## General (your) Ideas to improve your detector:

* Get more data
* Collect a more diverse training sets (various possible ways).
* Train the algorithm longer, by running more gradient descent iterations.
* Try a bigger neural network, with more layers/hidden units/parameters.
* Try a smaller neural network.
* Try adding regularization (such as L2 regularization).
* Change the neural network architecture (activation function, number of hidden units, etc.)

# What Al is popular now:

* Data availability
* Computational scale: We started just a few years ago to be able to train neural networks that are big enough to take advantage of the huge datasets we now have.

# Why NN:

* Even if we have more data the performance of older learning algorithms, such as logistic regression, “plateaus.(there will be little change)”
* If you train a small neutral network (NN) (small NN: neural network with only a small number of hidden units/layers/parameters )on the same supervised learning task, you might get slightly better performance:
* With larger neural networks(more number of hidden units/layers/parameters), you can obtain even better performance

One of the more reliable ways to improve an algorithm’s performance today is still to (i) train a bigger network and (ii) get more data.

Note: if we have very less data then NN or normal ML algorithms doesn’t differ much but if we have large dataset then NN will give better outcome.

We usually define:

* Training set​ — Which you run your learning algorithm on.

• Dev (development) set​ — Which you use to tune parameters, select features, and make other decisions regarding the learning algorithm. Sometimes also called the hold-out cross validation set​.

* Test set​ — which you use to evaluate the performance of the algorithm, but not to make any decisions regarding what learning algorithm or parameters to use.

Once you define a dev set (development set) and test set, your team will try a lot of ideas, such as different learning algorithm parameters, to see what works best. The dev and test sets allow your team to quickly see how well your algorithm is doing.

In other words, ​the purpose of the dev and test sets are to direct your team toward the most important changes to make to the machine learning system​.

So, you should do the following:

* Choose dev and test sets to reflect data you expect to get in the future and want to do well on.
* Collect the data while performing and test before sending to prod
* Randomly pic different kinds of data that is not trained and check the accuracy
* Once you define the dev and test sets, your team will be focused on improving dev set performance. performance should be improved for the data that collected from different ways
* The test set is not necessarily harder, but just different, from the dev set. So what works well on the dev set just does not work well on the test set. In this case, a lot of your work to improve dev set performance might be wasted effort

# Imbalance or Unbalanced data:

A classification data set with skewed class proportions is called [**imbalanced**](https://developers.google.com/machine-learning/glossary/#class_imbalanced_data_set). Classes that make up a large proportion of the data set are called [**majority classes**](https://developers.google.com/machine-learning/glossary/#majority_class). Those that make up a smaller proportion are [**minority classes**](https://developers.google.com/machine-learning/glossary/#minority_class).

What counts as imbalanced? The answer could range from mild to extreme, as the table below shows.

| Degree of imbalance | Proportion of Minority Class |
| --- | --- |
| Mild | 20-40% of the data set |
| Moderate | 1-20% of the data set |
| Extreme | <1% of the data set |

# DownSmapling:

 means training on a disproportionately low subset of the majority class examples.

Upweighting:

 means adding an example weight to the downsampled class equal to the factor by which you downsampled.

The weight should be equal to the factor you used to downsample:

 {example weight} = {original example weight} × {downsampling factor}

# Why Downsample and Upweight?

It may seem odd to add example weights after downsampling. We were trying to make our model improve on the minority class -- why would we upweight the majority? These are the resulting changes:

* **Faster convergence**: During training, we see the minority class more often, which will help the model converge faster.
* **Disk space**: By consolidating the majority class into fewer examples with larger weights, we spend less disk space storing them. This savings allows more disk space for the minority class, so we can collect a greater number and a wider range of examples from that class.
* **Calibration**: Upweighting ensures our model is still calibrated; the outputs can still be interpreted as probabilities.