Distance-vector Routing

- *Algorithm:* Distance-vector Routing (algo. 1)
- *Input:* A graph of network G, composed of n routers
- *Complexity:* $\mathcal{O}(n)$
- *Data structure compatibility:* Arrays
- *Common applications:* Implemented and used by some network protocols including RIP and BGP.

Problem. Distance-vector Routing

Given a graph representing the topology of a certain Network, which contains n routers, with each of them having a local forwarding table, initialize and maintain dynamically the forwarding table in order to send out the packets correctly and efficiently.

Description

Network routing algorithms, which are implemented by Network Layer protocols, play a major role in defining how the network is working. Usually, the routers in the network will perform certain routing algorithms to decide the next hop for a packet with certain destination. If we consider the intra-AS routing scenario, which means that the scale of the topology is relatively small within the control of a single administrator, one can always pursue the performance and use the paths with the minimum cost without considering external facts like policies [2].

Distance-vector Routing is one of the widely used dynamic routing algorithm in the field of intra-AS routing. Especially, it is a representative **decentralized routing algorithm**. By "decentralized", it assumes that the router always have and only have the information from its own side and all its **directly attached** neighbors. The algorithm is called "Distance-vector" because each node will collect all the information it knows in a distance vector, representing the current minimum-cost path to all the other nodes. This distance vector is dynamically updated whenever the router receives the incoming distance vector message from other routers. If any change occurs in the current distance vector, i.e., the router calculates a less-expensive path than current path, it will update its distance vector and send newly updated distance vector to all its directly attached routers. By dynamically update the distance vector within a network, the routers in the network can always keep the currently best path and thus construct an efficient network.

Before introducing the detailed algorithm, it is beneficial to first look at how the routers update its current distance vector based on the information from its neighbors. The essential point here is the **Bellman-Ford** equation defined as follows

$$D_x(y) = min_v c(x, v) + D_v(y)$$

where y is any node in network N and v is any directly attached neighbor of x. This is essentially a dynamic programming style equation which uses DFS. Miraculously enough, as long as all the nodes continue to exchange their distance vectors in an asynchronous fashion, each cost estimate $D_x(y)$ converges to the actual least-cost path [1].

The time complexity for this algorithm is easy to figure out, since we can imagine that the worst case is that the network forms a line shaped topology. Then it will require $\mathcal{O}(1)$ time for the information from the last router to pass the distance information to the second to the last router. Same situation happens to all the other routers. Then it will totally cost $\mathcal{O}(n)$ time for the first node to have the distance of the last router updated.

The pseudo-code of this algorithm is shown in algo.1.

Algorithm 1: Distance-vector Routing

Input : A dynamically network graph N

Output: The dynamically updated routers ensuring minimum-cost for all packets

Function `Initialize()`**:**
  foreach *router* r_i *in network* **do**
    foreach *router* r_j *other than* r_i *in network* **do**
      if r_j *is a neighbor of* r_i **then**
        $D_{r_i}(r_j) \leftarrow c(r_i, r_j)$; `/* c here represent the cost(weight) of the edge */`
      else
        $D_{r_i}(r_j) \leftarrow \infty$; `/* Set non-neighbor nodes to have infinite distance */`
      end if
    end foreach
    foreach *router* r_j *other than* r_i *in network* **do**
      if r_j *is a neighbor of* r_i **then**
        send distance vector D_{r_i} to r_j;
      end if
    end foreach
  end foreach
end
Function `Update(`$r_i, E(r_i, \cdot), D_{r_j}$`)`**:**
  `/* To update the distance vector for` r_i`,` $E(r_i, \cdot)$ `means the set of all the edges connected to` r_i `*/`
  if *any* $E(r_i, \cdot)$ *changes* **then**
    foreach *router* r_x *in network* N **do**
      `/* We need to consider the change of edge due to route living the network, so we should traverse all the other node to update the new distance */`
      $D_{r_i}(r_x) = min_v\{c(r_i, v) + D_v(r_x)\}$
    end foreach
  else if r_i *receives a new distance vector* D_{r_j} **then**
    foreach *router* r_x *in network* N **do**
      $D_{r_i}(r_x) \leftarrow min\{D_{r_i}(r_x), c(r_i, r_j) + D_{r_j}(r_x)\}$;
    end foreach
  end if
  if *any* $D_{r_i}(r_x)$ *changes* **then**
    foreach *router* r_j *other than* r_i *in network* **do**
      if r_j *is a neighbor of* r_i **then**
        send new distance vector D_{r_i} to r_j;
      end if
    end foreach
  end if
end
`/* initialize each router */`
`Initialize();`
`/* keep updating dynamically */`
while *true* **do**
  foreach *router* r_i *in network* N **do**
    `Update(`$r_i, E(r_i, \cdot), D_{r_j}$`);`
  end foreach
end while
return

Review

Based on the analysis above, it seems that the Distance-vector routing algorithm is both efficient and space saving, Indeed, it's implementation allows this algorithms to handle relatively huge network. But there is a fatal problem people have spotted which may prevent it from practical use.

We consider the following situation. 1

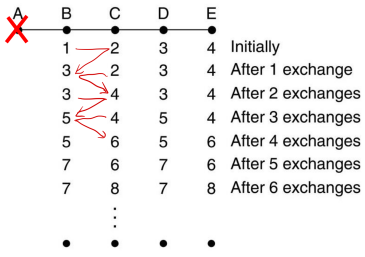

Figure 1: The Count-to-infinity problem
[2]

When a router A lives the network, its neighbor B will detect this change and seek to update its distance vector. The router B then check its forwarding table and find a new route via router C with cost 2. So by the algorithm we discussed above, B will use the Bellman-Ford equation to update the distance to 3, and resend the updated distance vector to C. Similarly, C will do the same update work and resend the distance vector to its neighbors. But as we see in Figure.1, the distance to A will keep increasing recurrently forever. Since sending message in the network will consume network resources, the network will crash. This fatal error is called **Count-to-infinity Problem**.

Luckily, this problem can be solved with various practical method and is indeed solved by Network Layer protocols like RIP and BGP. The RIP(Routing Information Protocol) limits the network to be no more than 15 hops. It means that if the router find that the distance vector has been updated for more than 15 times, the router will treat the distance to that node as infinity. This sacrifies the potential of adopting this protocol to a large scale of network. The BGP(Border Gateway Protocol) however, choose to send the path information together with the distance vector, so that when the router finds that the distance information it collects from its neighbor is actually calculated based on the former information given by itself, it will decide to ignore this update and replace the distance to infinity. Using this method, it is safe to delete a router.

References.

[1] Dimitri P Bertsekas. “Linear Network Optimization”. In: *The MIT Pr* (1991) (cit. on p. 1).

[2] Liping Shen. *Network Layer, CS339 Computer Networking, SJTU*. 2021 (cit. on pp. 1, 3).

Link-state Routing

- *Algorithm:* Link-state Routing (algo. 2)
- *Input:* A graph of network G, composed of $|V|$ routers and $|E|$ links
- *Complexity:* $\mathcal{O}(|V|^2)$, can be optimized to $\mathcal{O}(|E| + |V| \log |V|)$
- *Data structure compatibility:* Priority Queue, especially Fibonacci Heap
- *Common applications:* Implemented and used by some network protocols including OSPF.

Problem. Link-state Routing

Given a graph representing the topology of a certain Network, which contains $|V|$ routers, with each of them having a local forwarding table, initialize and maintain dynamically the forwarding table in order to send out the packets correctly and efficiently.

Description

Network routing algorithms, which are implemented by Network Layer protocols, play a major role in defining how the network is working. Usually, the routers in the network will perform certain routing algorithms to decide the next hop for a packet with certain destination. If we consider the intra-AS routing scenario, which means that the scale of the topology is relatively small within the control of a single administrator, one can always pursue the performance and use the paths with the minimum cost without considering external facts like policies [5].

As we discussed in Problem-60, the Distance-vector is a representative **decentralized routing algorithm**, which essentially means that the algorithm is carried out by the collaboration of all the routers in the network. In the contrast, the **centralized routing algorithm** is a class of routing algorithms that each router has the whole information and thus can carry out the routing algorithm all by itself. The link-state algorithm is the representative algorithm belonging to this class. By "link-state", the routers are supposed to know all the current link costs in the whole topology.

Since a router is aware of the whole topology, this problem can be then easily transformed to a problem which requires finding the minimum cost path in a weighted graph. The popular Dijkstra's algorithm will be a efficient method to solve this problem. Note that if we use the intuitive searching approach, which traverse the entire graph to find the next node with minimum cost, we will result in a overall time complexity of $\mathcal{O}(|V|^2)$. A priority queue can be used in the algorithm to reduce the time complexity. Particularly, a Fibonacci Heap can be used to implement and then reduce the time complexity to be $\mathcal{O}(|E| + |V| \log |V|)$.

The pseudo-code of this algorithm is shown in algo.2. And we define some of the parameters as follows in order to represent the algorithm better:

1. $D(v)$. The cost of the least-cost path from the source node to the destination node v as of this iteration of algorithm.
2. N'. Subset of nodes; v is in N' if the least-cost path from the source to v is definitely known.

[2]

Algorithm 2: Link-state Routing

Input : A dynamically network graph N
Output: The dynamically updated routers ensuring minimum-cost for all packets

```
Function Initialize():
    foreach router r_i in network do
        foreach router r_j other than r_i in network do
            if r_j is a neighbor of r_i then
                D_{r_i}(r_j) ← c(r_i, r_j);          /* c here represent the cost(weight) of the edge */
            else
                D_{r_i}(r_j) ← ∞;                    /* Set non-neighbor nodes to have infinite distance */
            end if
        end foreach
        foreach router r_j other than r_i in network do
            broadcast the link state (A graph N_{r_i}) to that router;
        end foreach
        queue_{r_i} ← [r_i];                         /* queue is a min-Heap or Fibonacci Heap */
    end foreach
end
Function Update(r_i):
    while N' ≠ N do
        pop the minimum node m from queue_{r_i} and add it to N';
        foreach neighbor of m v_i that is not in N' do
            D(v) = min{D(v), D(m) + c(m, v)};
            push v_i into the queue_{r_i};
        end foreach
    end while
end
Initialize();                                        /* initialize each router */
                                                     /* keep updating dynamically */
while true do
    foreach router r_i in network N do
        Update(r_i);
    end foreach
end while
return
```

Review

Compared with the Distance-vector routing algorithm, the Link-state routing algorithm needs more space for the router to store all the topology of the network. Also, it requires a larger time complexity to initialize and update the forwarding table. But we have to admit that the Link-state routing algorithm is **relatively easy to implement** compared with the Distance-vector routing algorithm. Also, it will not suffer from the "Count-to-infinity" problem we discussed in Problem-60. The wide-spread OSPF protocol is designed based on using this algorithm in multi-level network, and it is adopted by SJTU's school network [5]. However, there are also two serious shortcoming or fatal problems for Link-state routing algorithm.

The first problem is that, in the pseudo code provided above, we do not take the time of broadcasting into consideration. That is, when the network needs to be initialized, it will actually cost far more time than we expected. Also, the network resources consumed by the broadcasting is huge. For example, we consider the following real-world problem:

In recent years, multihop mesh networks have been advocated as a cost-effective approach for providing high-speed last-mile connectivity, supporting community networks, and so forth [4]. This technology is widely used in places like high buildings. However, one drawback of using multichannel communications is the high overhead

involved in broadcast operations: a transmitter should transmit a broadcast packet to all channels that may be occupied by receivers. This makes certain broadcast-intensive mechanisms, such as link-state routing, difficult to implement. [3].

From this concrete example, we can see that the broadcast through the network a costly operation. Thus needs some advanced technology to solve such problem.

The second problem is that, there might be some oscillations in the network under certain condition, which is shown in 2. Since all the routers update its forwarding table synchronously, it stands a large possibility to have

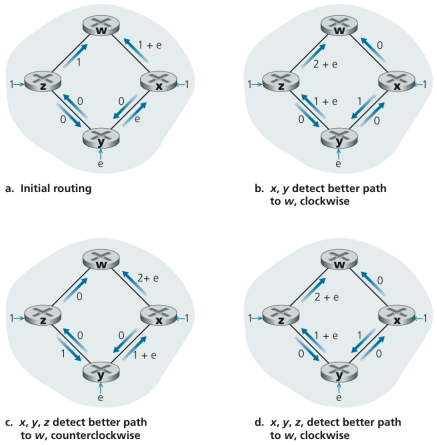

Figure 2: The Oscillation Problem
[2]

all the routers find the same least-cost path, and the packets are then all pushed to that path. Then the same situation happens to the original path. One trivial solution is to make different routers update their forwarding table asynchronously. However, it is find that the initially unsynchronized behaviors of each routers updating their routing table tend to be synchronized [1]. So the only way to solve this problem is to randomize the time that the routers update its forwarding table.

The overall performance of Link-state routing is excellent, and there is no obvious winner when compared with the Distance-vector routing algorithm. Both algorithms are widely used in today's network.

References.

[1] S. Floyd and V. Jacobson. "The synchronization of periodic routing messages". In: *IEEE/ACM Transactions on Networking* 2.2 (1994), pp. 122–136. DOI: 10.1109/90.298431 (cit. on p. 6).

[2] Keith W.Ross James F.Kurose. *Computer Networking A Top-Down Approach*. Pearson, 2021 (cit. on pp. 4, 6).

[3] Cheolgi Kim, Young-Bae Ko, and Nitin H Vaidya. "Link-state routing without broadcast storming for multichannel mesh networks". In: *Computer Networks* 54.2 (2010), pp. 330–340 (cit. on p. 6).

[4] P. Kyasanur, Jungmin So, C. Chereddi, and N. H. Vaidya. "Multichannel mesh networks: challenges and protocols". In: *IEEE Wireless Communications* (2006) (cit. on p. 5).

[5] Liping Shen. *Network Layer, CS339 Computer Networking, SJTU*. 2021 (cit. on pp. 4, 5).

Max-min Fairness

- *Algorithm:* Max-min Fairness (algo. 3)
- *Input:* n client-server pairs, sharing a same bottleneck in the network
- *Complexity:* $\mathcal{O}(n^2)$

- *Data structure compatibility:* Array, linked list
- *Common applications:* Implemented and used by some network protocols including TCP.

Problem. Max-min Fairness

Given n client-server pairs, each of them have different hosts compared with other links, but they all need to transfer the packet through a bottleneck with its bandwidth to be r. The client-server pairs may have different bandwidth respectively as well. The max-min fairness require the bandwidth of the bottleneck to be fairly distributed.

Description

The real network system has complex topological structures. Billions of hosts are connected by numerous links with different bandwidth. Although there are often different paths for one host to reach the other host, but it indeed happens that at some point of the network, there is only one link between two hops, with each of the hops connected to several hops. If all the hops belong to different link, it will result in a condition that several links are trying to share the only one bottleneck. Under this condition, the Internet is supposed to allocate the bandwidth equally under the constrains of the bandwidth of different links. A congestion-control is said to be fair if the average transmission rate of each connection is approximately R/n, where R is the bandwidth and n is the number of links [2]. But the Internet also follows the rule of serving with **best-effort**, which means that the network will allocate a larger throughput if both the bottleneck and some link have spare bandwidth.

The trivial case is that, when all the links have the same bandwidth, then it is obvious that the bandwidth will be equally divided into n parts with each of them with bandwidth begin either R/n or the maximum bandwidth of the link (when the bandwidth of the bottleneck is greater than the sum of all the bandwidth of the links). The fairness can be easily reached. However, when the links have different bandwidth, which is a more common case in real life, the bandwidth, if large enough, will not be shared equally due to the **best-effort rule**.

To combine the two rules discussed above, the concept of **Max-min Fairness** was defined by Jaffe as: Max-min Fairness is said to be achieved by an allocation if and only if the allocation is feasible and an attempt to increase the allocation of any participant necessarily results in the decrease in the allocation of some other participant with an equal or smaller allocation [1]. In other word, if the Max-min fairness has been reached, you cannot increase one flow without sacrificing other flows.

To reach the Max-min fairness at a bottleneck, the usually called **water filling** algorithm can be applied. The basic idea is

1. Start the algorithm with each link sharing 0 throughput.
2. Increase the throughput of each link in the same speed.
3. If the bottleneck is shared already, then we are done. Otherwise we will reach the first link (or links) that is fulfilled.
4. Get rid of the fulfilled links, redo the step 2 to step 3 recursive until the bandwidth of the bottleneck has been all used.

For example, given a bottleneck with the bandwidth $250Mbits/s$ and four links with bandwidth $50Mbits/s$, $50Mbits/s$,$100Mbits/s$,$100Mbits/s$. The water-filling algorithm will first fill four links with $50Mbits/s$ each, and then disregard the fulfilled links, and fill the last two links with $75Mbits/s$ each.

The time complexity for this algorithm is $\mathcal{O}(n^2)$, since the worst case requires n updates with each update fulfilling one link with a traversal through the array.

The detailed pseudo code is shown as follows 3.

Algorithm 3: Max-min Fairness with water filling

Input : n client-server pairs, sharing a same bottleneck in the network with bandwidth r
Output: An allocation of bandwidth to reach the Max-min fairness state

```
total ← n;
list ← [ ];
Througput ← [1 : n] and initialize to 0;
foreach link do
    insert link into list;
end foreach
while Total ≠ 0 && list ≠ empty do
    min ← the minimum bandwidth among all the links in the list;
    if total/sizeof(list) ≤ min then
        foreach link i in the list do
            Throughput[i] ← total/sizeof(list);
            total ← total - total/sizeof(list);
        end foreach
    else
        foreach link i in the list do
            Throughput[i] ← min;
            total ← total - min;
            if bandwidth of i is minimum then
                delete i from list;
            end if
        end foreach
    end if
end while
return
```

Review

The Max-min fairness is one of the core fairness that is implemented in the network nowadays. It is adopted by the TCP protocol in Transport Layer and then provide reliable network connection between two hosts. However, another aspect of network connection is the efficiency of transporting packets through the links. Thus, one of the critical question that one must answer is: **Is Max-min Fairness efficient**?

The answer for this question is that, the Max-min fairness is not the most efficient way in transmitting packets even when the length of the file sent by different links are equal. If we consider two links with their bandwidth to be $100Mbits/s$, and they share the bottleneck with the bandwidth to be $100Mbits/s$ as well. The Max-min Fairness will allocate each link with a $50Mbits/s$ throughput each connection. Assuming that both links are sending a $100Mbits$ file, then it will take 2 seconds for two files to be fully transmitted. Another strategy is that we let link 1 occupy all the bandwidth first and then the link 2. In this case, the first file is transmitted in the first second and the second file is transmitted in the second second. The efficiency of the link is actually evaluated by analyzing the **average flow completion time**. That is the average moment that the file has been fully transmitted. In this situation, the Max-min Fairness version has both files fully sent at the moment $2s$, the average flow completion time is $t = 2s$. While the second strategy has two files sent in moment $t = 1s$ and $t = 2s$, the average flow completion time is $\frac{1+2}{2} = 1.5s$, which is shorter than $2s$ [3].

From the discussion above, we analyze the Max-min fairness together with the water filling algorithm which makes the network throughput converge to the Max-min fairness state. We also see that the Max-min fairness sacrifices part of the transmission efficiency in order to implement the fairness of network connection. When we consider the overall performance of Max-min fairness, it is outstanding.

References.

[1] Jeffrey Jaffe. "Bottleneck flow control". In: *IEEE Transactions on Communications* 29.7 (1981), pp. 954–962 (cit. on p. 7).

[2] Keith W.Ross James F.Kurose. *Computer Networking A Top-Down Approach*. Pearson, 2021 (cit. on p. 7).

[3] Shizhen Zhao. *Network Layer, CS339 Computer Networking, SJTU*. 2021 (cit. on p. 8).

Faster R-CNN(Region Proposal Networks)

- *Algorithm:* Faster R-CNN(Region Proposal Networks)
- *Input:* A picture p
- *Complexity:* not the main concern
- *Data structure compatibility:* picture
- *Common applications:* Image matching, object detection, recognition and location

Problem. Faster R-CNN(Region Proposal Networks)

The original R-CNN and Fast R-CNN have already provide us with efficient ways to detect and classify the objects in a given picture. But they rely on some offline algorithm to generate proposal region. It takes time and requires offline calculation. The Faster R-CNN enables a faster and more efficient object detection and classification than R-CNN and Fast R-CNN by using the Region Proposal Networks.

Description

The object detection and classification is the major field of interest in the computer vision. The basic task is that, given a picture, there might be some objects in the picture. The computer should be trained to be able to tell what objects are in the picture. In the past fifteen years, the major detection method is using the feature descriptors. Among them, the SIFT and HOG are the most popular feature descriptors. However, the time consumption and the error rate is relatively high. Then in 2012, the newly developed CNN "AlexNet", successfully reached an error rate of 15%. This milestone draws people's attention back to CNN, and the R-CNN, proposed by Girshick , became popular at that time. It basically combines the region proposal and CNNs performing the following three steps (also see picture 3):[1]

1. Given a picture, extract about 2000 region proposals. This method takes the advantage of sliding window.
2. Use CNNs to create feature vector with fixed length for each region.
3. Classify each region using SVM (support vector machines).

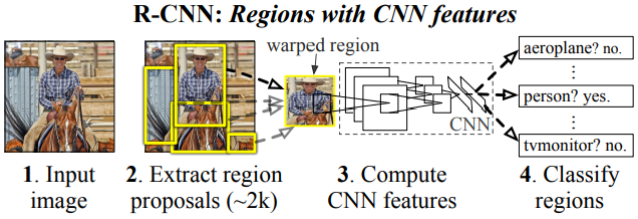

Figure 3: Object Detection System Overview
[1]

Then, the algorihtm of R-CNN has been improved. As discussed in [2]: Although region-based CNNs were computationally expensive as originally developed, their cost has been drastically reduced thanks to sharing convolutions across proposals. The latest incarnation, Fast R-CNN, achieves near real-time rates using very deep networks, when ignoring the time spent on region proposals.

So the bottleneck for the further development of the R-CNN lies in the improvement of the algorithms to generate region proposals. And this was solved by applying the **Region Proposal Networks (or RPN)** to the process. And the application of this technique gives the name **Faster R-CNN**.

The overall layout of Faster R-CNN consists of four different stages, as shown in the following figure 4. It is

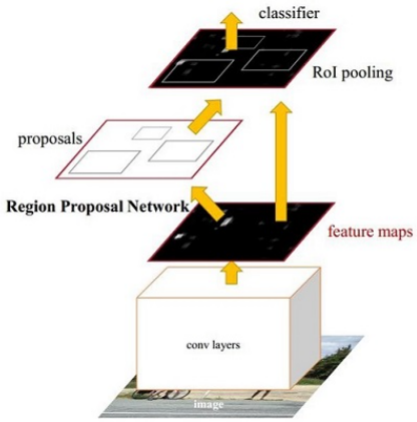

Figure 4: Overall layout of Faster R-CNN
[1]

divided into four layers with each of them performing certain kind of work:

1. Convolution layers. The convolution layer plays the transformation of input picture just as most of the CNN algorithms do. A feature map will be generated after all the operations and calculations done in the convolution layer. This feature map will be used in the following layers.

2. Region Proposal Network. This layer is the most critical layer in the Faster R-CNN algorithm, since it is the very reason that makes Faster R-CNN a real-time object detection algorithm. This layer will first judge whether the anchor (the reference box of the proposals) is positive and then amend the anchor with bounding box regression.

3. RoI pooling. This pooling layer is relatively simple compared with other layers. This layer will gather the feature map and proposals generated from convolution layer and Region Proposal network. And then it will send these message to the classifier.

4. Classification. It will use the proposal and feature maps provided by the RoI pooling layer and calculate as well as classify the proposal. It will also use the bounding box regression again to set down the final position of the detection box.

Convolution Layer

Just like all the CNNs, the convolution layers contains several convolution layers as well as the relu layers and pooling layer. Specifically, there are 13 convolution layers, 13 relu layers and 4 pooling layers used in this model, as shown in figure 5 The convolution layers, as the name indicates, will do a convolution to the picture pics and get the feature maps. Note that we have totally 13 convolution layers, with all of them having the same core with size 3, while the parameters in the matrix can differ. Also, the Faster R-CNN does padding with size 1 each convolution layer, so that the convolution result will have the same size as the map before convolution, as shown in figure 6

Figure 5: The detailed structure of convolution layer

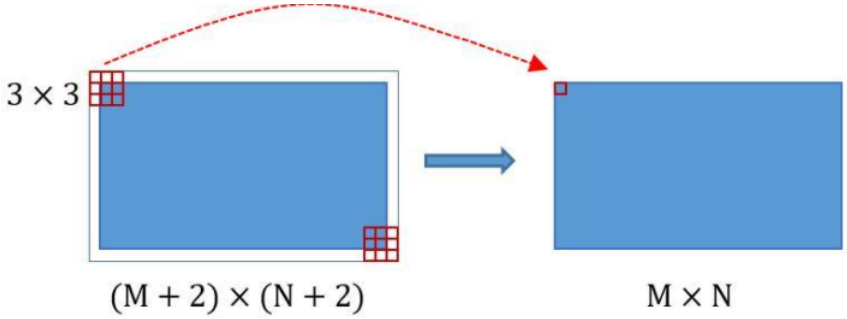

Figure 6: The convolution of convolution layer

The relu layers are layers that performing the activation functions. Note that the activation functions are important because it enables the CNN to fit curves. Without the activation functions, the CNN can only fit linear input [3]. The Rectified Linear Unit (ReLU) is applied in these layers.

The pooling layers are similar to the convolution layers, except that the core size is 2 and the stride is 2 as well. From the discussion above, one can deduce that each pooling layer will shrink the picture to half of the initial size in both height and width. As a result, the four pooling layer shrink the input to $\frac{1}{16}$ of the original size in both width and height. This will make each feature map corresponding to the original picture.

Region Proposal Network

The basic structure of Region Proposal Network is shown in the figure 7 It can be clearly seen that the Region

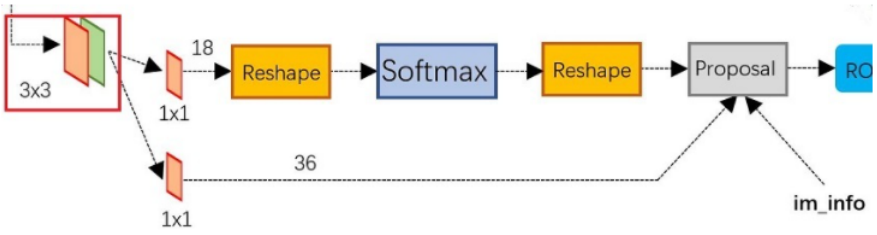

Figure 7: The overview of Region Proposal Network [2]

Proposal Network is divided into two sub-lines, the first line uses the Softmax to classify or filter the anchors, while the second line calculates the offset with respect to the anchor using the bounding box regression. Then the proposal step will combine the information from two lines. Since the position of the anchor and the offset are all calculated, the position of the final proposal will be fully decided. Note that some proposals with their sizes being two small or with their positions out of original range will be ruled out.

Softmax

The Softmax will judge the anchors to be positive or negative. So what exactly anchors are and how to judge them? Generally speaking, the anchors roughly define the position of the proposals. In faster R-CNN, there are in total 9 anchors with different high-width ratio in the range of 1:2 to 2:1. Notice that there is a process to

reshape the picture before the Softmax, which will resize the input to appropriate size so that it can be covered by different anchors as a whole.

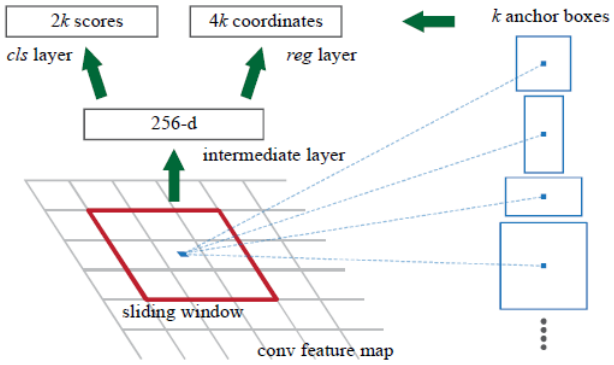

Figure 8: Translation-Invariant Anchors
[2]

As we can see from the figure 8, the center of the sliding window on the feature map has a base point, and 9 anchors with different size will be applied to the sliding window. These anchors are the rudiment of the final windows in the result. Though the current anchors are neither precise or effective (may contain no objects), they are ready for further operations.1

The Softmax is another widely used activation function that can perform a classification. It takes an array as input, and judge on the maximum value of the array, then map it into the set $\{0, 1\}$. In this method, the Softmax can tell whether a given anchor contains a solid object.

Bounding Box Regression

The second parallel line in the Region Proposal Network is the bounding box regression process. This process requires to modify the anchors so that it can contains the whole object instead part of the object. For example, as shown in the figure 9 The red rectangle is the original anchor, which contains only the core part of an

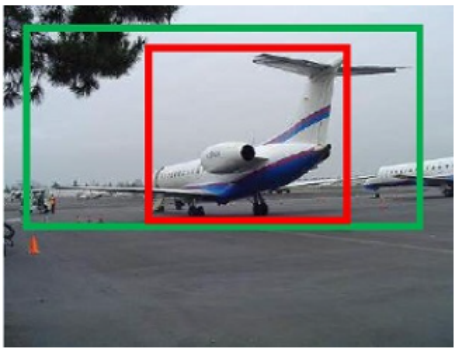

Figure 9: Bounding Box Regression Example

aeroplane. However, the object detection requires the whole object contained in the rectangle, which means the green rectangle is preferable. Thus, the bounding box regression is introduced to modify this anchor. Obviously, mapping the original anchor to the desired one needs two steps – translation and expansion. The bounding box regression introduces four mapping functions $d_x(\cdot)$, $d_y(\cdot)$, $d_w(\cdot)$, $d_h(\cdot)$, which corresponds to translation in x-axis, translation in y-axis, expansion in width, expansion in height, respectively. And the four mapping formula is designed as

$$G_x = A_w \cdot d_x(A) + A_x$$

$$G_y = A_h \cdot d_y(A) + A_y$$

$$G_w = A_w \cdot exp(d_w(A))$$

$$G_h = A_h \cdot exp(d_h(A))$$

where G is the anchor after the mapping and A is the original anchor. These four mapping functions can be learned using the linear regression model.

Proposal

The proposal step will perform the following steps and then send the result to the Rol Pooling layer:

1. Based on the positive max score, select a certain number of anchors with highest score. This operation will select the positive anchors after bounding box regression, which are exactly what we are interested in.
2. Limit the borders of the picture, cut off part of the positive anchors which lay outside of the graph.
3. Get rid of the positive anchors that are two small.
4. Perform the NMS (Non-maximum Suppression) to all the remaining anchors. The NMS will select the anchors with biggest size if several anchors overlapping with each other.
5. Output the proposal result in the form of one dimensional array with four entries, representing the four border lines of the proposal.

Rol Pooling

The traditional CNN, it will require the input size of the picture to be a certain number, and the output will also be a certain size. However, when the input sizes are different to what the model requires, the picture needs either being cropped or being wrapped (zooming in or out). Both ways destroy the original information of the input. The Rol pooling is designed to solve this problem.

To begin with, we notice that the proposals should be in the size of $M \times N$, which in the order of the original input picture. The Rol Pooling will first map them into the size $\frac{M}{16} \times \frac{N}{16}$ which is in the size of a feature map. And for a certain Rol Pooling with input parameter $pooled_w$ and $pooled_h$, it will divide each of the shrunk proposal areas into $pooled_w \times pooled_h$ small squares. Then it will perform a max pooling to each of the mesh, selecting the maximum number in the mesh. It can now ensure that the output is in the size $pooled_w \times pooled_h$, no matter what the input size is.

Classification

There is nothing special about the classification. The classification is composed of fully connected layers, which may be implemented by convolution and map every element from the upper layer to an element in the next layer. Basically, this process can be trained beforehand and thus be able to tell what exactly the object in the anchors are.

References.

[1] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 580–587 (cit. on pp. 9, 10).

[2] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. "Faster r-cnn: Towards real-time object detection with region proposal networks". In: *Advances in neural information processing systems* 28 (2015), pp. 91–99 (cit. on pp. 10–12).

[3] Rui Wang. *Sequence Tagging, CS438 Information Extraction of Internet, SJTU*. 2021 (cit. on p. 11).

SIFT

- *Algorithm: SIFT* (algo. 4-5)
- *Input:* the original picture
- *Complexity:* not the main concern
- *Data structure compatibility:* picture
- *Common applications:* image matching, object recognition

Problem. SIFT

In the field of computer vision, the recognition of objects is a core problem. Despite difference of illumination, scale, rotation, distortion, or noise, one algorithm should capture some invariant characteristics of one object, in order to recognize it, and make an effective matching between two snapshots of this same object. SIFT, abbreviation for Scale Invariant Feature Transform, is such an algorithm to extract distinctive characteristics of one image.

Description

The most fundamental and most critical problem of computer vision is to recognize objects or motions in an image like human beings. Human's vision processing system can extract the basic information of one object even the picture of that object varies a lot, like being scaled, rotated or placed under different illuminating conditions.

SIFT, abbreviation for Scale Invariant Feature Transform, gives a solution to that problem. It can extract some "keypoints" as the features of one image, which is invariant under scaling and rotations, and part of change in illuminating condition and viewpoint. In order to improve the efficiency of the algorithm, SIFT use a cascade filtering approach, in which the potential keypoints are filtered on each stage, to minimize the amount of data that needs to be processed. The stages are: 1) scale-space extrema detection; 2) keypoint localization; 3) orientation assignment; 4) keypoint descriptor. After the four stages, the scale-invariant coordinates will be extracted from the original picture. [2]

Scale-space Extrema Detection

Constructing Difference of Gaussian Pyramid

In order to find the characteristics that are invariant to the scaling, one needs to search across all the scale space, which is built by the convolution between the original picture and the **Gaussian function**

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2} \tag{0.0.1}$$

By make the convolution of $G(x, y, \sigma)$ and the input image $I(x, y)$, one can get the scale space of the image as a function $L(x, y, \sigma)$

$$L(x, y, \sigma) = G(s, y, \sigma) * I(x, y) \tag{0.0.2}$$

where σ is a coefficient which reflects the scale level. This is also called **Gaussian blur**, which will filter out the high dimension factors in one picture, in order to get a more blur one, and greater σ features better blurring.

This is imitating objects in human's vision when they are faraway. When σ differs, all the $L(x, y, \sigma)$ compose a scale space, and it can be proved that Gaussian is the only function that provides a linear scale space [3]. The algorithm attempts to stack the convoluted picture together, from the smallest σ (clearest) to the biggest σ, to construct the whole scale space.

After the construction of the octaves of Gauss-blurred pictures, the algorithm make the difference between the neighboring two pictures in each octave. This is called **Difference of Gaussians (DoG)**. DoG provides a very close approximation to the scale-normalized Laplacian of Gaussian [1], which can be used to find the extreme values.

For the implementation, to increase efficiency, SIFT divide the process into several octaves. In each octave, since s valid (non-marginal) DoG images are required (in the right of Figure 10) , there are $s+3$ images should be taken in octaves (in the left of Figure 10), and σ should obey the iteration relationship

$$\sigma_{o,a+1} = 2^{\frac{1}{s}}\sigma_{o,a} \tag{0.0.3}$$

Between octaves, SIFT requires σ enlarges to its twice when it enters the next octave. Hence, the total iteration formula for σ can be deducted

$$\sigma_{o,a} = 2^{o+\frac{a}{s}}\sigma_0 \tag{0.0.4}$$

where $a \in [0, 1, \dots s, s+1, s+2]$, and σ_0 is the initial value taken for Gaussian. Notice that

$$\sigma_{o+1,0} = \sigma o, s = 2^{o+1}\sigma_0 \tag{0.0.5}$$

$(o+1, 0)$ and (o, s) possess the same σ, since, one can simply jump to the next octave by re-sampling the third-to-last image in current octave, by only taking half the pixels to make it more blur.

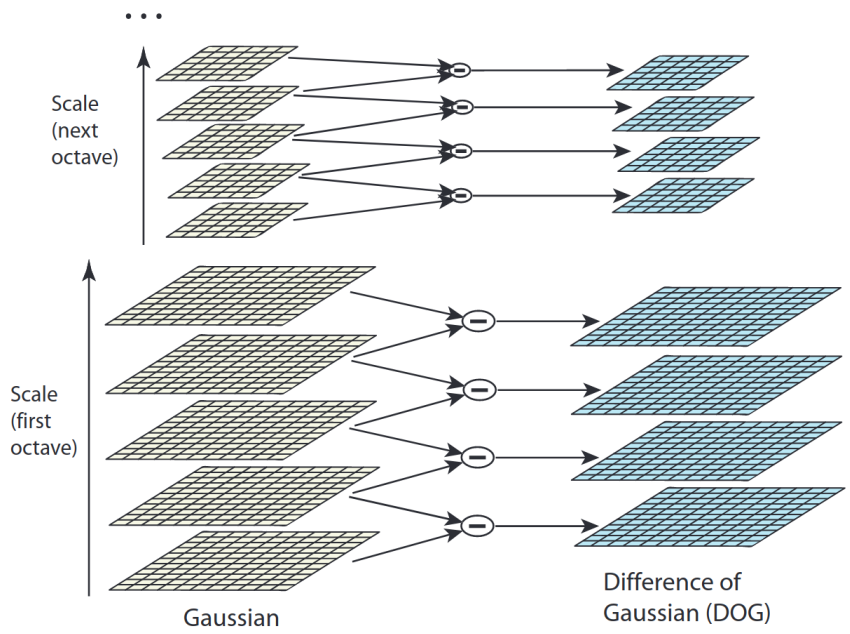

Figure 10: Scale space

The pseudo-code is shown in Algorithm 4.

Local Extrema Detection

Then, SIFT takes the local extrema of the difference of Gaussians. SIFT compare every pixel with the 26 neighboring pixels in a 3×3 cube, within the stack of DoGs. This is shown in Figure Notice that this requires the operated pixel to be not in the top or the bottom of one octave, and that's why SIFT needs $S+3$ Gaussians

Algorithm 4: Construction of the scale space

Input : original image I, the specific number of DoGs s, the number of octaves $onum$, the initial Gaussian coefficient σ_0

Output: A sequence of Difference of Gaussians, $DoG[][]$

```
S ← I;                                          /* the initial sample used for current octave */
σ ← σ0;
Initialize the Gaussian images L[][];
for o ← 0 to onum − 1 do
    for i ← 0 to s + 2 do
        if i = 0 then
            G[o][i] ← S(x, y);
        else
            σ ← 2^(1/s) σ;
            L[o][i] ← 1/(2πσ²) e^(−(x²+y²)/2σ²) ∗ I;        /* convolution with Gaussian */
        end if
        if 1 ≤ i ≤ s + 2 then
            DoG[o][i − 1] ← L[o][i] − L[o][i − 1];
        end if
    end for
end for
```

(i.e. $S+2$ DoGs) to create S useful layers of DoGs. For instance, in Figure 10, the first octave needs 5 Gaussians to create 4 DoGs, in which the middle 2 will be used.

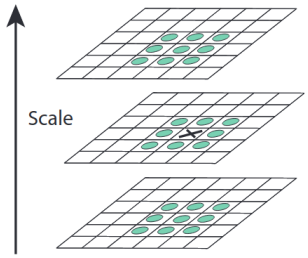

Figure 11: Select local extremum

Accurate Keypoint Localization

Taylor Estimation

The detected local extremum points are in the discrete space, and will be unstable when encountered with the edges and low contrast, which is not accurate enough. SIFT needs a more accurate method to find the keypoints in the original space. SIFT uses the Taylor expansion, at the sample points (i.e. detected extremum points in the last section), to mimic the continuous function:

$$D(\mathbf{x}) = D + \frac{\partial D^T}{\partial \mathbf{x}}\mathbf{x} + \frac{1}{2}\mathbf{x}^\mathbf{T}\frac{\partial^2 D}{\partial \mathbf{x}^2}\mathbf{x} \tag{0.0.6}$$

where D is the DoG function $D(x, y, \sigma)$ and $\mathbf{x} = (x, y, \sigma)^T$ is the offset from the sample point. [2] This is an estimate of the continuous function. Hence, the extremum point of this estimate

$$\hat{\mathbf{x}} = -\frac{\partial^2 D^{-1}}{\partial \mathbf{x}^2}\frac{\partial D}{\partial \mathbf{x}} \tag{0.0.7}$$

is the offset vector from the sample points to the expected extremum points. If the offset vector is too long (in [2], norm > 0.5), it means the extremum point is more closer to another sample point, then SIFT should choose that point to do Taylor expansion and offset calculation.

Reject Low Contrast

For the extremum vector $\hat{x}$, the DoG function is

$$D(\hat{\mathbf{x}}) = D + \frac{1}{2}\frac{\partial D^T}{\partial \mathbf{x}}\hat{\mathbf{x}} \tag{0.0.8}$$

if $|D(\hat{\mathbf{x}})| < 0.03$ [2] it is discarded due to low contrast.

Eliminating Edge Responses

Moreover, SIFT needs to eliminate the poorly defined extremas in difference of Gaussians when the keypoint is closed to edges.

SIFT uses the curvature to detect whether the keypoints are on some edge. The characteristics along the edge will remain almost the same, but that crossing the edge can be vary a lot.[2] Hence, in the keypoint near the edge will feature in a significantly great ratio of its two principle curvatures, which can be obtained by the ratio of the two eigenvalues of the Hessian matrix $\mathbf{H}$ at the keypoint. Given the Hessian

$$\mathbf{H} = \begin{pmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{pmatrix} \tag{0.0.9}$$

and denote the ratio of two eigenvalues α and β as $r = \alpha/\beta$, one have

$$\frac{\mathrm{Tr}(\mathbf{H})}{\mathrm{Det}(\mathbf{H})} = \frac{(\alpha+\beta)^2}{\alpha\beta} = \frac{(r+1)^2}{r} \tag{0.0.10}$$

This is a function takes minimum when $r = 1$, and becomes larger when r becomes larger or smaller. Hence one only needs to check if $\frac{\mathrm{Tr}(\mathbf{H})}{\mathrm{Det}(\mathbf{H})}$ exceeds some threshold to filter the edge values out.

Orientation Assignment

In order to assign orientation to the keypoints, SIFT uses a statistic method: endue the keypoint with the most dominant orientation of the all the Gaussian smooth points $L(x, y)$ with the closest scale σ. The determination of the orientation vector of $L(x, y)$ is using the discretization technique:

$$\mathbf{v} = \begin{pmatrix} L(x+1, y) - L(x-1, y) \\ L(x, y+1) - L(x, y-1) \end{pmatrix} \tag{0.0.11}$$

$$m(x, y) = \sqrt{(L(x+1, y) - L(x-1, y))^2 + (L(x, y+1) - L(x, y-1))^2} \tag{0.0.12}$$

$$\theta(x, y) = \tan^{-1}\frac{L(x, y+1) - L(x, y-1)}{L(x+1, y) - L(x-1, y)} \tag{0.0.13}$$

Then, the orientation vectors are put into 36 bins covering 360 degree, in order to form a histogram. When one vector is put into the bin, SIFT adds the magnitude of that vector, weighted by a Gaussian blur of 1.5σ [2]

$$\tilde{m}(x, y) = m(x, y) * G(x, y, 1.5\sigma) \tag{0.0.14}$$

SIFT takes the highest peak as the dominant direction, and remains the local peaks with 80% of the highest peak as auxiliary directions, which will be explained as other keypoints by SIFT. Finally, SIFT takes the 3

closest bins to the picked peak, and make a parabola fit. The procedure is shown in Figure 12 [3].

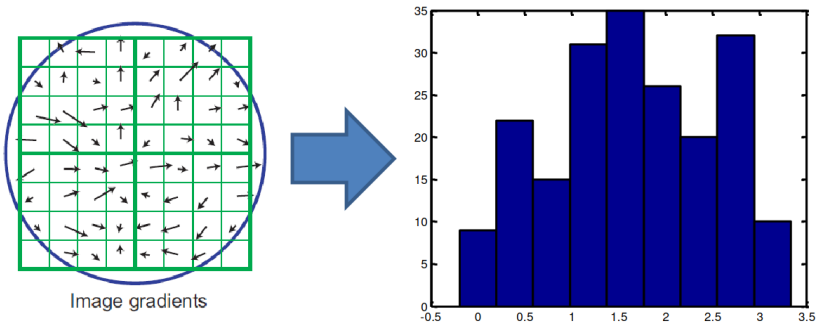

Figure 12: Orientation assignment

Local Image Descriptor

Finally SIFT expects to obtain the accurate descriptor of each of the keypoints. Consider a neighborhood region around the keypoint, in which lots of sample points lie. SIFT divides this region into 2×2 subregions. Now one expects to get a tuple of 8-directional vectors for each subregion and make the $2 \times 2 \times 8 = 32$ vectors as the descriptor of the keypoint.

First, in order to eliminate the influence of the direction of the keypoint's vector obtained from the section *Orientation Assignment*, one should rotate the neighborhood and make the keypoint's vector points to the x-axis, as shown in Figure 13.

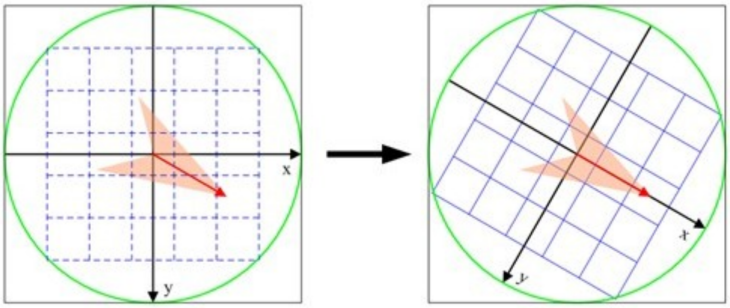

Figure 13: Rotation compensation

Second, use the method shown in equation (0.0.11), derive the direction and magnitude for every sample point. However, this time 0.5σ is taken [3], to get a Gaussian-weighted vector, endued with magnitude and direction, for every sample points in the neighborhood region.

Third, use **trilinear interpolation** to get the eight-direction vector for each subregion. This vector is represented as a histogram of 8 bins, standing for the 8 directions. Here, the three dimensions used for trilinear interpolation is x, y and direction (θ). This interpolation is not usual; it starts from the known points and keeps adding weighted value to the unknown "bins". For instance, in Figure 14, the red point is the sample point, the green grid is the boundary of subregions, and the blue grid is the auxiliary grid. The center of subregions are tranformed as vertices of blue grid. By trilinear interpolation, this red sample point will increment the subregion center $(2, 0), (2, 1), (3, 0)$ and $(3, 1)$, i.e. the four vertices of the blue sqaure in which the read sample point lie. The pseudo-code for this algorithm is shown in Algorithm 5.

Finally, after the normalization, the descriptor of the keypoint is extracted as the eight-direction vectors, and SIFT finishes all its work.

Algorithm 5: Trilinear interpolation for SIFT

Input: len, the length of each subregion; $a \times a$, number of subregions; in each subregion, $b \times b$ samples $samples$

Output: the histogram $hist[][][]$

Function `coeff`($index$, $sample$, len)**:**
    $x \leftarrow (0.5 + index)len$;
    $coeff \leftarrow 1 - |sample - x|/len$;
    return $coeff$;
end

Function `increment`($hist$, $indx$, $indy$, $inddeg$, $sample$, len, $lendeg$)**:**
    $hist[indx][indy][inddeg] \leftarrow hist[indx][indy][inddeg] + sample.mag \times$ `coeff`($indx$, $sample.x$, len) $\times$ `coeff`($indy$, $sample.y$, len) $\times$ `coeff`($inddeg$, $sample.deg$, $lendeg$)
end

foreach $sample \in samples$ **do**
    $lu.x \leftarrow \lfloor (sample.x + 0.5len)/len \rfloor - 1$; $lu.y \leftarrow \lfloor (sample.y + 0.5len)/len \rfloor - 1$;
    $ru.x \leftarrow lu.x + 1$; $ru.y \leftarrow lu.y$;
    $ld.x \leftarrow lu.x$; $ld.y \leftarrow lu.y + 1$;
    $rd.x \leftarrow lu.x + 1$; $rd.y \leftarrow rd.y + 1$;
    $ldeg \leftarrow \lfloor sample.\theta/45 \rfloor$; $udeg \leftarrow ldeg + 1$;
    foreach $region \in \{lu, ru, ld, rd\}$ **do**
        `increment`($hist$, $region.x$, $region.y$, $ldeg$, $sample$, len, 45);
        `increment`($hist$, $region.x$, $region.y$, $udeg$, $sample$, len, 45);
    end foreach
end foreach
return hist;

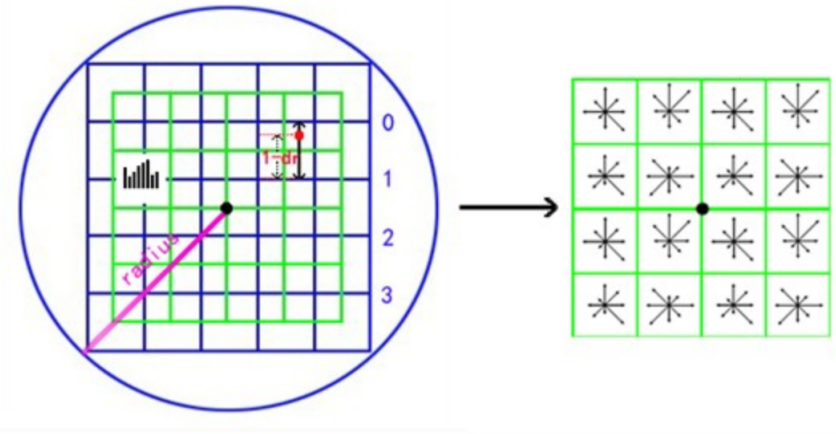

Figure 14: Trilinear interpolation

References.

[1] Tony Lindeberg. "Scale-space theory: A basic tool for analyzing structures at different scales". In: *Journal of applied statistics* 21.1-2 (1994), pp. 225–270 (cit. on p. 15).

[2] David G Lowe. "Distinctive image features from scale-invariant keypoints". In: *International journal of computer vision* 60.2 (2004), pp. 91–110 (cit. on pp. 14, 16, 17).

[3] Ajit Rajwade. *Scale Invariant Feature Transform (SIFT), CS763 - Computer Vision, Indian Institute of Technology Bombay*. 2014 (cit. on pp. 15, 18).