CLEANING UP INCORRECTLY MISLABELED DATA

* (DL) algorithms are
  - quite robust to random errors
  - less robust to systemic errors

  in the training set

* is it worth spending time in reducing the number of mislabeled images?

  It depends! You should look at →

overall dev set error	10%	2%
error due to incorrect labels	0.6%	0.6%
error due to other causes	9.4%	1.4%

  - 9.4% → FIRST YOU SHOULD TACKLE YOUR OVERALL DEV SET ERROR
  - 1.4% → ONCE YOU SOLVE YOUR DEV SET AND OTHER CAUSES ERRORS, YOU SHOULD TACKLE THE INCORRECT LABELS ISSUE.

* correcting incorrect dev/test set examples:
  - (#1) apply same process to the dev and test sets (THEY NEED TO COME FROM THE SAME DISTRIBUTION)
  - (#2) examine examples your algorithm got right as well as those it got wrong.
  - (#3) train and dev/test set may now come from slightly (≠) distributions. (NOT NECESSARILY BAD)

How to start prioritizing and see where to go next?

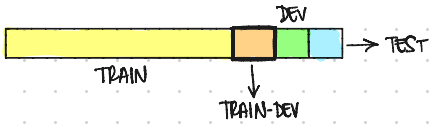
BUILD YOUR SYSTEM QUICKLY, THEN ITERATE!

MISMATCHED TRAINING AND DEV/TEST DATA

* training and testing on (≠) distributions:
  - (#1) OPTION 1: combine both datasets, shuffle them, and split it into train/dev/test
    advantage: same distribution / disadvantage: most data will come from one of the distributions.
  - (#2) OPTION 2: training set remains the same and dev/test sets will all come from a (≠) distribution
    advantage: you are aiming the target where you want it to be / disadvantage: train (≠) dev/test
  - (#3) OPTION 3: add 50% of new distribution to training set and dev/test have all the remaining

BIAS AND VARIANCE WITH MISMATCHED DATA DISTRIBUTIONS

* training-dev set ⟶ same distribution as training set, but not used for training



Human error	0%	0%	0%	0%
Train error	1%	1%	0%	10%
Train-dev error	9%	1.5%	11%	11%
Dev error	10%	10%	12%	22%
	VARIANCE PROBLEM (not generalizing well)	MISMATCH PROBLEM (⊕ distributions)	AVOIDABLE BIAS	AVOIDABLE BIAS + DATA MISMATCH

ADRESSING DATA MISMATCH

#1 Carry out manual error analysis to try to understand the difference between training and dev/test

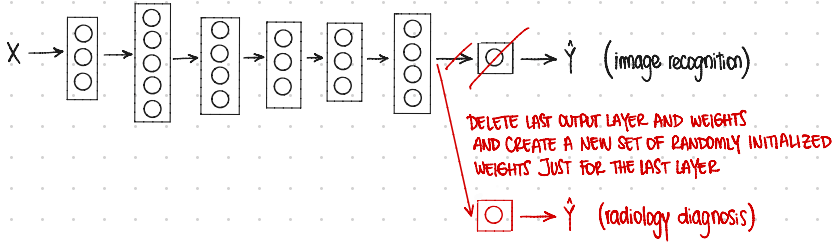
#2 Make training data more similar or collect more data similar to dev/test set

* you can also do artificial data synthesis (but you might overfit)

TRANSFER LEARNING

* take the knowledge a NN learned from one task and apply that knowledge to a ⊕ task.

* you can train a NN to recognize cats and then use that knowledge to help you do a better job reading X-ray scans.

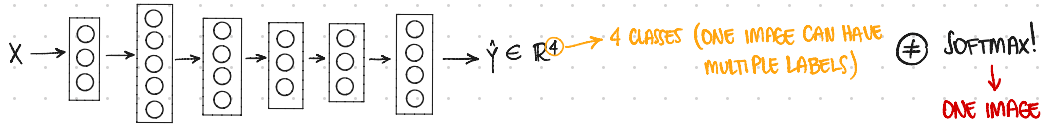


* you swap the input of the NN and use a new one. So the NN has been pre-trained and you may fine-tune it.

* when does transfer learning make sense? ⟶ when you have a lot of data from the problem you are transferring from and relatively less data for the problem you are transferring to.

MULTI-TASK LEARNING

* you start off simultaneously trying to have one (NN) do several things at the same time. Each of these tasks helps hopefully the other tasks.



$$J: \underset{(4,1)}{y^{(i)}} = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4}\mathcal{L}(\hat{y}_j^{(i)}, y_j^{(i)}) = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4}\left[-y_j^{(i)}\log\hat{y}_j^{(i)} - (1-y_j^{(i)})\log(1-\hat{y}_j^{(i)})\right]$$

LOGISTIC LOSS

YOU SUM OVER j=4 (MULTIPLE LABELS)

* if you train a (NN) minimizing this cost function, then you are solving multiple problems at once. Since you are doing a multi-label classification, then you are saving time compared to having to train one (NN) for each type of label you have.

* when does transfer learning make sense?

  (#1) training on a set of tasks that could benefit from having shared lower-level features.

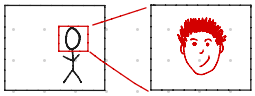
  (#2) amount of data you have for each task is quite similar (NOT ALWAYS TRUE THO)

  (#3) if you can train a big enough (NN) to do well on all tasks. The alternative is to train a (NN) for each task.

END-TO-END DEEP LEARNING

* you may need a lot of data before you reach a decent accuracy.

* used in face recognition and machine translation.



* instead of just take a whole image as input, the (NN) breaks it down into different steps to focus on the object you want to classify.

* why does it work?

  (#1) the tasks are much simpler to solve when broken down into (≠) tasks.

  (#2) there is usually a lot of data for all sub-tasks.

PROS	CONS
* it lets the data speak * less hand-designing of components needed	* may need large amounts of data * excludes potentially useful hand-designed components