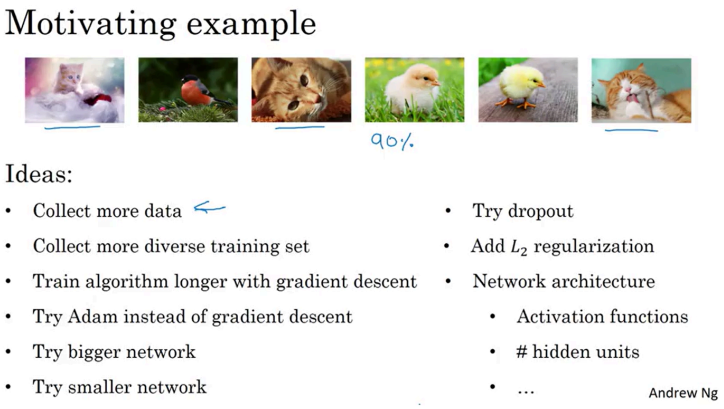
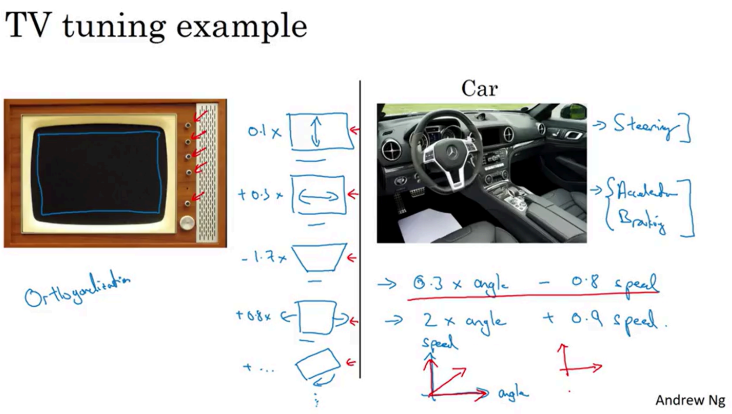
Week 1: ML Strategy

Why ML Strategy?



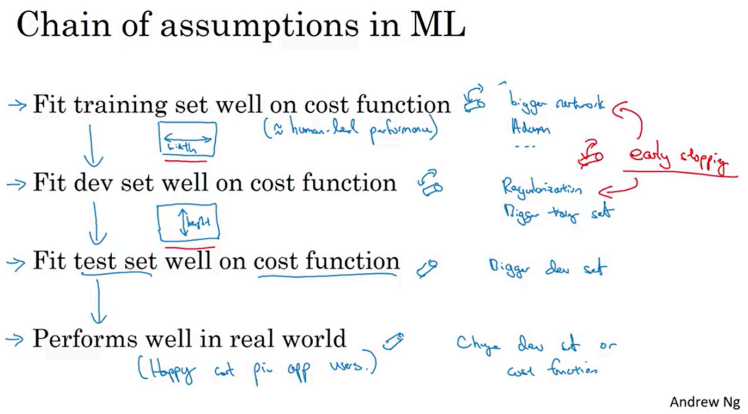
- In ML projects, reaching high accuracy (like 90%) often isn't enough—you'll want to go further.
- You will have *many possible things to try*: more data, new models, different regularization, new optimizers, etc.
- Not all ideas work equally well—some can waste a lot of time with little effect.
- ML strategy means analyzing your problem and data, so you select ideas that really improve your system.
- Many teams waste months collecting data or tuning parameters, only to find little impact. Choosing the right path is essential.
- The course teaches practical, industry-proven strategies for ML project improvement—often not found in standard university classes.
- New ML strategies are needed for deep learning problems (not always classical ML rules).
- Good ML strategy skills make you much more effective and productive in developing, improving, and deploying ML solutions.

Orthogonalization

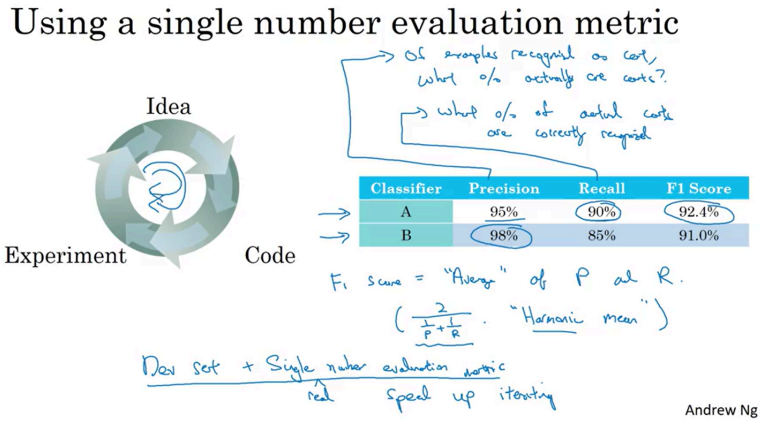


- **Orthogonalization** is about designing your ML workflow so each "knob" (hyperparameter or step) lets you adjust one specific part of system behavior, without accidentally changing others.

- **Analogy:** Old TVs had separate knobs for width, height, etc., making tuning simple. If a knob changed multiple things at once (width, position, etc.), tuning would be confusing.
- **ML Application:** In your ML system, you want to clearly know:
  - Which parameter or method improves training set performance?
  - Which one tunes dev set generalization?
  - Which one helps test set or real-world results?
- **Typical "knobs" for each step:**
  1. **Training set fit**: Bigger networks, better optimizers (e.g., Adam).
  2. **Dev set generalization**: Regularization (dropout, weight decay), more data.
  3. **Test set performance**: Make dev and test sets more representative, get a larger dev set.
  4. **Real world performance**: Redefine cost function or data distribution to match real scenarios.
- **Bad example:** Early stopping can affect both training and dev set performance—less orthogonal, harder to control.
- **Goal:** For every problem (poor training, dev, test, or real world fit), use a mostly independent "knob" for tuning, making debugging faster and improvements easier.
- **Summary:** Orthogonalization gives you clear, separated controls over each aspect of your ML system—just like tuning a car's steering and speed separately, not with a single mixed joystick.



Single Number Evaluation Metric



Another example

Algorithm	US	China	India	Other	Average
A	3%	7%	5%	9%	6%
B	5%	6%	5%	10%	6.5%
C	2%	3%	4%	5%	3.5%
D	5%	8%	7%	2%	5.25%
E	4%	5%	2%	4%	3.75%
F	7%	11%	8%	12%	9.5%

averaging error rate of each algorithm and taking the least one (i.e., C has least among these)

Purpose of Single Number Evaluation Metric:

- In machine learning model development, progress is **empirical**—constantly trying new ideas, tuning hyperparameters, changing algorithms.
- **Rapidly evaluating changes** is critical. Comparing many models is most efficient if you can use a **single real-number metric** to judge performance.

Key Video Concepts:

- For tasks such as cat detection, you could measure **precision** and **recall**.
  - *Precision*: Of all the images the model said "cat," how many really were cats?
  - *Recall*: Of all the actual cat images, how many did the model identify?
  - Usually, precision and recall are a tradeoff—improving one can lower the other.
- Comparing models is difficult if one has better precision and another has better recall.
- Best practice: **Combine metrics** into a single score for ease of comparison and faster iteration.

Expanded Metric Definitions (With Formulas & Examples):

Metric	Formula	What it tells you	Best used when...
Accuracy	$\frac{TP+TN}{TP+FP+FN+TN}$	**Fraction of all correct predictions**	Classes are balanced
Precision	$\frac{TP}{TP+FP}$	**How many predicted positives are truly positive**	False positives are costly
Recall	$\frac{TP}{TP+FN}$	**How many actual positives are caught**	False negatives are costly
F1 Score	$2 \times \frac{Precision \times Recall}{Precision+Recall}$	**Harmonic mean of precision/recall**	Imbalanced/classes, want single score

- **TP** – True Positives: Correctly predicted positive cases (model says "cat" and it's a cat).
- **FP** – False Positives: Incorrectly predicted positive (model says "cat" but it's not a cat).
- **FN** – False Negatives: Actual positives missed (model says "not cat" but it's a cat).
- **TN** – True Negatives: Correctly predicted negatives (model says "not cat" and it's not a cat).

Example (Cat Classifier):

Suppose your model outputs:

- 90 true cat images as “cat” (TP)
- 10 images wrongly marked as “cat” (FP)
- 20 cat images missed (FN)
- 80 non-cat images as “not cat” (TN)
- Total = 200 images
- **Accuracy:** $(90 + 80)/200 = 85\%$
- **Precision:** $90/(90 + 10) = 90\%$
- **Recall:** $90/(90 + 20) = 81.8\%$
- **F1 Score:** $2 \times \frac{0.9 \times 0.818}{0.9 + 0.818} \approx 0.857$

  or 85.7%

Why F1 Score for Single Number?

- F1 gives you a **single value** summarizing how well the model balances between precision and recall.
- If either precision or recall is low, F1 drops quickly—so both must be strong to get a high F1.
- This single value enables:
  - Faster iterations and empirical improvements.
  - Easier model ranking.
- Multiple metrics—precision and recall separately—are still useful for diagnostics, but team progress hinges on a **single-number metric**.

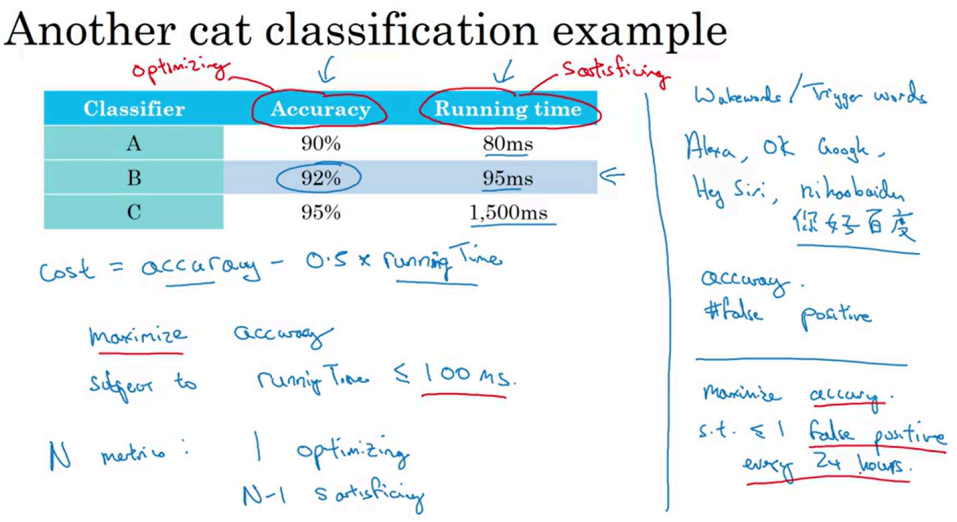
Best Practices (from Video & ML Science):

- **Define your primary evaluation metric in advance** (accuracy if balanced, F1 for imbalance, etc.).
- Use other metrics for **analysis**, but guide model selection by your main single-number metric.
- For geo-segmented or multi-task settings, consider averaging error/score across all tasks to get one scalar for comparison.
- Future steps (as hinted by the video): Learn about “optimizing” and “satisficing” metrics for advanced models with multiple performance criteria.

Core Insight:

> Use a single-number evaluation metric (like accuracy or F1) as your main compass for iterative ML development. This maximizes the speed and clarity of progress, especially when testing and comparing many models. Keep precision and recall close by for diagnosis, but let one scalar lead the way.

Satisficing and Optimizing Metric



- **Challenge in Metrics:** It's often hard to combine all aspects you care about in a single numeric metric for model evaluation.
- **Satisficing vs. Optimizing Metrics:**
  - *Optimizing metric* is the key metric you want to maximize or minimize (e.g., accuracy).
  - *Satisficing metric* only needs to reach a "good enough" threshold; beyond that, improvements are not your focus (e.g., running time should be <100 ms—once satisfied, faster isn't important).
- **Example:** Given three classifiers with different accuracies and running times:
  - Instead of a weighted sum (e.g., accuracy minus 0.5 * running time), choose to maximize accuracy as long as running time ≤ 100 ms.
  - Here, accuracy = optimizing metric, running time = satisficing metric.
- **General Rule:** Out of N metrics:
  - Pick one for optimizing (do as well as possible on it).
  - Set thresholds on the remaining (satisficing) metrics—they must just meet the requirement.
- **Wake Word Example:** For a voice assistant:
  - Maximize wake word detection accuracy (optimizing).
  - Satisfice on false positives (e.g., ≤1 per 24 hours).
- **Benefit:** This approach gives a practical, almost automatic way to pick the "best" model among multiple candidates—just check who passes all satisficing thresholds, then pick the best on the optimizing metric.
- **Application:** Always evaluate these metrics on the appropriate dataset (train, dev, or test).

Train/Dev/Test Distributions

Cat classification dev/test sets

development set, hold out cross validation set

Regions:

- US
- UK
- Other Europe
- South America

Dev

- India
- China
- Other Asia
- Australia

Test

Randomly shuffle into dev/test

dev set + metric

Idea

Experiment

Code

Andrew Ng

True story (details changed)

Optimizing on dev set on loan approvals for medium income zip codes

$x \rightarrow y$ (repay loan?)

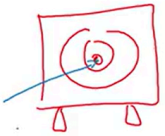
Tested on low income zip codes

~3 month

Guideline

Same distribution

Choose a dev set and test set to reflect data you expect to get in the future and consider important to do well on.



training

dev metric

test

Key Concepts & Practical Recommendations:

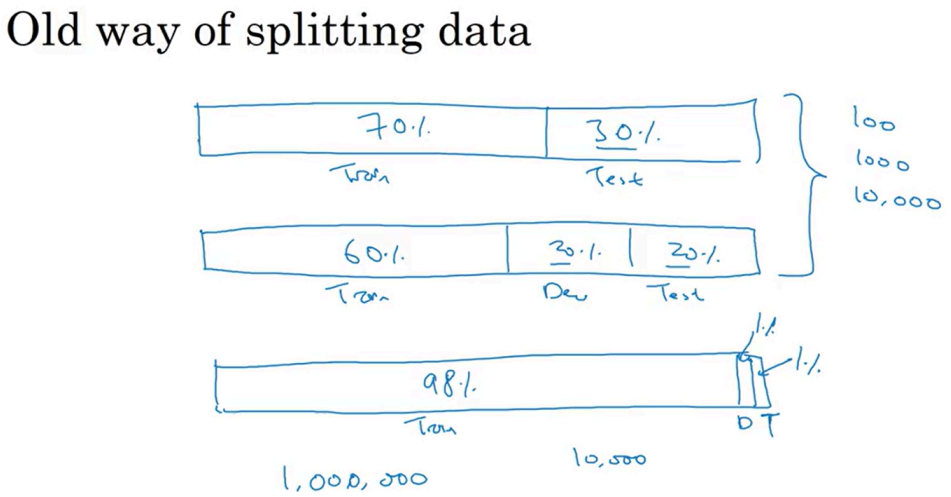
- **Significance of Dataset Setup:**
  - How you set up your train, dev (development), and test sets greatly affects *how quickly and effectively* you can build and improve machine learning applications.
  - Poor setup—even in large organizations—can slow down progress and waste effort.
- **Dev Set = Development Set = Holdout/Cross-validation Set:**
  - Used for evaluating ideas/models during development.
  - The workflow: Train different models → Evaluate them on the dev set → Pick the best-performing one on dev → *Then* evaluate that model on the test set.
- **Common Mistake:**
  - **Setting dev and test sets with different distributions:**
    - Example: If working on a cat classifier covering multiple world regions (US, UK, India, China, etc.), don't assign certain regions only to dev and others only to test.
    - This makes the dev and test sets *distributionally different*, causing problems—models that perform well on dev might fail on test, wasting months of optimization.
- **Proper Practice:**
  - **Dev and test sets should have the *same* distribution:**
    - Randomly shuffle and sample data so both sets include examples from *all* relevant regions/types/classes.
    - This places the "target" (the metric you want to optimize) in one clear spot for your team, so their improvements on dev are predictive of improvements on test.
- **Analogy:**
  - Setting different distributions is like aiming for one bullseye for months, only to be judged suddenly on a different target.
  - Teams need a stable, reliable "bullseye" (metric + data distribution) to innovate efficiently.
- **Real-World Example:**
  - ML team optimized a loan approval classifier using only *medium income* zip codes for dev.
  - Tested later on *low income* zip codes—distribution is different, performance dropped, and months of work was lost.
  - Lesson: Always ensure dev and test distributions *match the data you care about in production*.
- **Guidelines:**
  - Choose dev and test sets that reflect the data you expect to see in the future.
  - The goal: Put the "target" where you *actually* want to hit—this lets your team optimize efficiently for real-world success.
- **Takeaway:**
  - Setting the *dev set and evaluation metric* defines the target for your project.
  - Set dev and test sets from the same distribution (as similar as possible to production/future data).
  - The training set setup is important—but will be discussed later.

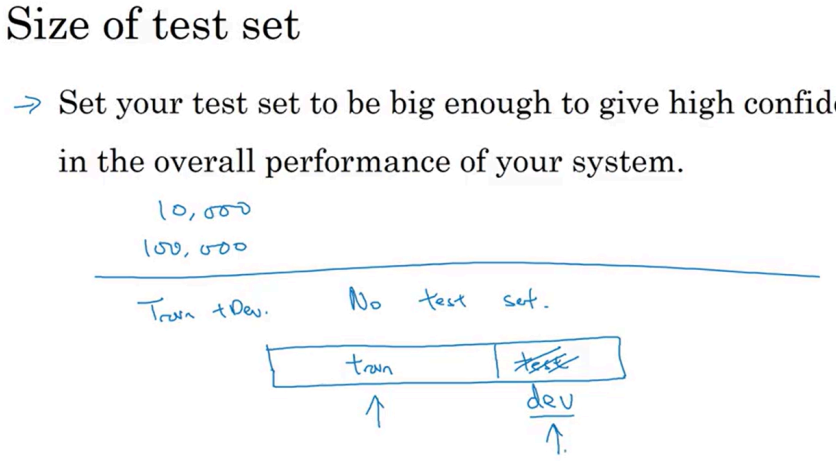
Summary Tips:

- *Never* use non-representative or separated data splits for dev and test.
- Always ensure dev/test mirror future/real-world data.

- Setting the right dev/test distributions can save you and your team *months* of wasted work; innovation becomes much faster and more effective.

Size of the Dev and Test Sets





Key Points & Practical Guidelines:

- **Traditional Split Ratios:**
  - Old rule of thumb: Use a **70/30 split** for training and test sets, or **60/20/20** for training, dev, and test.
  - This worked well when datasets were small (hundreds to a few thousands of examples).
- **Modern "Big Data" Practice:**
  - When you have **very large datasets** (example: 1,000,000+ samples), you *do not* need such a large percentage allocated to dev/test.
  - A good practice: **98% train / 1% dev / 1% test**.
    - For 1 million examples, 1% = 10,000 samples, which is *enough* for both dev and test.
- **Why So Much Training Data?**

    - **Deep learning models have a huge "hunger" for data**—the more they can train on, the better they perform, so prioritize putting most data into training.
- **How Large Should Test Set Be?**
    - Test set's purpose: to provide **high confidence in your model's final performance** before deployment.
    - You *don't* need "millions" in test set—just enough for a **reliable performance estimate** (e.g., 10,000 or 100,000, depending on application).
    - If you only need *low* confidence, you can use a smaller test set—or *possibly* skip the formal test set (though this is uncommon and not generally recommended for important projects).
- **Historical Practice Series**
    - Sometimes, people split their data into "train" and "test," but actually used the test set for iterative tuning.
    - If you're actively adjusting models based on test set performance, that set is really serving as a **dev set**—not a true, final "blind" test.
    - **Best practice:**
        - Formally distinguish dev (for iteration/tuning) vs test (final performance unbiased estimation) sets.
- **Summary Guidelines:**
    - In the era of big, modern datasets, the **old 70/30 rule** is outdated.
    - Use a **much larger percentage for training**, and much *smaller* percentages (often 1% each) for dev and test when you have enough data.
    - Main goal: make dev/test sets **big enough for their purpose**, NOT a fixed percentage.
- **Caveats:**
    - For projects with little or no test set (rare), call your split "train" and "dev," and be honest about not having a genuine, final test assessment.
    - However, for real-world machine learning systems—**always keep an untouched test set** for a fair evaluation before shipping/deployment.

Quick Recap ("Cheat Sheet"):

- Traditional splits (70/30, 60/20/20) are **outdated for big data.**
- Train/dev/test split for large data: **~98%/1%/1%** is reasonable.
- **Purpose determines size:**
    - Dev set = enough examples to reliably compare/tune models.
    - Test set = enough to confidently judge final system accuracy (often far less than 30% of data).
- **Never tune models using the final test set!**
- If a formal test set is omitted (rare), be clear and call it a train/dev split.

When to Change Dev/Test Sets and Metrics?

Cat dataset examples

Metric + Dev : Prefer A
You/users : Prefer B.

→ Metric: classification error

Algorithm A: 3% error → pornographic

✓ Algorithm B: 5% error

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

($y_{pred}^{(i)}$: predicted value (0/1))

$$\rightarrow w^{(i)} = \begin{cases} 1 & \text{if } x^{(i)} \text{ is non-porn} \\ 10 & \text{if } x^{(i)} \text{ is porn} \end{cases}$$

Orthogonalization for cat pictures: anti-porn

→ 1. So far we've only discussed how to define a metric to evaluate classifiers. ← Place target

→ 2. Worry separately about how to do well on this metric.

(An (shot at target)

$$\rightarrow J = \frac{1}{\sum w^{(i)}} \sum_{i=1}^{m} w^{(i)} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

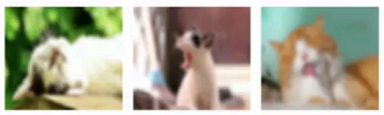
Another example

Algorithm A: 3% error

✓ Algorithm B: 5% error ←

→ Dev/test

→ User images



If doing well on your metric + dev/test set does not correspond to doing well on your application, change your metric and/or dev/test set.

- Your **dev set and evaluation metric** act like a target for your team—guiding what model behaviors you optimize for.

- Sometimes, you may realize halfway through a project that your current metric **does not truly reflect what matters most for your application**. In such cases, you should **change your dev/test sets or metric**.

Example:

- Imagine a cat classifier:
  - **Metric**: classification error.
  - Algorithm A: 3% error, but lets through many pornographic images.
  - Algorithm B: 5% error, but doesn't allow such images.
  - Even though A performs better on the metric, B is better for user experience. This means your metric isn't truly capturing what matters.
- **Key Insight**: If your evaluation metric prefers a model that's *actually worse* for the end goal (user/business), it's time to change your metric or dataset.

How to update the metric?

- For special cases (e.g., more serious errors), you could assign a **higher weight** to those cases in your error calculation.
- Example formula update: If mislabeling a pornographic image as a cat is a serious issue, increase its weight:
  - Regular images: weight = 1
  - Problematic images: weight = 10 or even 100
- Your new error = weighted sum of mistakes, normalized over total weights.

Practical Steps:

- Manually label key cases (e.g., problematic images) in your dev/test set to implement weighted metrics.

General Guideline:

- If your current metric and dataset do NOT reflect **real-world performance or what matters to your users/app**, then you should:
  - Change your **dev/test sets** (e.g., include more real-user photos, not just high-res web images).
  - Change your **metric** (e.g., add weights, use a different error type).

Orthogonalization Principle:

- *First*, define what matters (**set the target/metric**).
- *Then*, optimize your algorithm to hit that target.
- Don't try to solve both at once—treat as two steps: define a great metric, then optimize for it.

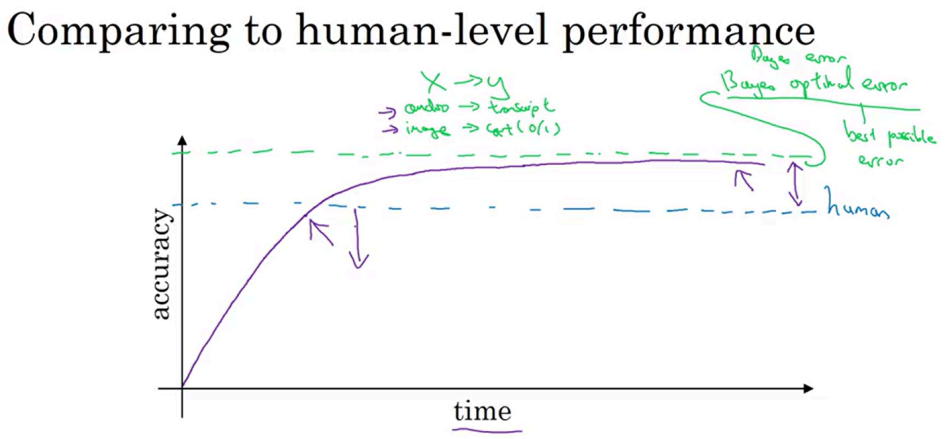
Advice for teams:

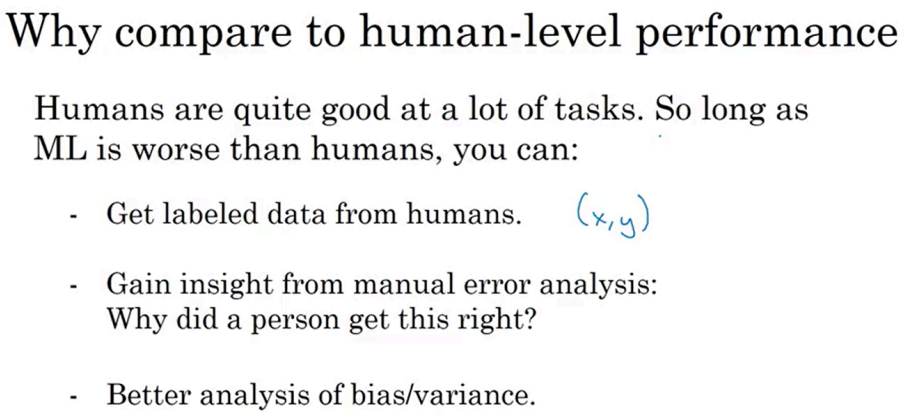
- It's better to have an "imperfect but useful" metric/dev set quickly, and iterate.
- Running too long without a clear metric/dev set makes progress slow and decision-making hard.
- **If you discover a better way to evaluate midway, change it—don't hesitate**.

Summary Guideline:

- *If success on your current metric/dev set doesn't translate into success on what you actually care about, update your metric and/or dataset to better reflect real needs and goals.*

Why Human-level Performance?





Why compare machine learning system performance to human-level performance?

- Two main reasons:
  - Recent advances (e.g., deep learning) mean algorithms can now actually match or exceed human-level performance on many tasks.
  - The design workflow is more efficient on tasks where humans can perform well—it's easier to mimic or compete with human performance.

Learning Curve Progress:

- When a research team starts working on an ML problem, progress is often *rapid* until reaching human-level performance.
- Once algorithms surpass human-level performance, further progress (accuracy improvement) often *slows down*.
- Model performance eventually approaches a theoretical upper bound, called the **Bayes optimal error** (the lowest error possible given the data due to inherent ambiguity/noise).

What is Bayes Optimal Error?

- Represents the best possible performance, given perfect algorithms and no labeling inconsistency.
- Some data is impossible for any system/human to classify correctly (due to noise, ambiguous images, poor audio, etc.).
- Even the best model can't achieve zero error if some information is inherently ambiguous or missing.

Why does progress slow after human-level performance?

- For many practical tasks (like image recognition, speech transcription), **human-level performance is already close to the Bayes optimal error**—there just isn't much room left for improvement.
- Also, as long as humans outperform your ML model, you can:
  - Get more high-quality labeled data from humans.
  - Perform manual error analysis (have humans review mistakes, compare with the model, and get insights).
  - Use humans to analyze bias and variance issues more effectively.
- Once your ML model exceeds human ability, the above techniques become hard or impossible to use.

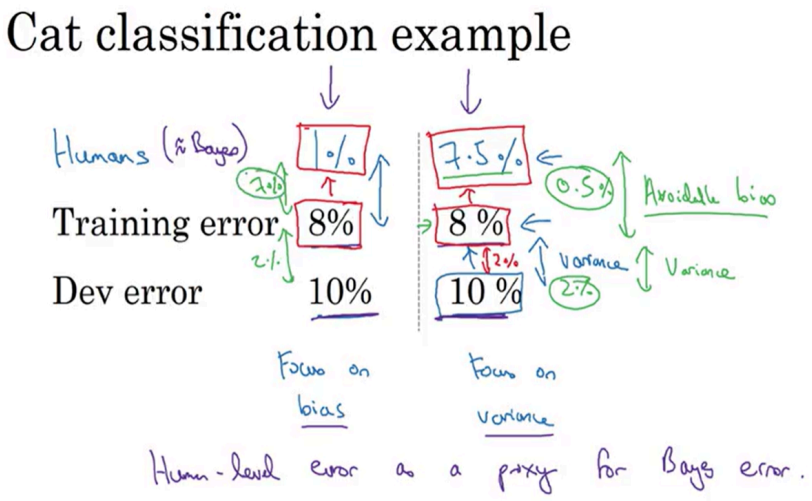
Practical Importance:

- Comparing against human-level performance gives you a *concrete benchmark* for progress.
- It helps determine if:
  - Your model needs better data/algorithms.
  - There are labeling or data quality issues.
  - You should focus on reducing bias or variance.

Summary Guideline:

- Human-level performance acts as a **reference point** in many ML projects—guiding progress, benchmarking, and decisions about next steps.
- Know the human performance limit for your task—aim for it, and use it to diagnose and improve your models.

Avoid Bias



Key Concept: Avoidable Bias

- **Human-level performance** serves as a reference or proxy for the best achievable accuracy on your problem (often approximates Bayes optimal error).
- **Avoidable bias** is the difference between (estimated) Bayes error (often human-level error) and your algorithm's training error.

Examples:

1. Suppose humans achieve 1% error on cat classification.
   - If your algorithm has 8% training error and 10% dev error:
     - The gap (7%) between human-level error (1%) and training error (8%) = **high avoidable bias**.
     - Focus on reducing bias: improve your model, train longer, or use bigger networks.
2. Suppose labeling is hard (e.g., blurry data), so human-level error is 7.5%.
   - Algorithm has 8% training error, 10% dev error:
     - Now, only 0.5% gap is **avoidable bias** (8% - 7.5%).
     - Focus more on **variance reduction** (closing the gap between training and dev—use regularization, more data, etc.).

Bias vs. Variance:

- **Avoidable bias**: Difference between human-level (Bayes) error and training error.
- **Variance**: Difference between training error and dev error.
- **Summary Table:**

Scenario	Human Error	Training Error	Dev Error	Avoidable Bias	Variance	Focus
Easy problem (low noise)	1%	8%	10%	7%	2%	Bias reduction
Hard problem (noisy/blurry data)	7.5%	8%	10%	0.5%	2%	Variance reduction

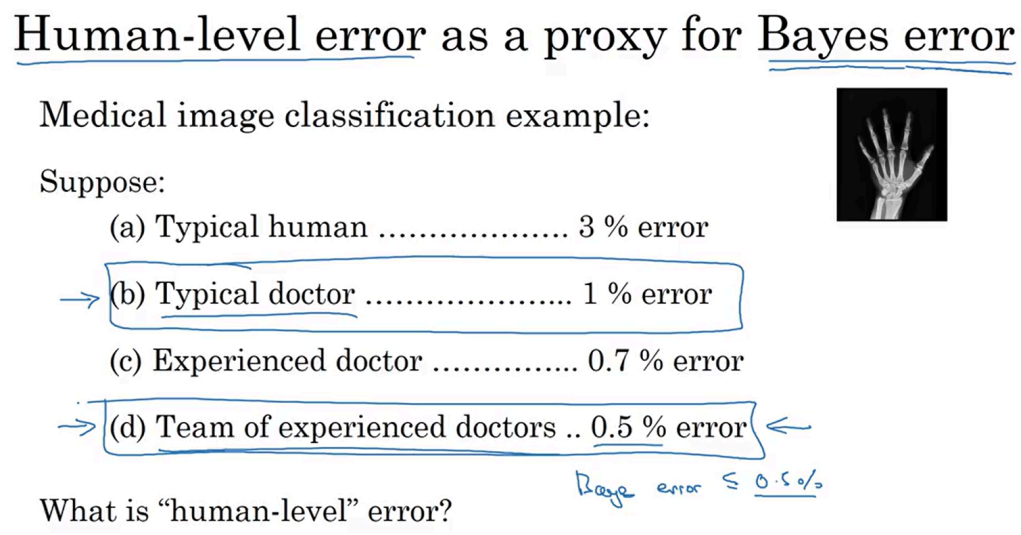
Practical advice:

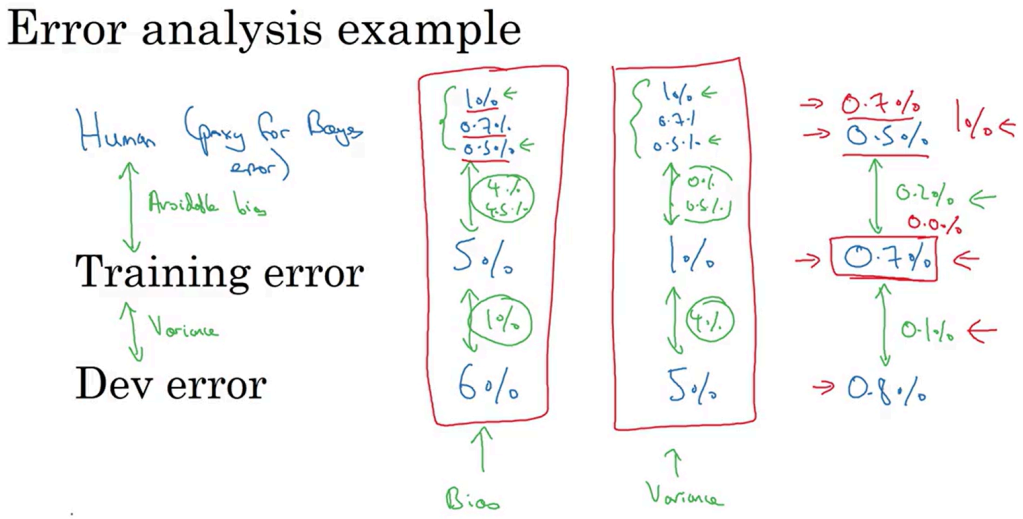
- **If avoidable bias is large:** Focus on making your model fit the training set better.
- **If variance is large:** Focus on closing the training-dev gap, e.g., regularization/more data.
- Don't aim for unrealistic accuracy—know what's possible with current data (Bayes/human-level error).

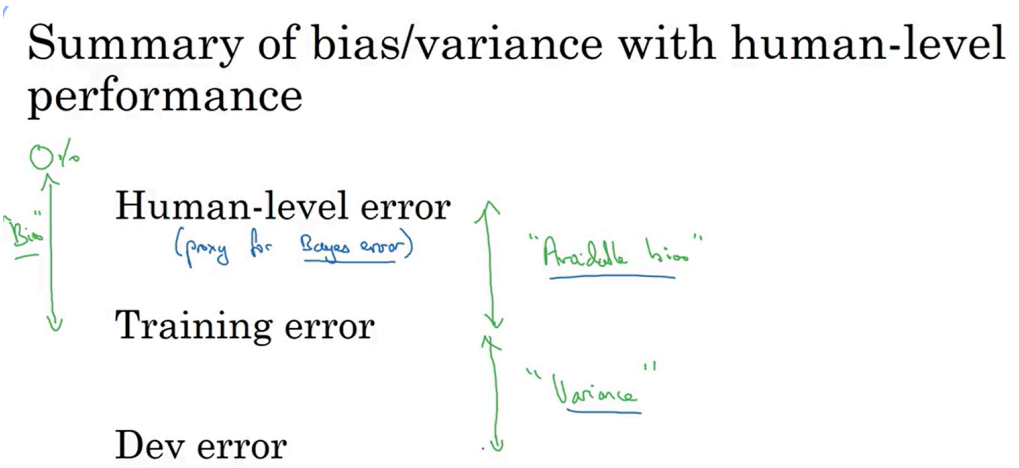
Bottom line:

Estimate human-level (or Bayes) error, then split your system error into avoidable bias and variance to choose the right improvement strategy.

Understanding Human-level Performance







What is Human-level Performance?

- The term *human-level performance* is often used in research, sometimes casually.
- It's helpful to define it precisely, especially for guiding **progress in machine learning projects**.
- One key use: Human-level error provides an estimate or proxy for **Bayes error** (the lowest possible error achievable, even by any system now or in the future).

Medical Imaging Example

- Suppose you want to classify medical radiology images:
  - **Untrained human:** 3% error
  - **Typical doctor (radiologist):** 1% error
  - **Experienced doctor:** 0.7% error
  - **Team of experienced doctors (consensus):** 0.5% error
- **Question:** Which of these defines "human-level error"? Is it 3%, 1%, 0.7%, or 0.5%?

Defining Human-level Error (for ML Analysis)

- The most useful definition: **Human-level error as an estimate or proxy for Bayes error**.
- If a team of doctors (consensus) achieves 0.5% error, the **Bayes error must be ≤ 0.5%**. There could be a better team, so the true Bayes error might be even lower, but not higher.
- For **bias and variance analysis**, use the best available human (or team) performance as human-level error.

Alternative Definitions (for Deployment or Publication)

- **Deployment/Publication context:** You may consider human-level error as the error of a single doctor, as surpassing this is practically useful (e.g., for justification to deploy a system).
- **Key takeaway:** Be clear on your *purpose* for defining human-level error:
  - For analysis (Bayes error estimate): Use the lowest (best) team performance.
  - For deployment argument: Surpass a typical doctor's performance.

Why Does This Matter? (Error Analysis Example)

- Suppose: Training error = 5%, Dev error = 6%, Human-level performance (Bayes proxy) = 0.5–1%.
- The **gap between Bayes error and training error = avoidable bias**.
- The **gap between training error and dev error = variance**.

Examples:

- If training error = 1%, dev error = 5%, human-level error = 0.5–1%:
  - Avoidable bias = small (000–0.50.50.5%)
  - Variance = large (4%)
  - **Action**: Focus on variance reduction (e.g., regularization, more data).
- If training error = 0.7%, dev error = 0.8%, human-level error = 0.5%:
  - Avoidable bias = 0.2%
  - Variance = 0.1%
  - Now, it's important to use the most accurate estimate of Bayes error, as you are close to optimal performance.

Difficulty Near Human-level Performance

- As you get close to human-level (or Bayes) error, it's harder to tell if you should focus on reducing **bias** or **variance**.
- When further from human-level, it's easier to decide which issue is dominant.
- Progress slows down as you approach human-level performance, as further improvements are challenging and depend on more subtle estimates of error.

Summary

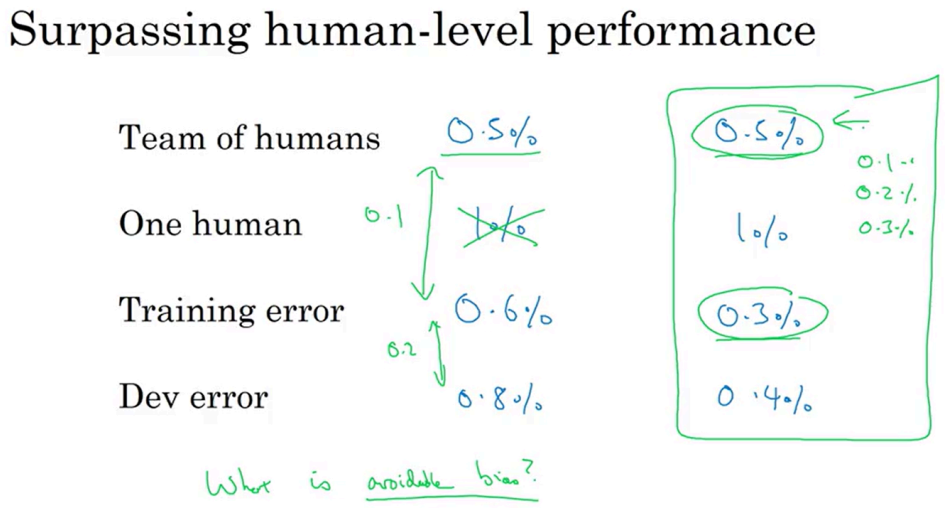
- **Human-level error** is a practical way to estimate Bayes error in tasks humans perform well.
- **Bias** = (training error) - (human-level error)
- **Variance** = (dev error) - (training error)

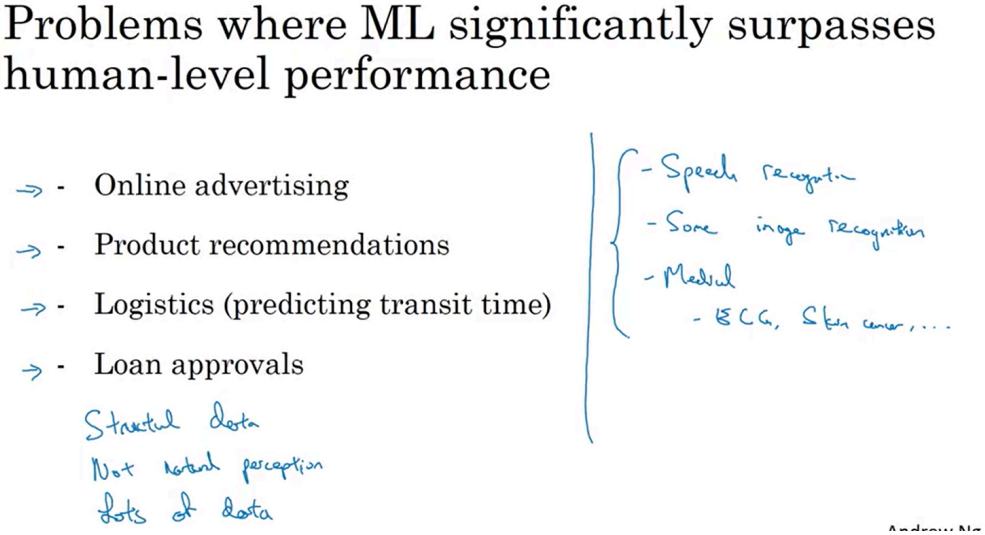
- Unlike earlier approaches (where bias was measured relative to zero), the new approach uses human-level error as a realistic benchmark.
- When data is noisy, **Bayes error will not be zero**, so human-level error gives a more accurate analysis.
- **Deciding whether to focus on bias or variance reduction** depends on these comparisons.

Final Points

- Having a good estimate of human-level performance (Bayes error proxy) helps you make smarter decisions to improve your model.
- For some tasks, modern ML already matches or surpasses human-level performance.
- The next lecture covers how to **surpass human-level performance**.

Surpassing Human-level performance





1. **Overview: Surpassing Human-level Performance**

- Many ML teams are excited to surpass human-level performance in classification tasks.

- As you get close to or exceed human-level performance, making progress becomes more challenging and less clear.

---

2. **Why Progress Gets Difficult Near/Superior to Human Performance**

- **Example:** If a team of humans (as a group) achieves 0.5% error, a single human achieves 1%, and your algorithm achieves:
  - 0.6% training error
  - 0.8% dev error
- Here, human-level (Bayes error) estimate is 0.5%.
  - Avoidable bias: at least 0.1% (0.6 - 0.5%)
  - Variance: 0.2% (0.8 - 0.6%)
- If you further improve and your algorithm gets 0.3% training and 0.4% dev error, it's much harder to know:
  - Is Bayes error 0.1%, 0.2%, or 0.3%?
  - Have you overfit, or are you approaching the irreducible error?
- When *your algorithm does better than even a team of humans*, it becomes difficult to determine whether you should reduce bias or variance.

---

3. **Loss of Human Intuition**

- When machines surpass humans, it's harder to use human intuition or expertise to improve further.
- Usual debugging methods (like analyzing errors by "becoming the expert") are less effective.

---

4. **Real-world Examples Where ML Surpasses Humans**

- Problems where ML already outperforms humans:
  - **Online advertising:** Predicting click probabilities.
  - **Product recommendations:** Suggesting movies, books.
  - **Logistics:** Predicting travel/delivery times.
  - **Credit scoring:** Predicting loan repayment.
- All these rely on *structured data* with large datasets — much more than any human could analyze.
- These are not typical *natural perception* tasks (vision, speech, language).

---

5. **Perception Tasks: Vision, Speech Are Harder**

- Humans excel at perception (image, audio, language).
- Computers can surpass human-level in some perception tasks (speech recognition, certain medical diagnostics, and some vision tasks), but it's harder and data-intensive.
- Recent advances in deep learning have enabled progress in these tasks, but surpassing humans is still a big achievement.

---

6. **Why Surpassing Humans Is Easier With Data**

- ML teams with access to massive data can spot statistical patterns invisible to humans.
- In highly-structured problems with abundant data, surpassing human performance is common.

---

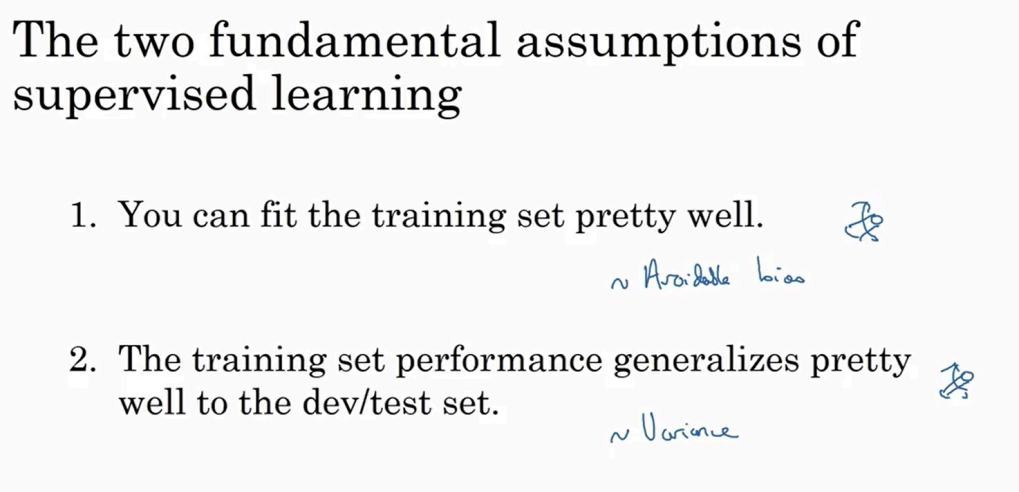
7. Key Takeaways

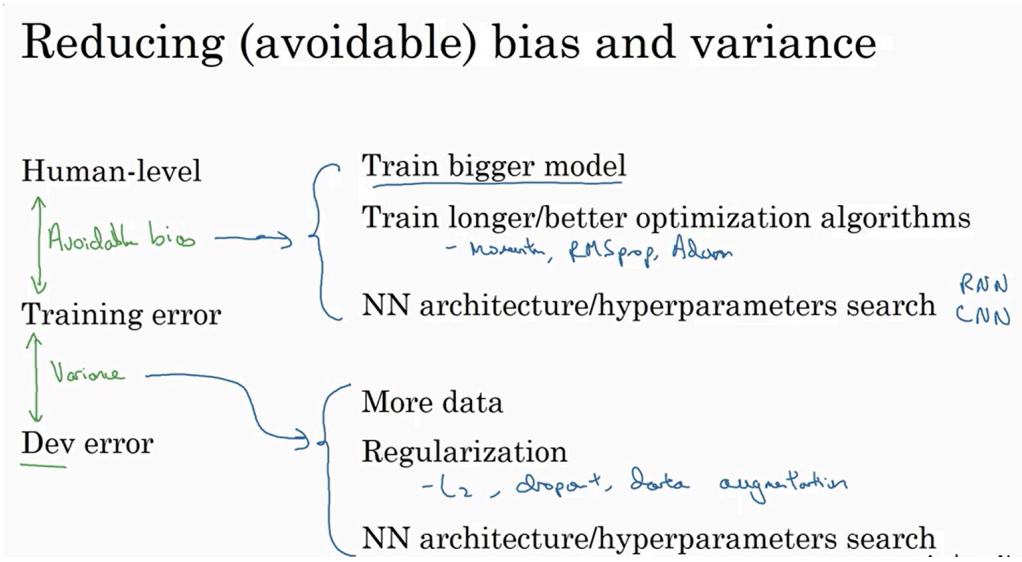
- Surpassing human-level performance **is possible but not easy**, especially for natural perception tasks.
- After surpassing humans, progress slows and measuring “how to improve” (bias vs. variance) becomes unclear.
- For best results, leverage lots of data and structured problems.
- Achievement of exceeding human-level on any perception-reliant task signals significant success.

Summary:

As ML systems approach and then surpass human-level performance, error analysis becomes ambiguous and human intuition is less useful. Successes are most common in structured-data problems with big data (recommendations, logistics, ads) but are becoming more frequent in perception tasks thanks to data and deep learning advancements. Continuous innovation and large datasets are critical to making further progress once human-level is beaten.coursera

Improving your Model Performance





1. Summary of Key Concepts

- This lecture pulls together core concepts: **orthogonalization**, setting up dev/test sets, using **human-level performance as a proxy for Bayes error**, and analyzing **avoidable bias and variance**.

---

2. Fundamentals for Good Supervised Learning Performance

- To make a supervised ML algorithm work well, you hope or assume two things:
  1. **Good Training Set Fit:** Achieve low error (low avoidable bias).
  2. **Good Generalization:** Low dev/test error compared to training (low variance).

---

3. Process for Improving Performance

- **Step 1:** Examine the gap between training error and your proxy for Bayes error (often human-level error).
  - This gap estimates your **avoidable bias**.
- **Step 2:** Examine the gap between dev set error and training error.
  - This gap estimates your **variance**.

---

4. How to Reduce Avoidable Bias

- Use methods that help your model fit the training data better:
  - **Train a bigger model** (with more capacity)
  - **Train longer**
  - **Use better optimization algorithms** (e.g., Adam, RMSProp)
  - **Find better neural network architectures or better hyperparameters** (e.g., change activation functions, number of layers, hidden units, etc.)

---

5. How to Reduce Variance

- Use methods to help your model generalize better from the training to dev/test data:
  - **Get more training data**
  - **Apply regularization** (e.g., L2 regularization, dropout, data augmentation)
  - **Experiment with network architectures/hyperparameters**
  - Try neural network architectures suited to your problem

---

6. Strategic Approach

- The concepts of avoidable bias and variance are "easy to learn but tough to master."
- Systematically and strategically applying these concepts helps you efficiently improve model performance—making your approach much more effective than random trial and error.

---

7. Key Takeaway

- By understanding and applying bias and variance analysis and the related improvement strategies, you will become a more strategic and systematic ML practitioner.

In short:

- Analyze your errors to decide whether to prioritize **reducing bias** or **reducing variance**.

- Apply appropriate techniques based on your analysis.
- Consistency and systematic application of these strategies drive meaningful progress in ML projects.

> 💡 *Personal note from Andrew: I've found practicing with scenarios like these to be useful for training PhD students and advanced Deep Learning researchers. This is the first time this type of "airplane simulator" for machine learning strategy has ever been made broadly available. I hope this helps you gain "real experience" with machine learning much faster than even full-time machine learning researchers typically do from work experience.*

Andrej Karpathy Interview

- **Early Exposure & Motivation**
  - First encountered deep learning at University of Toronto under Geoffrey Hinton, which inspired fascination in neural networks.
  - Initially unsatisfied with classic "artificial intelligence" and was drawn to the paradigm of machine learning, where optimization finds solutions given input-output pairs.
- **Human Benchmark for ImageNet**
  - Became a "reference human" for the ImageNet competition by manually labeling images to establish a baseline for human performance.
  - Created a JavaScript tool to classify thousands of images, noting how humans compared to deep learning models.
  - Discovered, for example, ImageNet's large bias toward dog species and highlighted the importance of measuring human-level performance as a benchmark.
- **Teaching & Community Impact**
  - Taught widely known deep learning classes (like CS231n), emphasizing deep, hands-on understanding over use of high-level frameworks.
  - Found teaching to be a highlight of his PhD and impactful for bringing the latest research into education.
- **Surprises in Deep Learning Evolution**
  - Was surprised by how convolutional neural networks (CNNs) scaled and by the power of transfer learning—using pretrained networks as general feature extractors.
  - Expected unsupervised learning to have more impact—supervised learning became dominant instead.
- **Perspectives on the Field & AGI**
  - Thinks AI will split into applied AI (task-specific, often with supervised learning) and artificial general intelligence (full agents able to reason more generally).
  - Believes breaking down intelligence into separate functions (vision, language, etc.) may be less fruitful than training unified, general neural networks.
- **Advice for Learners**
  - Strongly advocates understanding the "full stack": implement algorithms from scratch before using high-level libraries.

    - Encourages building your own neural network implementations (as he did with ConvNetJS) for genuine understanding and better problem-solving skills.
- **Long-term Vision**
    - Currently focused on questions around AGI at OpenAI, and how to create objectives that foster true intelligence in neural networks.

This interview is rich with career insights, technical reflections, and important advice for anyone pursuing a path in deep learning or AI.