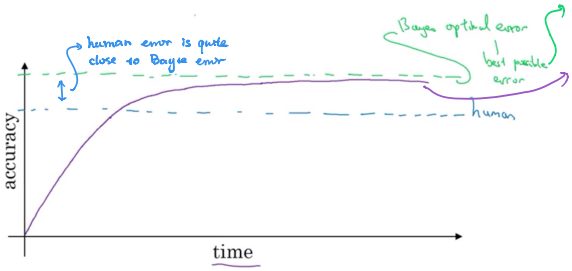
This lesson introduces some strategies that we can try on our machine learning projects. One of the challenges with building machine learning systems is that there are so many things we could change, e.g. hyperparameters tuning. To get the desired results, it's important to apply Orthogonalization Principle, which means that when we tune one thing, it only has one effect on the results without affecting other things. For a supervised machine learning system, here is a map of the process of machine learning and what to tune in order to achieve them one at a time:

Desired Effects	What to Tune / knobs
Fit training set well on cost function (well → comparable to human performance / low bias)	- bigger network - better algorithms (Adam etc.) - NN architecture/hyperparameters search
↓ then Fit dev set well on cost function (well → low variance)	- regularization (early stopping is semi-orthogonal because it also affects training set performance) - bigger training set - NN architecture/hyperparameters search
↓ then Fit test set well on cost function (well → high confidence)	- bigger dev set
↓ then Perform well in real world (well → satisfied users)	- change dev set or cost function.

- **Comparing to human-level performance.**

First we the training set to fit well and we can tell whether it's fitting well by comparing it to human performance. But why? First is that many machine learning algorithms has become competitive with human-level performance. (so it's comparable) Second is that designing and building a machine learning system is much more efficient when doing some task that human can also do. Third is human error is quite close to Bayes Optimal error or Bayer error for short, so achieving human error is good enough:



maybe the picture is too blurry, the audio clip is too noisy, the perfect level is not 100%. Bayes error can never be surpassed.

machine learning progress is quite fast until it surpasses human-level performance, and slows down after it surpasses:

- Humans are quite good at certain tasks (human error is not quite far from Bayes error) ⇒ There isn't much room to improve by the time the ML surpasses human-level performance.
- So long as ML is worse than humans, there are couples of things we can try to improve:
  - get human labeled data
  - better analysis of bias/variance.
  - get insights from manual error analysis

note:
Definition of human error depends on what we have access to.
Usually we use the best possible ones as human error, e.g.
a group of professionals labeled data, to get a proxy for Bayes error

Andrew Ng

Now let's discuss how we can perform better bias/variance analysis. Here is a simple example:

Human error	1%	→ 7% bias	Human error	7.5%	→ 0.5% bias
Training error	8%	⇒ reduce bias	Training error	8%	⇒ reduce variance.
Dev error	10%	→ 2% variance	Dev error	10%	→ 2% variance.

In short,

Human error, Training error → avoidable bias ⇒
- if avoidable bias is bigger ⇒ bias reduction tactics
- if variance is bigger ⇒ variance reduction tactics.

Dev error → variance

Even though humans are quite good at natural perception tasks, there are also case where ML surpasses human-level performance, especially in dealings with structure data because of the vast amount of data that computers can process.

Problems where ML significantly surpasses human-level performance

- Online advertising
- Product recommendations
- Logistics (predicting transit time)
- Loan approvals

Structured data
Not natural perception
Lots of data

- Speech recognition
- Some image recognition
- Medical
  - ECG, Skin cancer, ...

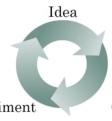
Natural data

Andrew Ng

- **Setting Up Targets** (decided by dev/test sets and metrics)

Single number evaluation metric: it tells us quickly if the new things we just tried is working better or worse than the last idea.

Here is an example of how to apply it:



Classifier	Precision	Recall
A	95%	90%
B	98%	85%

% of examples being recognized as cats, % of being correct

% of all cat images, % images being correctly recognized

→ Instead of using 2 numbers, we can use a single value called F1 score:

$$\frac{2}{\frac{1}{P}+\frac{1}{R}}$$ (harmonic mean of P and R)

P: precision, R: recall

Andrew Ng

Having well-defined dev set, with which we're measuring precision and recall, plus a single number evaluation metric, allows us to quickly tell which classifier is better among all of them, and thus speed up the iterative process of improving the machine learning algorithm

Satisficing and optimizing metric: we can satisfice something as long as they achieve the minimum requirement and optimize the other one:

For example:

Classifier	Accuracy	Running time
A	90%	80ms
B	92%	95ms
C	95%	1,500ms

could be F1 score

cost = accuracy − 0.5 × running time

→ instead of using single number evaluation metric, which seems too artificial, we could try satisficing and optimizing metric:

maximize accuracy → optimizing

subject to running time ≤ 100 ms → satisficing

Andrew Ng

Set up training/dev/test set: ideal way of setting up training/dev/test sets can greatly improve efficiency of finding the best model.

For example, we have data from different region:

- US
- UK
- Europe
- South America
- India
- China
- Australia
- Other Asia

what NOT to do:

Dev: US, UK, Europe, South America

Test: India, China, Australia, Other Asia

What should be done:

Randomly shuffle into dev/test set

Guideline:
- choose a dev set and test set to reflect data we expect to get in the future and consider important to do well on.
- dev set and test set should come from the same distribution (but not necessarily for train/dev set)
- 98:1:1 split for large data set (e.g. 1 million) or big enough for the purpose of dev/test set (e.g. 10,000)

When to change dev/test sets and metrics:

- when some mistakes are more severe than the others (e.g. incorrectly recognizing pornographic pictures as cat pictures, incorrectly recognizing important emails as spams, etc.) ⇒ put more penalties on these mistakes, e.g. $\text{cost} = \frac{1}{\sum_i w^{(i)}} \sum_{i=1}^{m_{dev}} w^{(i)} \mathbb{I}\{y^{(i)} \neq \hat{y}^{(i)}\}$ where $w^{(i)} = \begin{cases} 1 & \text{if } y^{(i)} \text{ is non-porn} \\ 10 & \text{if } y^{(i)} \text{ is porn} \end{cases}$
- If the model is doing well on our metric + dev/test set but does not correspond to doing well on the application, e.g. high solution images being trained but users upload low quality pictures
- new data is given (e.g. new species, new regions etc.); new requirements (e.g. higher false negatives etc.)

- **Error analysis**

Recall that if our ML project is performing worse than human-level performance, one of the techniques we could try is error analysis, which means manually examining mistakes that the algorithm is making. Here is an example:

Suppose we have found out that the misrecognized pictures are either of dog pictures, great cat (lions, tigers, etc.) pictures, blurry pictures, and pictures with fancy Instagram filters. There are a few directions that we could work on:

- fix pictures of dogs being recognized as cats
- fix great cats (lions, tigers etc.) being misrecognized
- improve performance on blurry images
- improve performance on Instagram filters

But before spending months on one of these only to find out it doesn't improve accuracy that much, let's find out how worthwhile it might be to work on each of these. We could perform it in the form of spreadsheet:

Misclassified Image	Dog	Great Cats	Blurry	Ins filters	Comments
1	✓			✓	Pitbull
2			✓	✓	Fancy filters
3		✓	✓		Rainy day at zoo
⋮	⋮	⋮	⋮	⋮	⋮
% of total	8%	43%	61%	12%	
Suppose we have error 10%, after fixing it we could improve to:	9.2%	6.7%	3.9%	8.8%	

→ This tells us that the ceiling in terms of how much we could improve performance is much higher for great cat pictures and blurry pictures.

As we're doing error analysis, sometimes we notice that some of the examples in the dev sets are mislabeled. Here's what we should do:

- If the deep learning algorithms are quite robust to random errors in the training, i.e. the labeler just wasn't paying attention and hit the wrong button accidentally, then it's probably alright to leave the errors and not spend too much time on fixing them.
- There is one caveat to this, when the algorithms are less robust to systematic errors, e.g. the labeler keeps labeling white dogs as cats.

If we're worried about the impact of incorrectly labeled examples on dev/test sets, we could add an extra column to count the number of examples where Y was incorrect during error analysis:

Misclassified Image	Dog	Great Cats	Blurry	Ins filters	Incorrectly labeled	Comments
1	✓			✓		
2			✓	✓		
3					✓	Labeler missed cats in the background drawing of a cat
					✓	
⋮	⋮	⋮	⋮	⋮	⋮	⋮
% of total	8%	43%	61%	12%	6%	
Suppose we have error 10%, how much each accounts for it:	0.8%	4.3%	6.1%	1.2%	0.6%	→ maybe not worthwhile to fix it since it's much smaller comparing to other mistakes (i.e. 10% − 0.6% = 9.4%)
Suppose we have now brought down the error to 2%, then:	—	—	—	—	0.6%	↓ since we haven't fixed the mislabeling, so it wouldn't change → now it's much bigger comparing to other mistakes (i.e. 2% − 0.6% = 1.4%)

we wouldn't know these until we do another error analysis

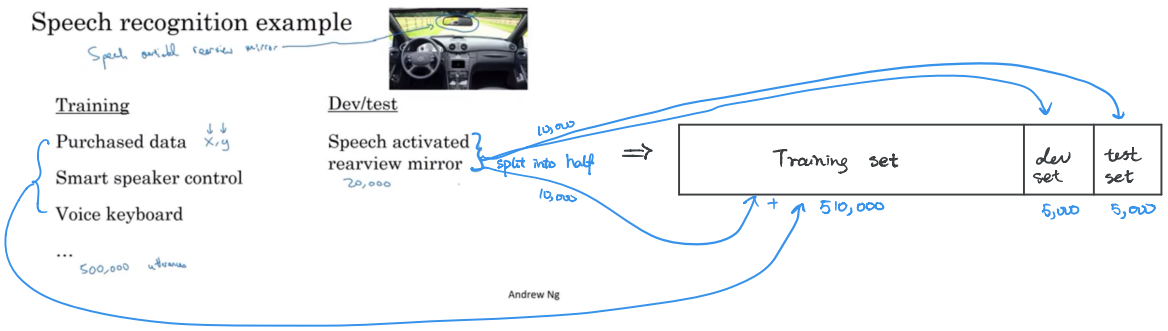
When correcting mislabeled examples, we should also notice:

- apply same process to dev/test sets to make sure that they continue to come from the same distribution
- consider examining examples the algorithms got right as well as wrong.
- train and dev/test may now come from slightly different distributions.

• Mismatched training and dev/test sets

Very often we find that training and dev/test sets might have different distribution. This is alright because the training set wouldn't affect the target that we want to aim at. Instead dev/test sets performance is what we really care about, so we should put data that we want to fit better on and data that we expect to get in the future in dev/test set.

Here is an example:



Since the distribution of training and dev/test set is different, the training error and dev/test error could be very different, but it doesn't neccessarily mean there is a large variance problem. It could one of the following:

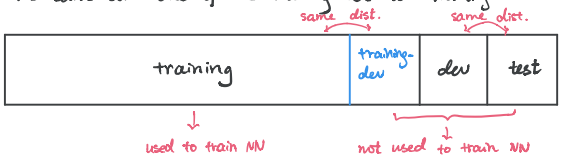
- the algorithm doesn't generalize well enough to data that it hasn't seen before → variance problem
- the distribution of data in dev set is different

In order to tease out these two effects, it will be useful to define a new piece of data, known as the training-dev set, which has the same distribution as training set, but is not used for training. Here's an illustration.

Suppose we have data like this:

training error: 1%
dev error: 10%

Now we carve out some of the training set to training-dev set so that they have the same distribution.



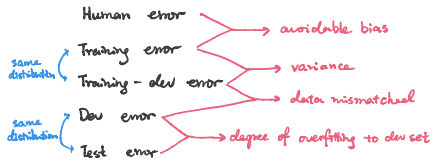
Example 1: variance problem
training error: 1%
dev error: 10%
training-dev error: 9%
→ even for data with the same distribution, the algorithm still doesn't generalise well

Example 2: data mismatched
training error: 1%
dev error: 10%
training-dev error: 1.5%
→ the algorithm works well on data from the same distribution but not on the distribution that we most care about (i.e. the dev set)

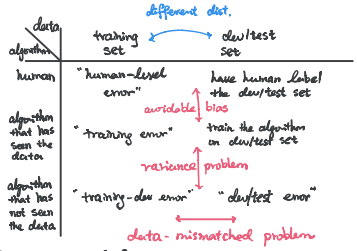
Example 3: bias problem
human error: 0%
training error: 10%
dev error: 12%
training-dev: 11%
→ avoidable bias problem

Example 4: bias problem + data mismatched
human error: 0%
→ avoidable bias problem
training error: 10%
dev error: 20%
→ data mismatched problem
training-dev error: 11%

General principle: by looking at human error, training error, training-dev error and dev error, it tells us roughly what kind of problem the algorithm has:



represented in a matrix



If there is a data-mismatched problem, then how do we address it? Unfortunately, there aren't completely systematic solutions to this, but there are a few things we can try:

- Carry out manual error analysis to try to understand difference between training and dev/test sets
  - e.g. in a speech-activated rear-view mirror application, listen to examples in dev set → maybe lots of car noise, more navigational queries with street numbers etc.
- Make training data more similar by using artificial data synthesis; or collect more data similar to dev/test sets
  - e.g. following the same example, we could simulate noisy in-car data, or get more data of people speaking out numbers
  - → how to simulate noisy in-car data:
  - artificial data synthesis example: "The quick brown fox jumps over the lazy dog."
    + car noise
    = synthesized in-car audio
  - → in particular, if we have way more audio clips than car noises, the algorithm might over-fit the car noises even though human ears cannot tell the difference

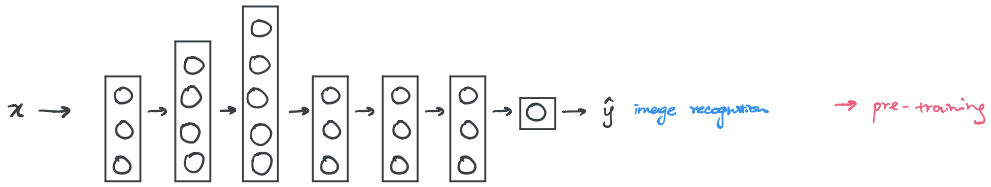
- **Learning from multiple tasks**

Sometimes the knowledge that an algorithm has learned from one task can be applied to do a seperate task. For example, we could have a NN learn to recognize cats and then use that knowledge or part of that knowledge to help us do

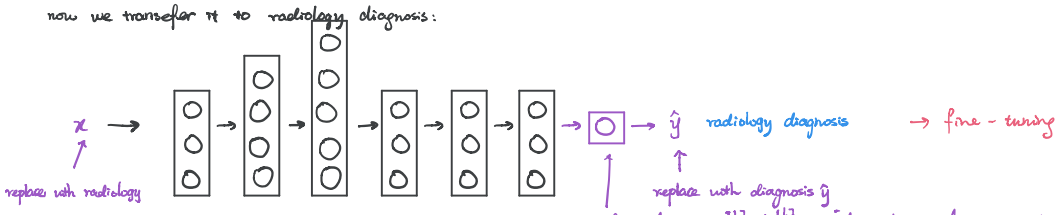
a better job in x-ray scans reading (radiology diagnosis), also known as transfer learning.

How transfer learning works:

Suppose we have a NN for image recognition:



now we transfer it to radiology diagnosis:



remove the last layer $w^{[L]}$, $b^{[L]}$, initialize them and retrain the NN on this new dataset (how we retrain the NN depends on how much data we have. If we have enough data, we can also retrain the entire NN; otherwise we could just retrain the last one or two layer and keep the other layers fixed.)

These processes are often known as pre-training (image recognition) and fine-tuning (radiology diagnosis). Why image recognition is helpful to radiology diagnosis is that a lot of low level features in earlier layers of the NN such as detecting edges, curves, positive objects etc. might help the algorithm to learn a bit faster in radiology diagnosis.

Note that instead of replacing the last layer with one layer, we could also add more layers and retrain the NN.

Transfer learning is especially useful when we have lots of data for the task that we transfer from (e.g. 1M images) and less data for task that we transfer to (e.g. 100 radiology images). And the low level features from task that we transfer from could be helpful for learning the task that we transfer to.

There's another version of learning from multiple tasks which is called multi-task learning, which is learning from multiple tasks at the same time rather than learning from one and transfer to another one sequentially. Here is a simplified autonomous driving example:

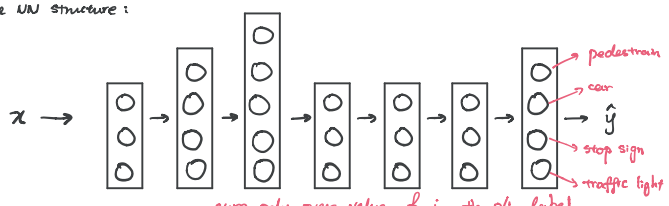
Suppose the self-driving car would need to detect things like pedestrains, cars, stop signs and traffic lights (of course in practice it would be more than 4 things, but for illustration purpose let's say we only need 4 labels)

Andrew Ng

→ for example, this image is labeled as $\begin{bmatrix}0\\1\\1\\0\end{bmatrix}$. So output $y^{(i)}$ is 4x1 matrix. Y is 4xm.

→ with each label being 0 or 1, it means that there isn't or is the corresponding item in the image. Note this is different from softmax regression where it only recognizes what object is it (i.e. only one label)

then the NN structure:



loss: $\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4} L(\hat{y}_j^{(i)}, y_j^{(i)})$ → sum only over value of j with 0/1 label

logistic loss: $-y_j^{(i)}\log\hat{y}_j^{(i)} - (1-y_j^{(i)})\log(1-\hat{y}_j^{(i)})$

Note that multi-task learning is useful when:

- Training on a set of tasks that could benefit from having shared lower-level features.
  (e.g. recognizing traffic lights and cars and pedestrains should have similar features that could also help recognizing stop signs)
- Usually true: amount of data we have for each task is quite similar.

e.g. task 1 : 1M data → transfer → task 2 : 1000 data

task 1 : 1000
task 2 : 1000
task 3 : 1000
⋮
task 100 : 1000
} multi-task learning

- Can train a big enough NN to do well on all tasks (comparing to training different NN to recognize different things in isolation)
  multi-tasking almost never hurts unless the NN is not big enough.

- **End-to-end deep learning**

Briefly describing end-to-end deep learning, it replaces all the multiple stages of data-processing system with just a single neural network.
In the example of speech recognition,

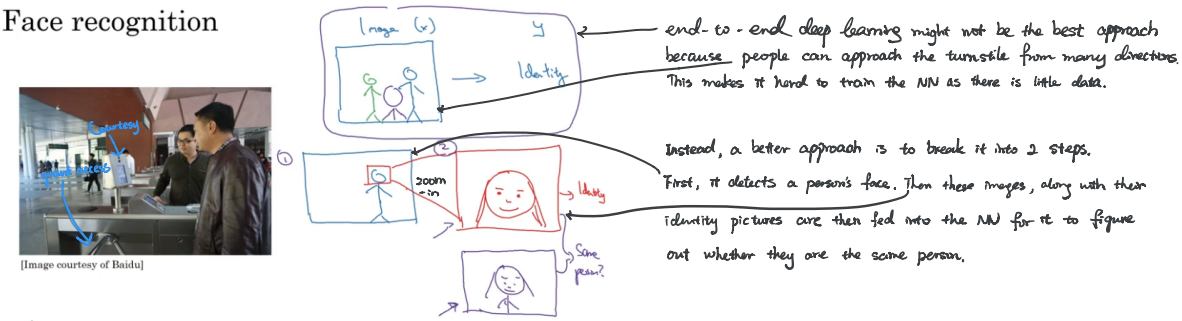
traditionally, people have used the pipeline approach.

x audio $\xrightarrow{\text{MFCC}}$ features $\xrightarrow{M_1}$ phonemes $\longrightarrow$ words $\longrightarrow$ y transcript

whereas in end-to-end deep learning, it bypasses all the intermediate stages

x audio $\xrightarrow{\text{NN}}$ y transcript

The difference is end-to-end deep learning is so much easier to apply than the pipeline approach but it requires much more data (e.g. 10,000 h of audio) for it to work well, whereas the pipeline approach works well even there is less data (e.g. 3,000 h of audio). However, the end-to-end approach doesn't always work better than multi-stages approach. For example, in the example of face recognition, when people approach the turnstile, it will recognizes their faces and decides if let them through.



Why the multi-steps approach works better than end-to-end is that it breaks the problem into 2 simpler problems and for each problem we have more data to feed into.

Here is a more systematic description of when we should/shouldn't use end-to-end deep learning.
First let's look at the pros and cons of end-to-end deep learning.
Pros:

- let the data speaks (let the NN figure out the features rather than forcing human preconceptions, e.g. phoneme)
- less hand-designing of components needed

Cons:

- may need large amount of data
- excludes potentially useful hand-designed components (hand-designed components can inject useful knowledge into the algorithm)

⇒ Key question is: do you have enough data to learn a function of the complexity needed to map x to y?

- sufficient data → let the NN learn by itself and decide what is the best representation
- insufficient data → hand-designed components can inject useful knowledge

It also depends on what kind of task that a NN can do best, e.g. face detecting, image recognition but certainly not route designing.