabelled image and form its heatmap. Then apply (say) an affine transformation to the image, compute the heatmap for the result, then apply the inverse transformation to the heatmap. The two heatmaps should be the same, up to some minor issues to do with interpolation. This is the covariance of Section 8.2.

Fine-tuning on unlabelled data then proceeds as follows. Take the current interest point detector, and build a training dataset by: taking an unlabelled image; applying a collection of randomly selected transformations; computing the heatmap for each resulting image; apply the inverse of each transformation to each heatmap, and averaging the results; then using this average as a target distribution

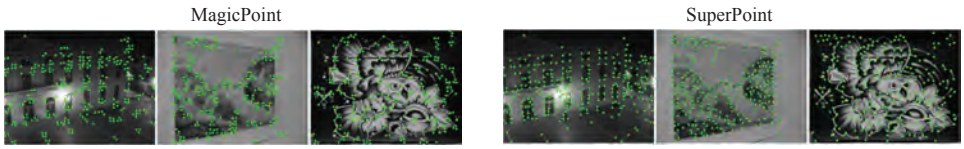


FIGURE 21.6: *Fine tuning MagicPoint on unlabelled data using the averaging procedure described in the text yields SuperPoint, and produces significant differences in the interest point locations that are marked.* **Left** *shows MagicPoint predicted locations;* **right** *shows SuperPoint predicted locations.* Image credit: *Part of Figure 7 of "SuperPoint: Self-Supervised Interest Point Detection and Description DeTone* et al.*, 2018.*

for training the unlabelled image. Now train the current interest point detector on this dataset. Repeat this process with the resulting detector as necessary. This approach leads to really significant improvements in performance (Figure 21.8).

21.4.3 Describing Interest Points

The locations of interest points are not much use without a description. This description will be used to match interest points so that, for example, images can be registered to one another. If you have an image and a transformed version image, then the interest points that correspond can be recovered from the transformation. This means that you know which descriptions should be similar. You also know that interest points that do not correspond should have different descriptions. This means you can learn descriptions using a self-supervised method.

A natural procedure is to have two decoders attached to the encoder (these decoders are sometimes called *heads*, so doing so produces a *multiheaded network*). One decoder produces the interest points, as above, and the other produces their descriptions. Because there is only one interest point per tile, it is enough to have one description per tile and upsample the descriptions by interpolation. Now take an image $\mathcal{I}$ and apply a transform $\mathcal{T}$ to the image to obtain $\mathcal{T}(\mathcal{I})$. This transform induces a correspondence between tiles, but remember the tiles are relatively coarse. For a point located at $\mathbf{x} = (i, j)^T$ in the image, write $\mathbf{u} = (u, v)^T = \mathcal{T}(\mathbf{x})$ for the transformed coordinates. Now write

$$s_{\mathbf{x},\mathbf{u}} = \begin{cases} 1 & \text{if } \|\mathbf{x} - \mathbf{u}\|_2 \leq 8 \\ 0 & \text{otherwise} \end{cases}$$

Here $s_{\mathbf{x},\mathbf{u}}$ is one if the tiles correspond (roughly - the tiles are on a grid) and zero otherwise. As in the case of edge detection, there are more pairs that do not correspond than there are pairs that correspond. Deal with this using a weight λ. Then for a pair $\mathbf{x} \in \mathcal{I}$tiles and $\mathbf{u} \in \mathcal{T}(\mathcal{I})$tiles, the cost

$$C(\mathbf{x}, \mathbf{u}) = \lambda s_{\mathbf{x},\mathbf{u}} \max(0, c_1 - \mathbf{d}^T(\mathbf{x})\mathbf{d}(\mathbf{u})) + (1 - s_{\mathbf{x},\mathbf{u}}) \max(0, \mathbf{d}^T(\mathbf{x})\mathbf{d}(\mathbf{u}) - c_2)$$

forces $\mathbf{d}(\mathbf{x})$ and $\mathbf{d}(\mathbf{u})$ to be similar when $\mathbf{x}$ and $\mathbf{u}$ correspond and different when

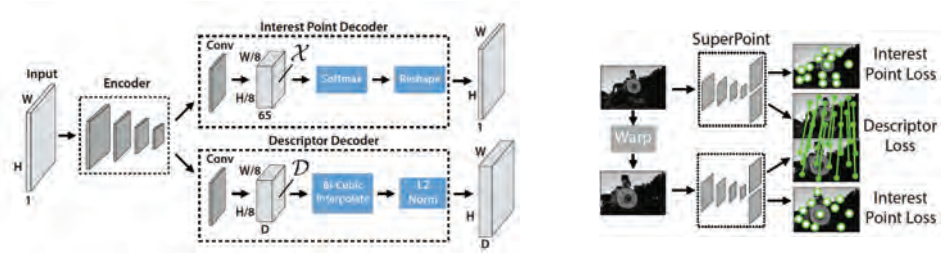


FIGURE 21.7: *Attaching a second decoder to SuperPoint makes it possible to produce descriptions for interest points, as well as their locations. On the* **left**, *the two decoder architecture. The second decoder produces a* $d \times (H/8) \times (W/8)$ *block from a* $H \times W$ *dimensional image. This block is then upsampled using interpolation to form a* $d \times H \times W$ *block. If the first decoder reports an interest point at* i, j *in the image, the* d *values in the second block at location* i, j *are taken as a vector describing that interest point. These vectors can be trained using a self supervised loss. You know which descriptors should and should not match between a warped version of an image and the original image, yielding the loss. Details in the text.* Image credit: *Parts of Figure 3 and 2 of "SuperPoint: Self-Supervised Interest Point Detection and Description DeTone* et al.*, 2018.*

they do not. This yields a loss

$$\sum_{\mathbf{x} \in \mathcal{I}\text{tiles}} \sum_{\mathbf{u} \in \mathcal{T}(\mathcal{I})\text{tiles}} C(\mathbf{x}, \mathbf{u})$$

The resulting descriptors are comparable in accuracy with SIFT descriptors []

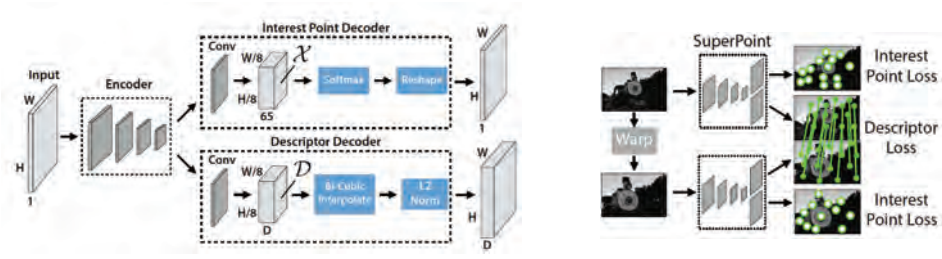


FIGURE 21.8: *Attaching a second decoder to SuperPoint makes it possible to produce descriptions for interest points, as well as their locations. On the* **left**, *the two decoder architecture. The second decoder produces a* $d \times (H/8) \times (W/8)$ *block from a* $H \times W$ *dimensional image. This block is then upsampled using interpolation to form a* $d \times H \times W$ *block. If the first decoder reports an interest point at* i, j *in the image, the* d *values in the second block at location* i, j *are taken as a vector describing that interest point. These vectors can be trained using a self supervised loss. You know which descriptors should and should not match between a warped version of an image and the original image, yielding the loss. Details in the text.* Image credit: *Parts of Figure 3 and 2 of "SuperPoint: Self-Supervised Interest Point Detection and Description DeTone* et al.*, 2018.*