COURSE 3: STRUCTURING MACHINE LEARNING PROJECTS **2**
Introduction to ML Strategy 2
Orthogonalization 2
Setting up your goal 2
Single number evaluation metric 2
Satisficing and Optimizing metric 3
Train/dev/test distributions 4
Size of the dev and test sets 5
When to change dev/test sets and metrics 5
Comparing to human-level performance 6
Why human-level performance? 6
Avoidable Bias 7
Understanding human-level performance 8
Surpassing human-level performance 9
Improving your model performance 9
Error Analysis 10
Carrying out error analysis 10
Cleaning up incorrectly labeled data 11
Build your first system quickly, then iterate 13
Mismatched training and dev/test set 13
Training and testing on different distributions 13
Bias and Variance with mismatched data distributions 14
Addressing data mismatch 15
Learning from multiple tasks 15
Transfer Learning 15
Multi-task learning 16
End-to-End Deep Learning 17
What is end-to-end deep learning? 17
Whether to use end-to-end deep learning 17

COURSE 3: STRUCTURING MACHINE LEARNING PROJECTS

Introduction to ML Strategy

Orthogonalization

The idea behind orthogonalization is that if you have all your settings/parameters/hyperparameters independent to each other, it'll be easier to tune each one of them separately. If every element is orthogonal to each other, you can change the values of the element without worrying about the other elements also changing its values, because they're linearly independent. Orthogonalization is about what to tune to achieve one effect, without disturbing other components. Let's analyse the chain of assumptions in ML, for example:

-> Fit training set well on cost function
    -It this goal is not achieved, you can take a bigger network, switch to a better optimization algorithm, and so on… without changing the performance of other stages
->Fit dev set well on cost function
    -If your dev set is not performing well, you can try different regularization methods, or to take a bigger training set.
->Fit test set well on cost function
    -You might choose a bigger dev set if your test set is not performing well. You have to try to not change the performance of the training set.
->Performs well in real world
    -You can change the dev set or the cost function, but always trying not to change other components that are already working well.

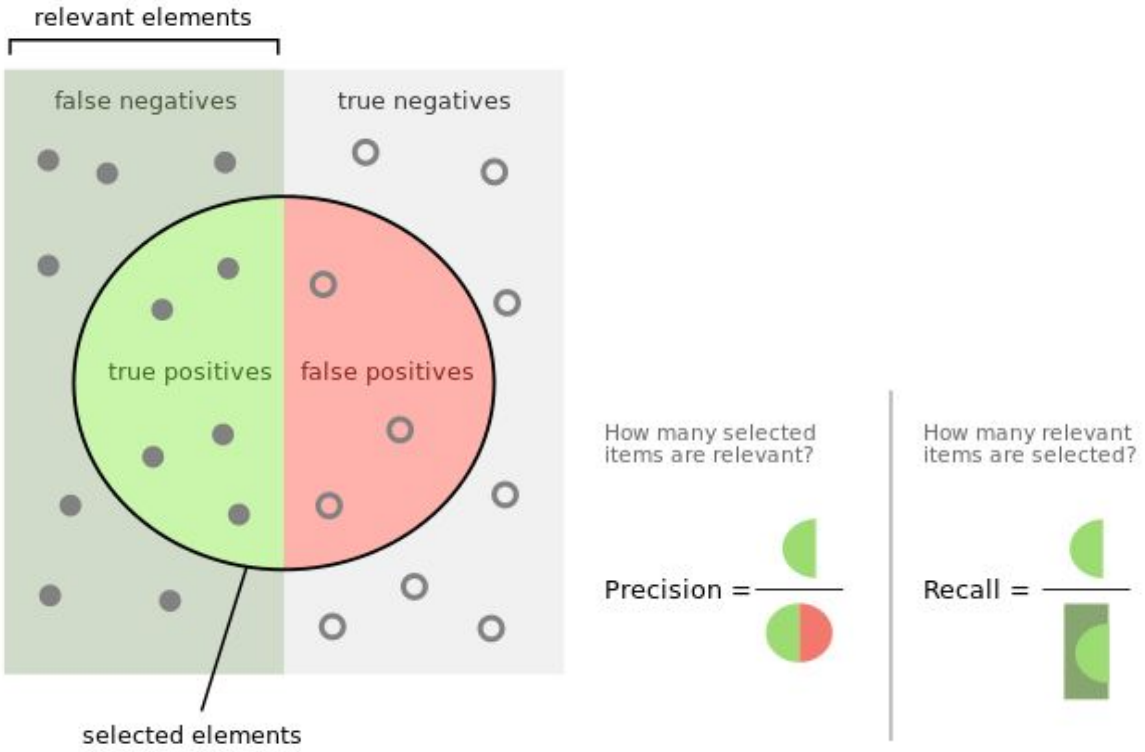
Setting up your goal

Single number evaluation metric

Whether you're tuning hyperparameters, or trying out different ideas for learning algorithms, or just trying out different options for building your machine learning system. You'll find that your progress will be much faster if you have a single real number evaluation metric that lets you quickly tell if the new thing you just tried is working better or worse than your last idea. When you are starting a new machine learning project, it is recommended that you set up a single real number evaluation metric for your problem. Let's see how it's done:
First, let's remember the definition of precision, and introduce the definition of recall:

-Precision: Of all the examples recognized by the neural network, what percentage of them are actually correct?
-Recall: What percentage of the relevant data is correctly recognized?

It turns out that there's often a tradeoff between precision and recall, and you have to care about both. You can see a more intuitive picture below:



It's actually quite challenging to evaluate the performance of a NN using only the precision and recall values. So in order to solve this problem, let's introduce a new metric called F1 Score. F1 score informally speaking, works as the "average" of precision and recall. It's defined by the following equation:

$$F1\ Score\ =\ \frac{2}{\frac{1}{P}+\frac{1}{R}}$$

It's actually a harmonic mean.

In conclusion: the idea behind choosing a single real number to evaluate the performance of your models is very helpful, so you can then choose what model performs better. There are different evaluations possibilities, such as simple mean, harmonic mean, geometric mean. You have just to decide which one is good for your application.

Satisficing and Optimizing metric

Let's suppose you have a cat classification model. You are trying to choose between 3 different classifiers. The performance of the classifiers is shown below:

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

In this case, let's suppose that you want to maximize the accuracy, subjecting the running time to be a maximum of 100ms. In this case the accuracy is your optimizing metric, and the running time is the satisficing metric. More generally, if you have N metrics, you'll choose 1 optimizing metric, and the others (N-1) metrics are going to be satisfying metrics. With this you'll have an automatic way of picking a good classifier. The optimization can be done using different optimization methods.

Train/dev/test distributions

The way you set up your training sets, development sets and test sets, can have a huge impact on how rapidly you can make progress on building your machine learning application. Let's take a look at how you can set up these data sets in order to maximize your efficiency - in this first moment, we'll start with the dev and test sets. Imagine you are setting up the data for a cat classification model. These regions below are the regions you've collected data:

Regions:

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

Now imagine if you set the dev set to be the data collected in the US, UK, Other Europe and South America, and your test set comes from India, China, Other Asia and Australia. In this case, the dev and test sets don't come from the same distribution. This can turn into a huge problem, your team can spend months adapting the model to work with the dev set, just to realize it doesn't work with the test set. In this case you should make sure the dev and test sets come from the same distribution. You can randomly shuffle the data from all the regions, and select part of them to be the dev set, and the rest to be the test set. In this case you make sure the sets will have the same distributions, avoiding problems.

One interesting analogy is that if you don't use the same distribution for your dev and test sets, you are just training how to hit one specific target. Some time later, you'll realise that there is another relevant target, and you are not trained to hit it.

Guideline:
Choose a dev set and test set to reflect data you expect to get in the future and consider important to do well on. In particular, the dev set and test set have to come from the same distribution.

Size of the dev and test sets

In earlier eras of machine learning, the datasets sizes were considerably small, ranging from 100 to a few thousands. It was very common to split the data in 70% train and 30% dev, ou even 60% train, 20% test and 20% dev. However, we are in the big data era, and datasets can have millions of examples. In this case, keeping the percentage of the datasets as it was in the past is not a wise decision. The best solution is to split the data in: 98% train set, 1% test set and 1% dev set.
You should set the size of your test set to be big enough to give high confidence in the overall performance of your system. Another consideration is that you don't hardly need a test set. If your dataset is not very big, you might consider just having the train and dev sets, and your model should work just fine (by the way, if you have enough data, use a test set!!!).

When to change dev/test sets and metrics

Let's start with an example. You are building a cat classifier to try to find lots of pictures of cats to show to your cat loving users and the metric that you decided to use is classification error. You've set up two different algorithms. Their performance can be seen below:

Metric: classification error

Algorithm A: 3% error

Algorithm B: 5% error

If you just look for the best performance, you would choose algorithm A. However, you discover that, for some reason, algorithm A is letting through a lot of

pornographic images. This behaviour is completely unacceptable, so you should choose algorithm B. When your evaluation metric is no longer correctly rank ordering preferences between algorithms, then that's a sign that you should change your evaluation metric or perhaps your dev set or test set. In this case, the misclassification error metric you are using can be written as follows:

$$Error = \frac{1}{m_{dev}} \sum_{i=1}^{m_{dev}} L(y_{pred}^{(i)} \neq y^{(i)})$$

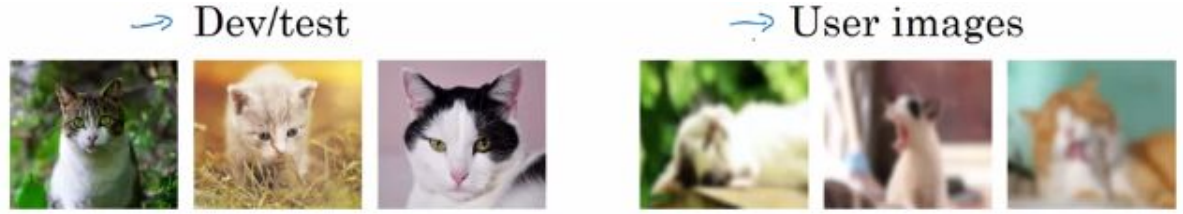
One solution to our misclassification problem, is to give different weights to different classifications. For example, if the classifier predicted some image as a porn image, it would give a high weight to this prediction, increasing the error. This new metric can be written as follows:

$$Error = \frac{1}{\sum_{i=1}^{m_{dev}} w^{(i)}} \sum_{i=1}^{m_{dev}} w^{(i)} L(y_{pred}^{(i)} \neq y^{(i)})$$

$$w^{(i)} = 1 \text{ if } x^{(i)} \text{ is non} - porn$$
$$10 \text{ if } x^{(i)} \text{ is porn}$$

With the philosophy of orthogonalization, one step you should always take is to define a metric to evaluate the classifiers, and the other separate step is to worry about how to do well on this metric.

Another example: you have 2 algorithms for a cat classifier. Algorithm A presented a 3% error, and algorithm B presented 5% error. You might think to choose algorithm A, however, when you test it with user images the algorithm B has a better performance. It happens because you've trained your classifier using high quality images, and the reality is not like that.



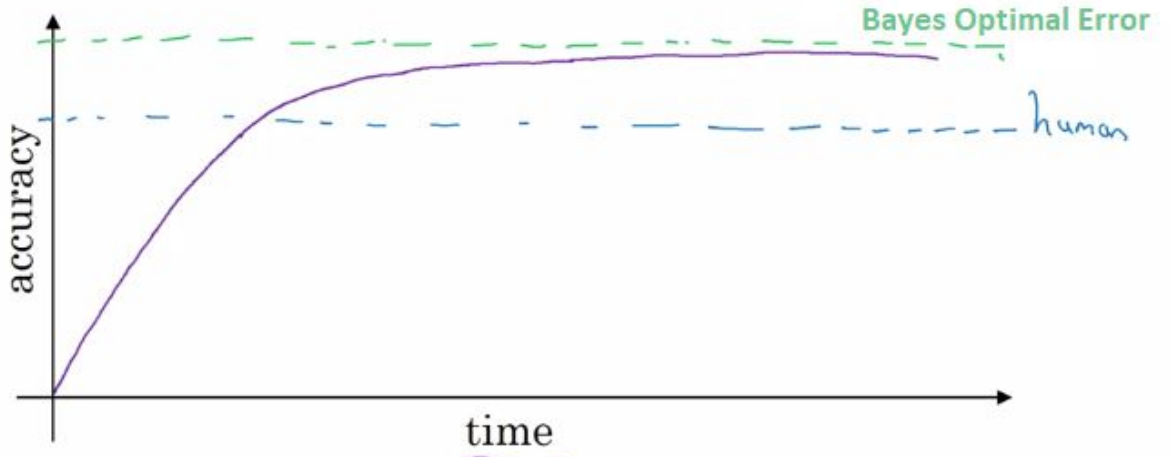
In this case, if doing well in your metric and in the dev/test sets does not correspond to doing well on your application, change your metric and/or dev/test set.

Comparing to human-level performance

Why human-level performance?

With the advancements in the deep learning area, machine learning algorithms are working much better, and the comparison between the algorithms and the human performances are much more common. The performance of machine and deep

learning algorithms can surpass the human performance, and achieve a certain limit, called the Bayes Optimal Error (you can think the Bayes Optimal Error as the best possible error that can be achieved by the learning model).



As you can see in the plot above, when the learning model passes the human performance, it slows down the learning. It happens because of two reasons: the human performance is usually really close to the Bayes Error, so it's quite challenging to get better than the human performance. Another reason is, if the performance of the learning model is still worse than human performance, there are some tools that can be used to improve performance, tools that are harder to use when you surpass human performance, you can see some of these tools below:

Humans are quite good at a lot of tasks. So long as ML is worse than humans, you can:
-Get labeled data from humans;
-Gain insight from manual error analysis: Why did a person get this right?
-Better analysis of bias/variance.
These techniques cannot be used when your learning model presents a better performance compared to the human performance.

Avoidable Bias

Let's suppose you have the following errors in your cat classifier for two models (each column is a different model):

Humans	1%	7.5%
Training error	8%	8 %
Dev error	10%	10 %

In the first model, you have trained the model with good quality images, and the human error in this case is only 1%. As we are dealing with images, we can consider the human error as an approximation to the Bayes Error. Your training error is 8%. The gap between the Bayes error and your training error is 7%, and this is a high number. The difference between the training error and the dev error is 2%. In this case, you really should try to focus on bias, to get a better value of the training error, as close as possible to the Bayes error.

In the second model, you have trained the model with bad quality images, and the human error has improved a lot, and is 7.5%. We still consider the human error being close to the Bayes Error. Your model somehow managed to keep the training and dev errors equals to the previous model. In this case, the difference between the Bayes error and your training error is only 0.5%. This difference is called the avoidable bias (you want the avoidable bias to be as close to zero as possible). The dev error is quite different from the training error, and as we have a low avoidable bias, you might focus your attention on the variance, to get a better value of the dev error. (the difference between the dev error and the training error is called variance)

In conclusion:
-The difference between the Bayes error and the training error is called the avoidable bias (you want the avoidable bias to be as close to zero as possible). If this difference is high, you might deepen your model, try some different architectures, use more hidden units...
-The difference between the dev error and the training error is called variance. To get a lower variance, you can try techniques like regularization, get more training data (using data augmentation, for example), and so on…
-To choose what you should improve, just analyse which one is bigger. If the avoiding bias is bigger than the variance, you should focus on the bias. If the variance is bigger than the avoiding bias, you should focus on the variance.

Understanding human-level performance

As we saw in the previous section, human-level error in certain applications is a very good proxy for the Bayes error. Let’s do an example of medical image classification:

Suppose:

(a) Typical human ..... 3 % error

(b) Typical doctor ..... 1 % error

(c) Experienced doctor ..... 0.7 % error

(d) Team of experienced doctors .. 0.5 % error

In this scenario, what error value should be the “human-level” error?
If we think of human-level error as an approximation to the Bayes error, we should consider the lowest error as possible to be the “human-level” error. In this case, the human-level is 0.5%. We know with this number that the Bayes error is $< 0.5\%$, and we can try to train our model to get close to this number (however, to deploy an application, an error close to 1% might be okay, because it’s the error of a typical doctor).

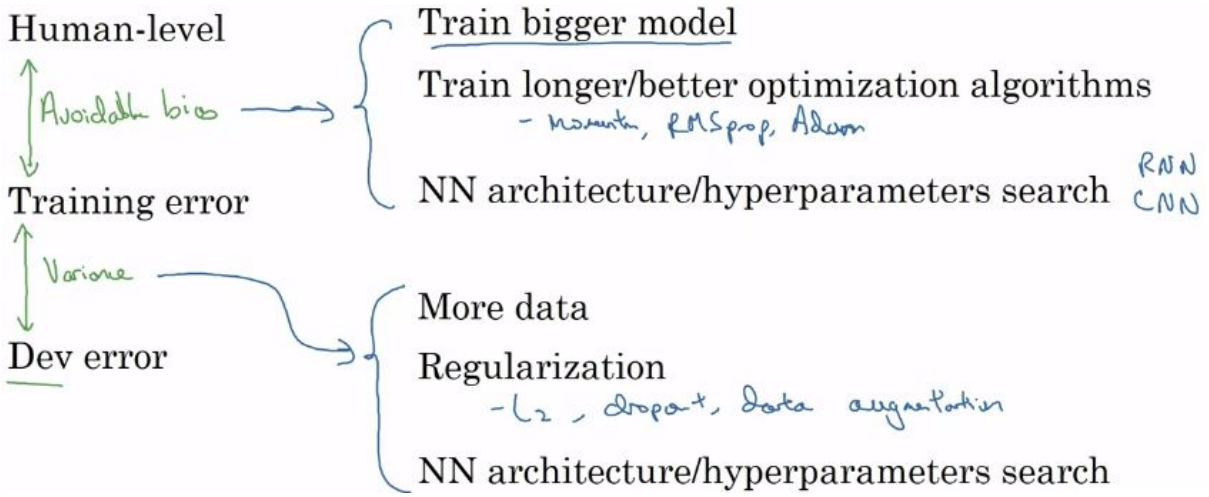
Surpassing human-level performance

When machine learning suspasses the humam-level performance, it gets very difficult to improve the performance of the model, because we don’t have a threshold anymore to base our learning on, and we don’t know what we should try to improve, like trying to reduce the bias or the variance. There are lots of problems where machine learning significantly surpasses human-level performance, such as:
-Online advertising;
-Product recommendations;
-Logistics (predicting transit time, for example);
-Loan approvals;
These four applications are actually learning from structured data, and these are not natural perception problems (they’re not computer vision, natural language processing or speech recognition) which humans are very good at. And finally, all these applications have analysed more data than any human would be capable of.

Improving your model performance

The two fundamental assumptions of supervised learning:
1 - You can fit the training set pretty well (you can think of this as roughly saying that you can achieve low avoidable bias).
2 - The training set performance generalizes pretty well to the dev/test set (this is sort of saying that variance is not too bad).

Then, to reduce avoidable bias and variance, you should examine these following values:



Error Analysis

Carrying out error analysis

The process of examining the mistakes that your algorithm is making can give you insights into what to do next. This process is called error analysis. Let's start with an example.

Imagine you are trying to improve the performance of your cat classification model. It has 90% accuracy and 10% error. You then realize some of the miscategorization are dog pictures. In this case, should you try to make your cat classifier do better on dogs?

The answer is - not yet. First you should make an error analysis, and find out if the dogs miscategorization is really the problem. To make this analysis, you should take around 100 mislabeled dev set examples, and analyse each one of them manually. If a really small number of these examples are dog pictures (like 5/100, for example), then you should not waste months of work trying to perform best with dog pictures, because this is not the main problem of your model. If a considerable number of the pictures are actually dog pictures (like 50/100, for example), now you could be a lot more optimistic about spending time on the dog problem. You can also evaluate multiple ideas in parallel, to see what is the performance issue of your model, ideas such as:

-Fix pictures of dogs being recognized as cats

-Fix greats cats (lions, panthers, etc…) being misrecognized

-Improve performance on blurry images

What you can do to make this analysis, is to create a spreadsheet, put all the ideas in the columns and each miscategorization image in a row, like this:

Image	Dog	Great Cats	Blurry	Comments
1	✓			Pitbull
2			✓	
3		✓	✓	Rainy day at zoo
⋮	⋮	⋮	⋮	
% of total	8%	43%	61%	

The conclusion of this process gives you an estimate of how worthwhile it might be to work on each of these different categories of errors. In the example above, clearly a lot of the mistakes we made on blurry images, and quite a lot on great cat images. This can give you a good idea on where to improve.

Cleaning up incorrectly labeled data

The data for your supervised learning problem comprises input X and output labels Y. What if you go through your data and you find that some of these output labels Y are incorrect. Is it worth spending time to fix up some of these labels? You'll see more about this problem in this section.

The term mislabeled example is used to refer if in the dataset, whatever a human label assigned to a piece of data, is actually incorrect. You can see an example below, in the cat recognition model:

In this case, there's a mislabeled example in the sixth picture. A dog picture is labeled as a cat picture.

If your mislabeled errors are in the training set, DL algorithms are quite robust to random errors, in this case you should not worry too much about this problem. Note that DP algorithms are robust to random errors only. If there are some systematic errors, you should worry about it (for example, if all the white dogs images are classified as cats, your classifier will also learn and make this mistake).

In the dev or test sets, if you want to fix these mislabeled errors, you can make a spreadsheet with the error analysis, adding an extra column that can help you to count up the number of examples where the label Y was incorrect. This analysis will help you to decide if it is worth spending some time fixing the wrong labels.

Image	Dog	Great Cat	Blurry	Incorrectly labeled	Comments
...					
98				✓	Labeler missed cat in background
99		✓			
100				✓	Drawing of a cat; Not a real cat.
% of total	8%	43%	61%	6%	

So the question now is, is it worthwhile going in to try to fix up the incorrectly labeled examples? The answer varies to each problem, but some advices can be seen below:

-If fixing the incorrectly labeled examples makes a significant difference to your ability to evaluate algorithms on your dev set, then go ahead and spend some time fixing the problem.

-If it doesn't make a significant difference to your ability to use the dev set to evaluate the cost bias, then it might be not the best use of your time trying to fix this problem.

Let's analyse one error example:

Overall dev set error 10%

Errors due incorrect labels 0.6%

Errors due to other causes 9.4%

In this case, the majority of the problems are not due incorrect labels. Of all the errors (10%), just 0.6% of them are errors due incorrect labels. It represents a 6% of the overall dev set error. So you should focus your time fixing the errors due to other causes (9.4%, that represents 94% of the overall errors).

Overall dev set error 2%

Errors due incorrect labels 0.6%

Errors due to other causes 1.4%

In this other example, of all the errors (2%), 0.6% of them are errors due incorrect labels. The incorrect labels represent 20% of all the errors. Then, it maybe seems much more worthwhile to fix up the incorrect labels in your dev set.

Obs: Remember, the goal of the dev set is to help you select between two classifiers A & B.

Guideline to correct incorrect dev/test set examples:

-Apply same process to your dev and test sets to make sure they continue to come from the same distribution
-Consider examining examples your algorithm got right as well as ones it got wrong
-Train and dev/test data may now come from slightly different distributions.

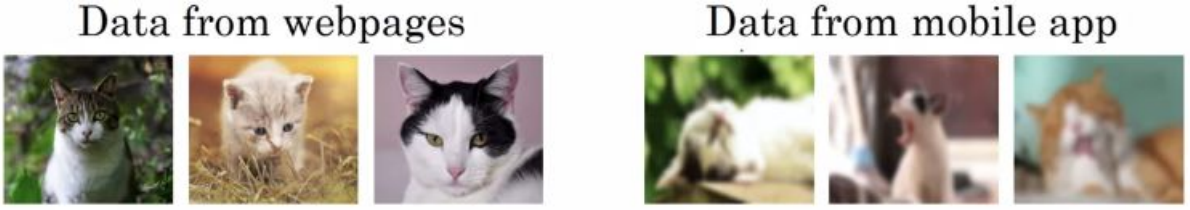
Build your first system quickly, then iterate

If you're working on a brand new machine learning application, one advice you should take, is to build your first system quickly and then iterate. First, you should set up your dev/test set and decide what metric to focus on, and then build an initial system quickly. After iterating one time, you should use bias/variance analysis & error analysis to prioritize next steps. The general idea is: build more simple systems, iterate, prioritize, evolve the system.

Mismatched training and dev/test set

Training and testing on different distributions

As we saw in the previous sections, it's a good practice to use the same distribution for the dev and test sets. However, some applications might need a lot of data, and this data is not always available. In this case, using data from different distributions can be a solution to this problem. In this section, we'll see some subtleties and some best practices for dealing with data from different distributions.
Let's do an example with the classic cat recognition algorithm.



Suppose you have 200k images downloaded from webpages and 10k pictures from the users of your mobile app. As you can see in the image above, the mobile app pictures have less quality compared to the pictures from web pages, this makes them come from different distributions. In this case, you care a lot about the 10k, because they come from the users of your app. You should divide the data in the following way:

Of all the 210k images you have, put all the 200k from the webpages in your training set. Split the data from the mobile app in 3 segments, one with 5k and two with 2.5k. Each dataset is going to be like this:
-Training set: 200k web images + 5k mobile app images
-Dev and test sets: 2.5k mobile app images

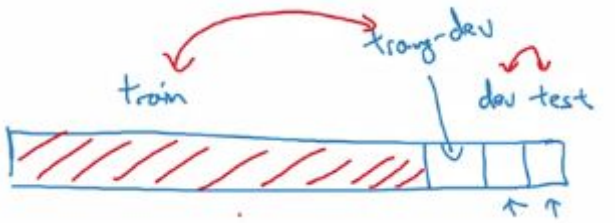
The disadvantage of this division is that the training and dev/test sets come from different distributions, however it will perform better compared to the dataset with only webpage images.

Bias and Variance with mismatched data distributions

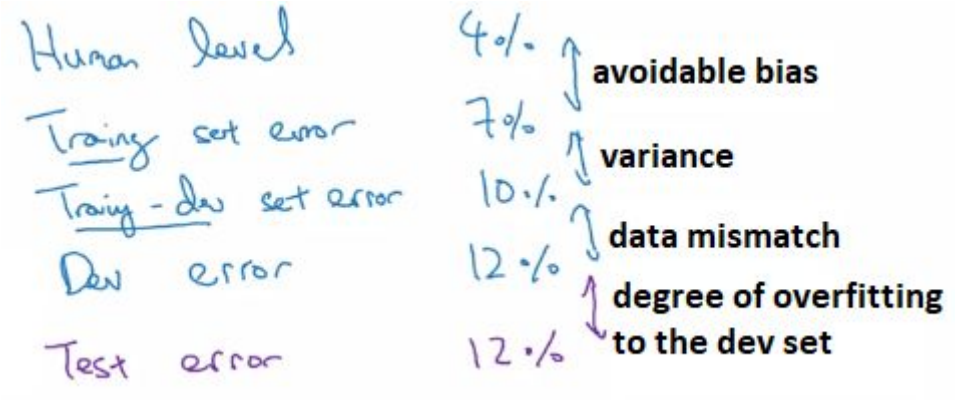
Estimating the bias and variance of your learning algorithms really helps you prioritize what to work on next. But the way you analyse bias and variance changes when your training set comes from a different distribution than your dev and test sets. Let's see how.

We'll keep using our cat classification example. Let's assume humans get almost 0% error (this will be our proxy for the Bayes Error). The training error is 1% and the dev error is 10%. In this case, using data from the same distribution, we would say that there is a variance problem. The problem with this analysis is that when you went from the training error to the dev error, two things changed at a time. One is that the algorithm is not used to the dev set data and two is that the distribution of data in the dev set is different. Because two things changed at the same time, it's difficult to know how much of the error is because the algorithm didn't see the data in the dev set (that's the variance part of the problem), and how much of the error is because the dev set is just different.

In order to solve this problem, it will be useful to define a new piece of data which we'll call the training-dev set. This is a new subset of data that should have the same distribution as training set, but not used for training.



In this case, you should analyse the training-dev error. In our example, if the training-dev error is 9%, then we can conclude that there is a variance problem, and our algorithm doesn't have a very good performance classifying the data (it's not generalizing well to data from the same distribution). If the training-dev error is just 1.5%, we can conclude that we have a pretty low variance problem and a data mismatch problem (the difference in the distributions are making a huge problem in the model). The image below represents a more general example:



Addressing data mismatch

If the error analysis shows you that you have a data mismatch problem, there aren't completely systematic solutions to solve this problem, but let's take a look at some things you could try:
-Carry out manual error analysis to try to understand differences between training and dev/test sets
-Make training data more similar; or collect more data similar to dev/test sets. To do this you can use some techniques, such as artificial data synthesis, that is explained below.

Artificial data synthesis consists in adding some additional characteristics to the already existing data, such as some noise (in audio cases), blur (in image cases), and so on… . You have to be careful using this technique, because it can generate an overfitting to the data you selected to synthetize.

Learning from multiple tasks

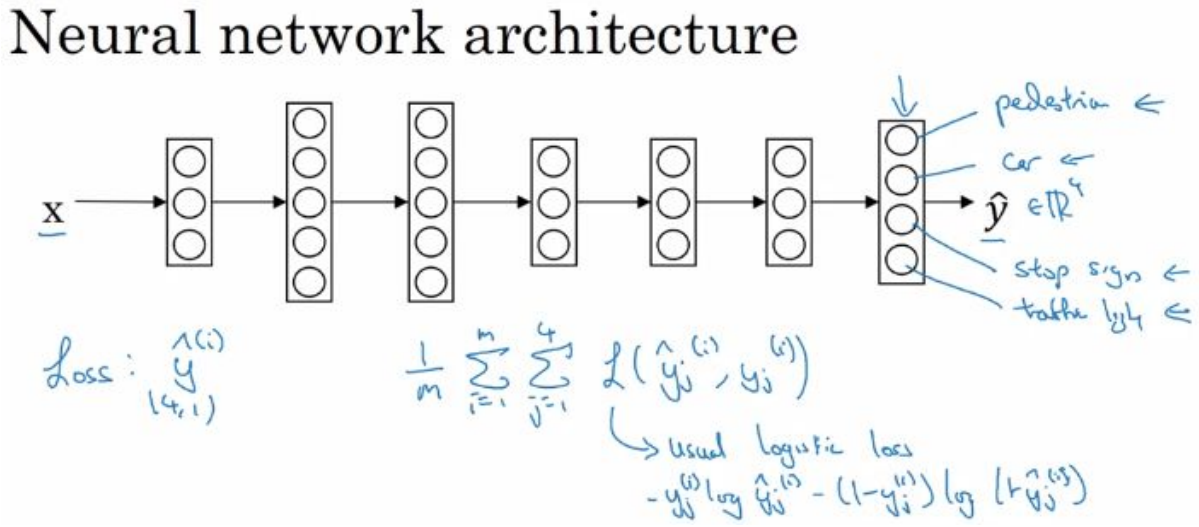
Transfer Learning

One of the most powerful ideas in deep learning is that sometimes you can take knowledge the neural network has learned from one task and apply that knowledge to a separate task. This is called transfer learning. For example, you can create a neural network model to recognize images, and then if you want to make radiology diagnosis, you can use the generic neural network and change the final layers of the neural network, making it work with radiology images. The process of training a generic model for image recognition is called pre-training, and making the adaptations in the model to work with the kind of image you want to recognize is called fine-tuning. You can also train a generic model for speech recognition, then specify the model to recognize tasks such as wake words for virtual assistants (such as "Alexa", "OK Google", "Hey Siri" or "Ni hao Baidu").

Transfer learning is usually used when you have a lot of data from the problem you're transferring from and usually relatively less data to the problem you are transferring to. To generalize, if you are trying to transfer from A to B, transfer learning makes sense when:
-Task A and B have the same input x (images, audios…)
-You have a lot more data for Task A than Task B.
-Low level features from A could be helpful for learning B.

Multi-task learning

Whereas in transfer learning, you have a sequential process where you learn from task A and then transfer that to task B, in multi-task learning you start off simultaneously, trying to have one neural network do several things at the same time. Then each of these tasks helps all the other tasks. Let's start with an example of an autonomous driving vehicle:
Suppose we have a model that needs to recognize pedestrians, cars, stop signs and traffic lights. The neural network is going to have 4 different outputs (4,1) vectors. You can see the NN architecture below:



As you can see, the NN has four output units. The difference of the cost function compared to the earlier binary classification examples, is that now it is summing over j = 1, …, 4 (1 to 4 output units). Also the main difference between this model and the softmax regression is that unlikely softmax regression which assigned a single label to a single example, now a single image can have multiple labels (outputs). In this case, a single image can have a pedestrian, a car, a stop sign and a traffic light.
You could create 4 different neural networks, each one to identify one label. However, some earlier features in the neural network usually can be shared between the four objects, then you'll find that training a single network to do 4 things results in better performance than training four completely separate neural networks to do the four tasks separately.
To summarize, multi-task learning makes sense when:

-Training on a set of tasks that could benefit from having shared lower-level features.
-Usually: amount of data you have for each task is quite similar
-Can train a big enough neural network to do well on all the tasks.

End-to-End Deep Learning

What is end-to-end deep learning?

Usually some data processing systems, or learning systems require multiple stages of processing. End-to-end deep learning is a technique that allows you to replace all those multiple stages with a single neural network. End-to-end deep learning works like a “brute-force” technique. Let’s see an example in the speech recognition domain.
The traditional approach design for a spoken language understanding system is a pipeline structure with several different components, exemplified by the following sequence:

Audio (input) -> feature extraction -> phoneme detection -> word composition -> text transcript (output)

A clear limitation of this pipelined architecture is that each module has to be optimized separately under different criteria. The E2E approach consists in replacing the aforementioned chain for a single neural network, allowing the use of a single optimization for enhancing the system:

Audio (input) -> ---------- (NN) ---------- -> transcript (output)

However, to make this “brute-force” process of replacing all the middle stages with a neural network requires tons of data, and this is a very difficult process to validate.

Whether to use end-to-end deep learning

Let’s take a look at some pros and cons of end-to-end deep learning.
Pros:
-Let the data speak
-Less hand-designing of components needed
Cons:
-May need large amount of data
-Excludes potentially useful hand-designed components
-Difficult to improve or modify the system
-Difficult to validate

Applying end-to-end deep learning

Key question: Do you have sufficient data to learn a function of the complexity needed to map x to y?
To make an end-to-end application to perform well, you need huge amounts of data.