DETR Model Loss

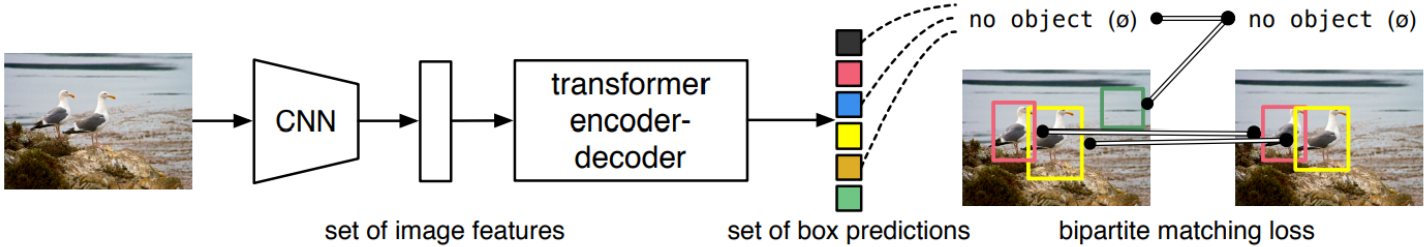
Section 3.1 in End-to-End Object Detection with Transformers

June 15, 2020

Paper overview

The point

Identify bounding boxes and category labels directly by combining a common CNN with a transformer architecture



What is the loss function trying to catch?

- Near-duplicates should be collapsed to single identification
- Loss function should not change if the predictions are rearranged

Section 3.1 Object detection set prediction loss

Setting:

- N is the number of predictions + an unspecified number of $\varnothing$-classified boxes that are determined to not be objects.
- y = ground truth size N set of objects including multiple $\varnothing$
- $\hat{y} = \{\hat{y}_i\}_{i=1}^{N}$ are the predicted objects
- We seek the lowest cost bipartite matching between the two sets
- We define the matching by applying the permutation σ to the elements in $\hat{y}$ so

$$\begin{aligned} \hat{y}_{\sigma(1)} &\leftrightarrow y_1, \\ \hat{y}_{\sigma(2)} &\leftrightarrow y_2, \\ &\dots \\ \hat{y}_{\sigma(N)} &\leftrightarrow y_N \end{aligned}$$

Pairwise match setting

Each ground truth element is $y_i = (c_i, b_i)$ where

- c_i is the target class label (a specified object or simply $\varnothing$)
- $b_i \in [0, 1]^4$ is the ground truth box
  - center coordinates
  - percentage height and width wrt image

And for each element $\hat{y}_{\sigma(i)}$

- We find the probability that the contained object is in class i: $\hat{p}_{\sigma(i)}(c_i)$
- The box is described by $\hat{b}_{\sigma(i)}$

Loss function for bounding boxes

$$\mathcal{L}_{\text{box}}(b_i, \hat{b}_{\sigma(i)}) = \lambda_{\text{IoU}}\mathcal{L}_{\text{IoU}}(b_i, \hat{b}_{\sigma(i)}) + \lambda_{\text{L1}}\|b_i - \hat{b}_{\sigma(i)}\|_1$$

Notes:

- IoU is the intersection over union which is a scale-invariant area comparison for bounding boxes.

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|} = \frac{|I|}{|U|}$$

  - There's a variant that introduces the relationship to the smallest convex hull containing A and B.
  - Either way, the loss is $\mathcal{L}_{IoU} = 1 - \text{IoU}$.
- The λs here are hyperparameters, and the two losses are normalized by the number of objects in the batch.
- The L1 norm term is just the sum of the absolute errors, so we're basically balancing between relative and absolute errors in this loss function.

Pairwise loss between one truth y_i and its pair $\hat{y}_{\sigma(i)}$

$$\mathcal{L}_{\text{match}}(y_i, \hat{y}_{\sigma(i)}) = -\mathbb{1}_{\{c_i \neq \varnothing\}} \hat{p}_{\sigma(i)} + \mathbb{1}_{\{c_i \neq \varnothing\}} \mathcal{L}_{\text{box}}(b_i, \hat{b}_{\sigma(i)}) \tag{1a}$$

We want to reward high classification probabilities and penalize large box differences. Loss doesn't consider $\varnothing$ predictions, so we're creating bijection with actual classification/box instances.

This gives us a framework in which we have a complete bipartite graph containing our predictions on the left and ground truth on the right, with loss-weighted edges. I think the edges headed to $\varnothing$-classified ground truth nodes would just be weighted with 0. The trick, as we'll see, is minimizing the true costs.

Loss – finding the best matching

Recall our framework in which we seek the best permutation σ defining the bipartite matching.

$$\hat{y}_{\sigma(1)} \leftrightarrow y_1,$$
$$\hat{y}_{\sigma(2)} \leftrightarrow y_2,$$
$$\ldots$$
$$\hat{y}_{\sigma(N)} \leftrightarrow y_N$$

We seek the permutation $\hat{\sigma}$ with minimum total loss:

$$\hat{\sigma} = \arg\min_{\sigma} \sum_{i=1}^{N} \mathcal{L}_{\text{match}}(y_i, \hat{y}_{\sigma(i)}) \tag{1}$$

where $\mathcal{L}_{\text{match}}(y_i, \hat{y}_{\sigma(i)})$ is the pair-wise matching cost between ground truth and prediction. The optimal cost is determined by the Hungarian Algorithm (jump ahead a few slides to understand).

Hungarian loss function for the best match

Once we've found the optimal bipartite matching, the loss combines a negative log-likelihood formulation for the class prediction with the loss for the box differences

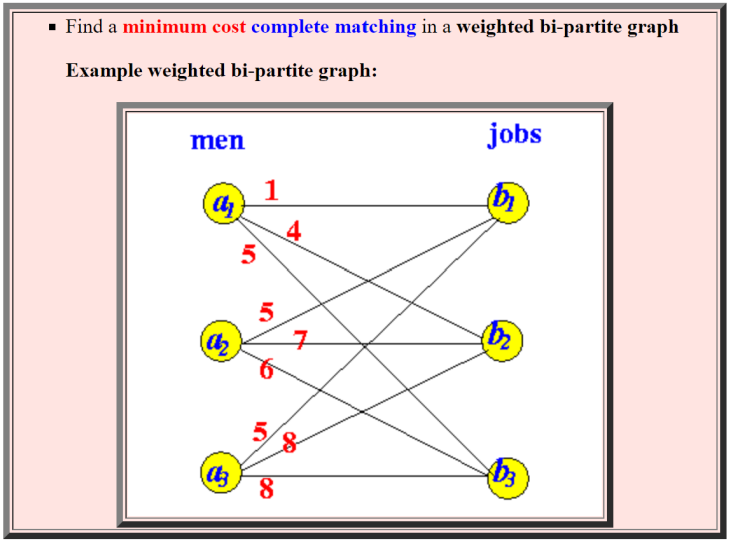
$$\mathcal{L}_{\text{Hung}}(y, \hat{y}) = \sum_{i=1}^{N} \left[-s(c_i) \log \hat{p}_{\hat{\sigma}(i)}(c_i) + \mathbb{1}_{\{c_i \neq \varnothing\}} \mathcal{L}_{\text{box}}(b_i, \hat{b}_{\sigma(i)}) \right] \quad (2)$$

where $s(c_i) = 0.1$ if $c_i = \varnothing$.
While they've formulated this using log-likeihood, the effect is that you've taken a number between 0 and 1 and made it between $-\infty$ and 0, and in doing so the negative no longer makes sense to me. I'm not sure if they really kept the negative. They said in the matching step the magnitudes of the original probabilities produced better results.

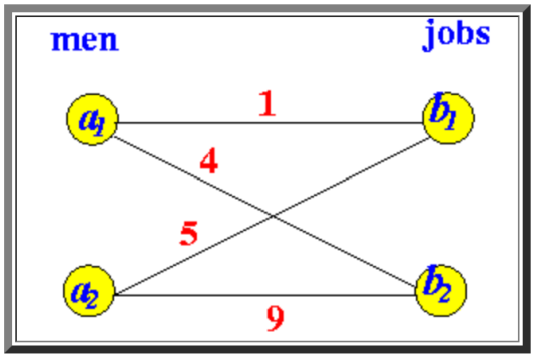
What is the Hungarian algorithm?

- You have n people and n tasks, and an $n \times n$ cost matrix that describes how much it will cost you if person i completes task j? What's the min cost to complete all tasks?
- View as a complete bipartite graph with n worker vertices, n job vertices, and cost $c(i, j)$-weighted edges.

Here's a picture from Professor Cheung's pages at Emory:



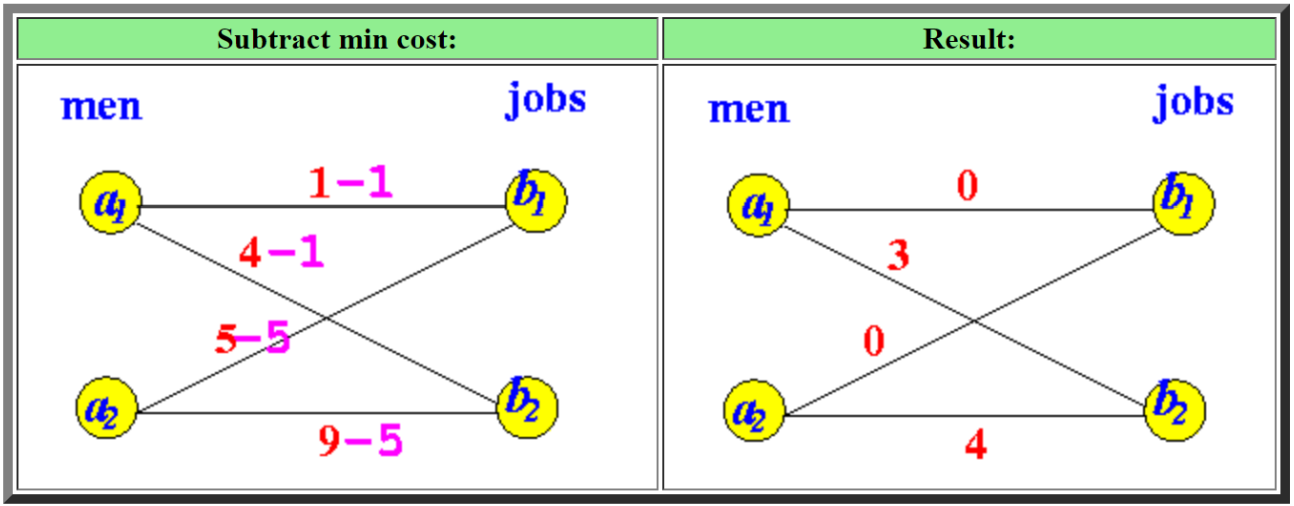
Why finding this minimum cost matching is hard



- Notice that 1 is the least-cost link. Should it be in the solution?
- Well, if we pick 1, then we also have to pick 9, and we get the wrong solution.
- Lesson: **Every lost opportunity carries a cost**

A start at calculating the cost of lost opportunity

Compute the **additional incurred cost** when using **non-optimal edged** by **subtracting** the **minimum cost** from the other edges that is **incident** to that **same node**:



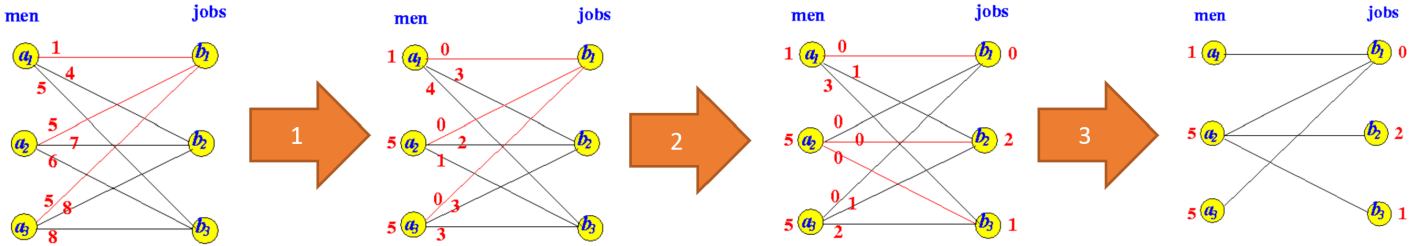
After subtracting the min cost edge, the 0 weight edges would be the best choice, but we can't pick both 0s here! Nonzero edge meaning:

- It will cost an extra \$3 over the best edge to match a_1 with b_2
- It will cost an extra \$4 over the best edge to match a_2 with b_2

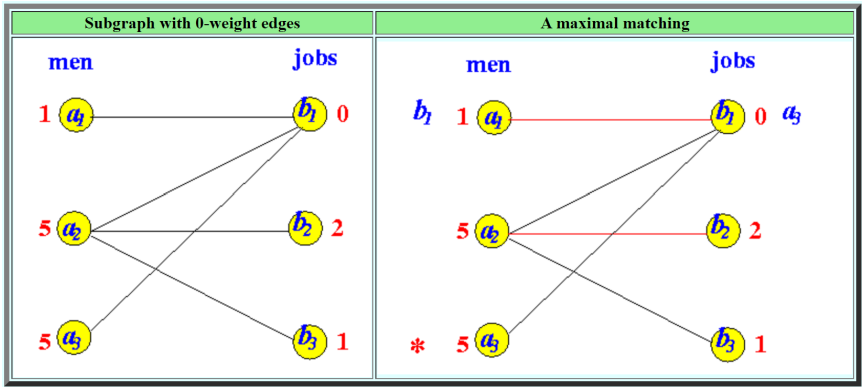
So this tells us the best solution is to match a_1 with b_2

The Hungarian Algorithm – getting started

1. For each of the n workers, find the minimal cost edge and subtract that weight (cost) from all weights *connected to that one vertex.* Note: at least one edge will necessarily have 0 weight
2. Label each of the n jobs with adjusted weight of the smallest entering edge. Subtract that label value all incoming edge weights to create new weights.
3. Consider the zero-weight subgraph. The edges in the subgraph will provide the lowest possible cost if we can find a maximum matching (but we can't match all nodes, right?)



Apply the max-flow algorithm

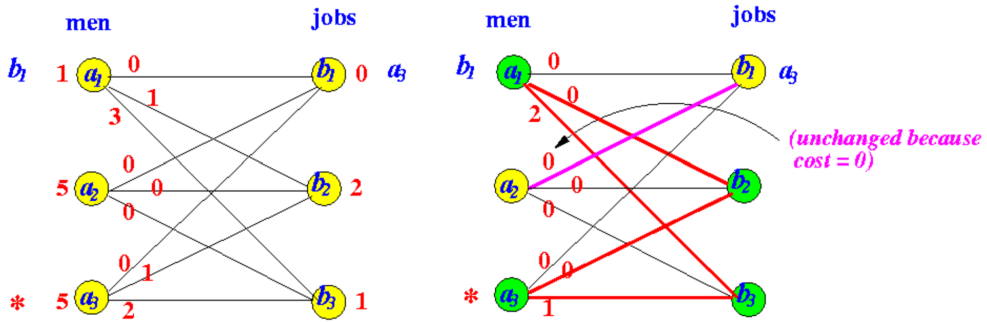


Alternating path algorithm for maximal matching (Sort of)

- First connect what you can (red)
- Label the first LSH vertex that doesn't have a friend with *
- Label all RHS vertices that are incident with *'s name
- For any labeled RHS vertex, label incident LHS vertices with RHS vertex's name
- Continue playing the game back and forth until everybody has a label or there are no more vertices you can reach

Iterative step in Hungarian alg when can't match all nodes

Add your labels to the step 3 graph that had two sets of weights subtracted:



Identify all positive cost edges – call the minimum positive cost δ. (Here $\delta = 1$)

For positive cost edges only:

- Subtract δ from the cost of any labeled LHS $\rightarrow$ **un**labeled RHS
- Add δ to the cost of any **un**labeled LHS $\rightarrow$ labeled RHS
- Make a new zero-weight graph
  1. If you can create a matching you're done – go to the original weights to calculate costs if there's more than one matching
  2. If you can't then you iterate starting over from this graph

Play problem solution