Tracking Objects Time Symmetrically

Sebastian Damrich
Heidelberg University
sebastian.damrich@gmx.de

Lukas Bold
Heidelberg University
lukasbold92@gmail.com

Abstract

The abstract paragraph should be indented ½ inch (3 picas) onboth the left- and right-hand margins. Use 10 point type, with a vertical spacing (leading) of 11 points. The word **Abstract** must be centered, bold, and in point size 12. Two line spaces precede the abstract. The abstract must be limited to one paragraph.

1 Tracking as a min-cost-flow problem

Out first strategy is to model the movement of the object in the video as a flow which is minimal with respect to some costs.

1.1 Min-Cost-Flow programs

Given a directed graph $G = (V, E)$ with edge weights c, edge capacities κ and vertex supply b, the min-cost-flow problem (MCFP) is the problem of finding a flow of minimal cost along the edges of the graph, which respects the capacity constraints and manages to match up the vertices with positive and negative flow supply. As a linear program this can be formulated as follows:

$$\begin{aligned} &\min_f \sum_{e\in E} c(e)\, f(e) \text{ s.t.} \\ \forall v \in V : &\sum_{(v,w)\in E} f(v,w) = \sum_{(u,v)\in E} f(u,v) + b(v) \\ \forall e \in E : &0 \leq f(e) \leq \kappa(e) \end{aligned} \tag{1}$$

where the first condition ensures that flow is conserved and the supply and demand at each vertex are met while the second condition guarantees the capacity constraints. In the following, we will often write c, b, κ, f as vectors of length $|E|$.

1.2 MCFP for object tracking

We construct the directed graph on which we want to compute a min-cost-flow as follows: The vertices are given by the pixels of the video in addition of an auxiliary node t, which serves as a sink. There are edges from a pixel to a square area of pixels below it in the next frame (pixel close to the borders of the frame have less outgoing edges). We call the length of the sides of this square window size. It should be an odd number. All the pixels in the last frame have an edge to the sink. The flow supply is one for every pixel in the first frame that belongs to the object. All other pixels have a flow supply of zero apart from t, which has a flow demand matching the cumulative flow supply.
The costs and capacities of the the graph are the output of a neural networks, which takes the video as input.
Given a min-cost-flow for the above described graph, we predict the location of the object in every frame by (a bounding box around) the pixels through which there is a positive flow. We can also associate a loss l to a flow. This is done subtracting the the amount of flow that goes through the

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

object in the last frame from the amount of flow that misses the object in the last frame. As a result, we could capture the change of the loss by the flow by the gradient:

$$\left(\frac{dl}{df}\right)_{(v,w)} = \begin{cases} 1 - 2 \cdot \mathbb{1}(\text{the pixel to } v \text{ is part of the object}) & \text{if } w = t \\ 0 & \text{else} \end{cases} \tag{2}$$

In order to train the neural network, we need to know $\frac{dl}{dc}$ and $\frac{dl}{d\kappa}$. Given $\frac{dl}{df}$ as above and using the chain rule, the key is to find $\frac{df}{dc}$ and $\frac{df}{d\kappa}$. In other words, we need to find out how the optimal flow, the solution to the linear program 1, changes with the problem parameter c and κ. For this task we explored the ideas of [AK17]

2 Special cases of the MCFP

We saw in the above section, that the approach of [AK17] seems not well suited for our problem. However, to train the neural network, we need to provide the network with some update direction for its output. Instead of differentiating through the optimisation problem to obtain a gradient, we use a solver to solve the version of the MCFP problem and then update the parameters of the problem in a way that reduces the loss at least heuristically at the solution. Then we updated parameters as targets for the neural network. In our analysis, we specialised the general MCFP to two corner cases. This reduces the computation time and highlights different features of costs and capacities. The general MCFP could also be treated in a similar way.

2.1 Shortest Paths

We explored the situation without upper capacity constraints in some detail. In our MCFP the pixel belonging to the object in the first frame all have a flow supply of one and the sink takes in all this flow. Therefore, without upper bounds on the capacities, our MCFP becomes the problem of finding the shortest paths from the pixels that constitute the object in the first frame to the sink. One advantage of this approach is that there a fast solvers for Shortest Paths problems, such as the Dijkstra (if all costs are non-negative) or the Bellman-Ford algorithm. [1]

(Note on perfromance and Message passing). These algorithms compute the shortest paths from all vertices to a given one. So by computing all the shortest paths to the sink, we have solved our version of the MCFP and obtain a prediction of the object's location in the last frame by the pixels through which the shortest paths starting at the object in the first frame pass. Those paths that do not go through object in the last frame, have lost the object at some point. Therefore, we penalise them by increasing the cost of the edges along such a path by some fixed penalty parameter (a penalty which decreases for edges between earlier frames). One the other hand, paths that go through the object in the last frame are rewarded. The costs along their edges is reduced by the (scaled) penalty parameter. The shortest paths algorithm gives the shortest paths from the background of the first picture to the sink for free. In order to increase the amount of information passed to the neural network, we also update the costs along the paths from the background to the sink. This is done in the same fashion as with the paths starting in the object, but in opposite direction: Paths from the background that pass through the background of the last frame are rewarded and those that pass through the object in the last frame are penalised.
One issue with this way of updating costs is that if a path that is penalised and one that is rewarded have a common end (they start in the object and the background respectively, but merge at some point), the rewards and penalties on the common end cancel. The possibility of the merging of paths itself could be an issue. If an edge has very small cost, it might induce several paths to use this edge and hence to merge. Without upper bounds on the capacities there is no possibility for two paths to separate once they have merged. So we might fear that many paths merge at some point and that as a result the last frame is only passed at very few pixels by the paths coming from the object. This would yield degenerated predictions of the objects location which only consist of a few pixels. Fortunately, this does not seem to be an issue, as we will see in section REF, where we discuss the implementation of this algorithm. We suspect that using a convolutional networks saves us.

MAYBE MENTION THE JUMPY BEHAVIOUR OF SHORTEST PATHS AND THE NON CONVERGENCE WITH A POINTER TO THE EXPERIMENTAL RESULTS

[1] Due to the special directed structure of our graph, finding shortest paths

2.2 Maximum Flow

The other extreme case of a MCFP would be to focus entirely on the capacities rather than the costs. By setting the costs to zero for all edges, the cost-minimisation problem becomes ill-defined. In order to dodge this problem, one could replace the cost-minimisation problem with a flow maximisation problem. Instead of defining a fixed flow supply for the nodes in the first frame and the sink, one could allow the pixels that constitute the object in the first frame to send as much flow as possible and the sink to take in as much flow as possible. For all other pixels flow-in equals flow-out still applies. The resulting linear program would look like this

$$\begin{aligned} &\max_f \sum_{(v,t)\in E} f(v,t) \text{ s.t.} \\ \forall v \in V \setminus (V' \cup \{t\}) \quad &\sum_{(v,w)\in E} f(v,w) = \sum_{(u,v)\in E} f(u,v) \\ &\forall e \in E : 0 \leq f(e) \leq \kappa(e) \end{aligned} \tag{3}$$

where V' are the pixels in the first frame that constitute the object. One advantage of such a simplification of the MCFP is that there are several fast algorithms for solving a maximum flow problem, for instance the Ford-Fulkerson (or Eddmund-Karp) algorithm or the Pre-Flow-Push algorithm. So as above, one could let a neural network predict capacities given the video, then find a maximum flow through the resulting network and finally update the capacities in a meaningful way to provide an update direction for the neural network. Similar to the case of shortest paths, one way to update the capacities is to increase the capacities along a path of positive flow to the object in the last frame and to decrease the capacities (bearing in mind the lower bound of zero capacity) along a path of positive flow to a background pixel in the last frame. A theoretical appeal of this strategy is that it nicely matches up with the notion of augmenting paths that are used to determine a maximum flow is determined with the Ford-Fulkerson algorithm.
Another advantage of using capacities rather than costs is that maximum flow behaves more continuously and piecewise even differentiably with regard to the capacities. That is to say that among the set of maximal flows there are some that change piecewise differentiably with respect to the capacities: Changing the capacity of an edge on which the flow is not maximal infinitesimally, does not change the flow at all. Decreasing the capacity of an edge on which the flow is maximal, reduces the flow on that edge with a derivative of one. Increasing the capacity on such an edge either increases the flow on that edge with derivative one (if the network allows more flow to enter and leave the edge) or it does not change the flow on that edge at all. The effect on other edges can be manifold, due to the non uniqueness of a maximal flow on some networks, however there always are maximal flows that change piecewise differentiably on all edges with respect to the edges. One could try to use this somewhat smooth behaviour to return a crude approximation to a derivative $\frac{dl}{dc}$ to the neural network. For that we use the loss from section 1.2, normalised it by the size of the flow and pretend that changing the capacity of an edge with maximal flow always results in a derivative of one:

$$\left(\frac{dl}{dc}\right)_{(v,w)} = \begin{cases} \frac{1-2\cdot\mathbb{1}(w \text{ is part of the object})}{\sum_{(u,t)} f(u,t)} & \text{if } w \text{ is in last frame and } f(v,w) = c(v,w) \\ 0 & \text{else} \end{cases}$$

However, this would only propagate information regarding the capacities of the last frame and it is unclear to us whether this is much better than omitting a maximum flow computation altogether and instead updating the capacities of edges leading towards the last frame depending on whether they lead to the object or not. This latter approach however seems to resemble an more and object detection scheme than an object tracking one and it is not timesymmetric at all.

3 Implementation

We have implemented the shortest paths approach.[2] For the neural network we drew inspiration from the architecture of [LZL+17]. The network is a convolutional network with a shallow U-form. A ReLu layer follows each convolutional layer. As described in [LZL+17], at first two 2D convolutions are applied. We use a kernel size equal to the window size of the graph. The 2D convolutions increase

[2] `https://github.com/sdamrich/FlowTracker`

the channel number from 3 over 5 to 10. Padding is used to keep the original dimensions. The result is recorded, let us call it X and a copy is 2D max pooled to decrease the dimensions of the frames by a factor of 2. Then, video is 3D convoluted to introduce that depend on the time structure of the video. The 3D convolutions keep the channel number constants. First, we use a kernel of dimensions (3, window size, window size). So, the convolution sees a square area of length window size in three consecutive frames. It should be noted, that this corresponds to information being convoluted that comes from an area of side length twice the window size as we have pooled before. We apply padding to keep the dimensions. Afterwards, we use a kernel of dimensions (2, window size, window size) with padding to keep the latter two dimensions. As a result, we reduce the time dimension by one. Then we upsample trilinearly keeping the time dimension but restoring width and height of the frames to the original values, let us call the result Y. Before adding the 2D information X and the 3D information Y, we convolute X in the time dimension with a kernel of size 2 to match up dimensions. In the last step, we apply a 2D convolution layer with a kernel of window size, which gives out window size^2 many channels. Here we do not apply a ReLu layer. We interpret the channels of the output as the costs for edges going out of a pixel. The code for this can be found in the NN.py file.

MSE LOSS!
The graph is built explicitly using the graph_tool package. Since we build a large graph, we chose this performant package. [3]

References

[AK17] Brandon Amos and J. Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. *CoRR*, abs/1703.00443, 2017.

[LZL+17] Kisuk Lee, Jonathan Zung, Peter Li, Viren Jain, and H. Sebastian Seung. Superhuman accuracy on the SNEMI3D connectomics challenge. *CoRR*, abs/1706.00120, 2017.

[3] `https://graph-tool.skewed.de/performance`