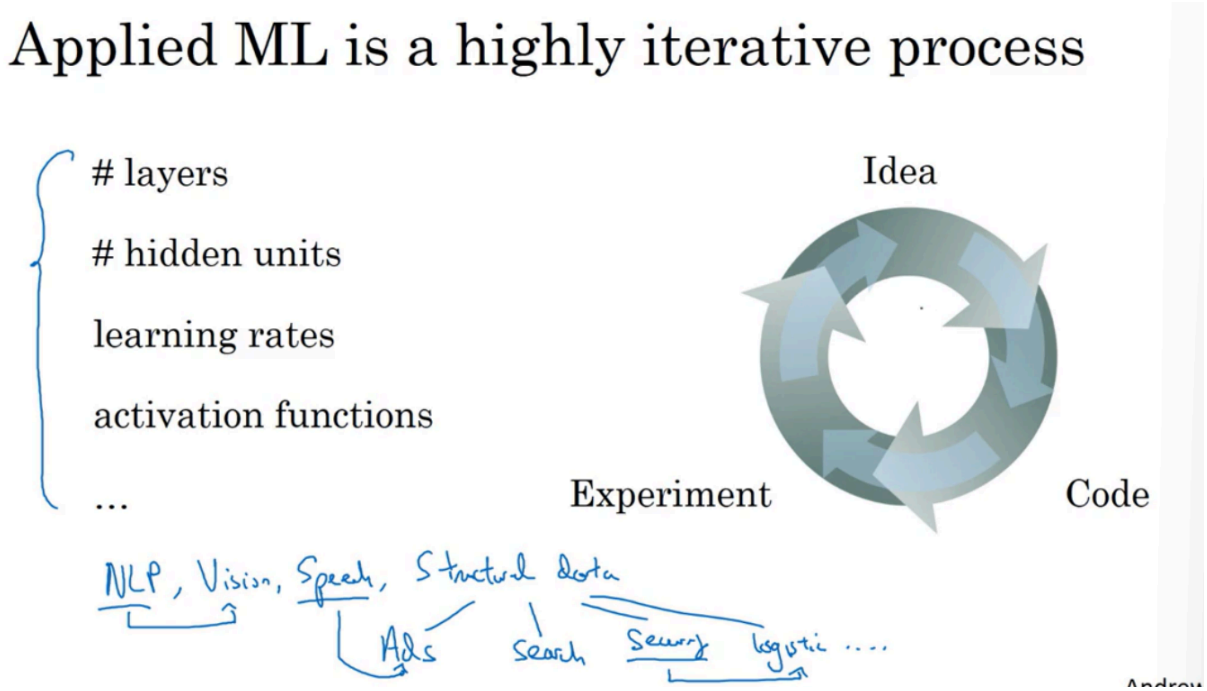
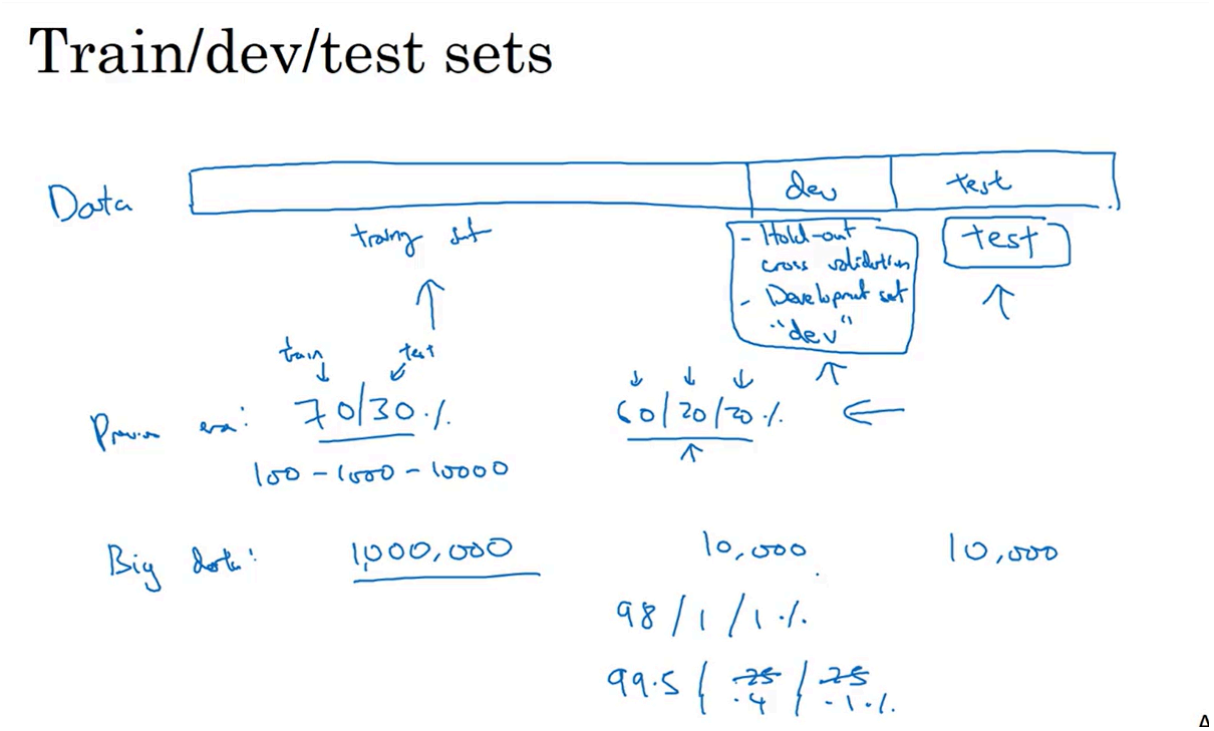
WEEK 1: Practical Aspects of Deep Learning

Train / Dev / Test sets

- Applied ML is a highly iterative process
- layers
- hidden units
- learning rates
- activation functions

...





1. What are Train / Dev / Test sets?

In machine learning & deep learning, we split our dataset into 3 parts to make sure the model **learns well** and also **generalizes to new data**.

- **Training set**
  - The data your model actually learns from.
  - Used to adjust weights/parameters through backpropagation + gradient descent.
  - Usually the largest portion of the dataset.
- **Development set (Dev set / Validation set)**
  - Used to **tune hyperparameters** (like learning rate, number of layers, hidden units, regularization strength, etc.).
  - Helps you decide between different models/architectures.
  - Not used in training, but in **model selection**.
- **Test set**
  - Used **only at the end** to estimate how well your final chosen model will perform on completely unseen data.

- Acts as a **proxy for real-world performance**.
- You should not go back and tune based on test set results, otherwise it "leaks" into training.

---

2. **Why split into three sets?**

- If you only have **Train + Test**, you may end up indirectly tuning your model to do well on the Test set → leading to **overfitting the test set**.
- Adding a **Dev set** prevents this problem, since you optimize on Dev and only check Test performance once at the end.

---

3. **Typical Split Ratios**

Depends on dataset size:

- **Large dataset (millions of examples)**
  - Train: 98%
  - Dev: 1%
  - Test: 1%
- **Medium dataset (hundreds of thousands)**
  - Train: 80%
  - Dev: 10%
  - Test: 10%
- **Small dataset (a few thousand)**
  - Train: 60%
  - Dev: 20%
  - Test: 20%

---

4. **Key Principles (Andrew Ng style)**

1. **Dev and Test sets should come from the same distribution**

   (e.g., if your application is self-driving cars in India, don't use US road data for Dev/Test).

2. **Your goal is to optimize performance on the Dev set**.

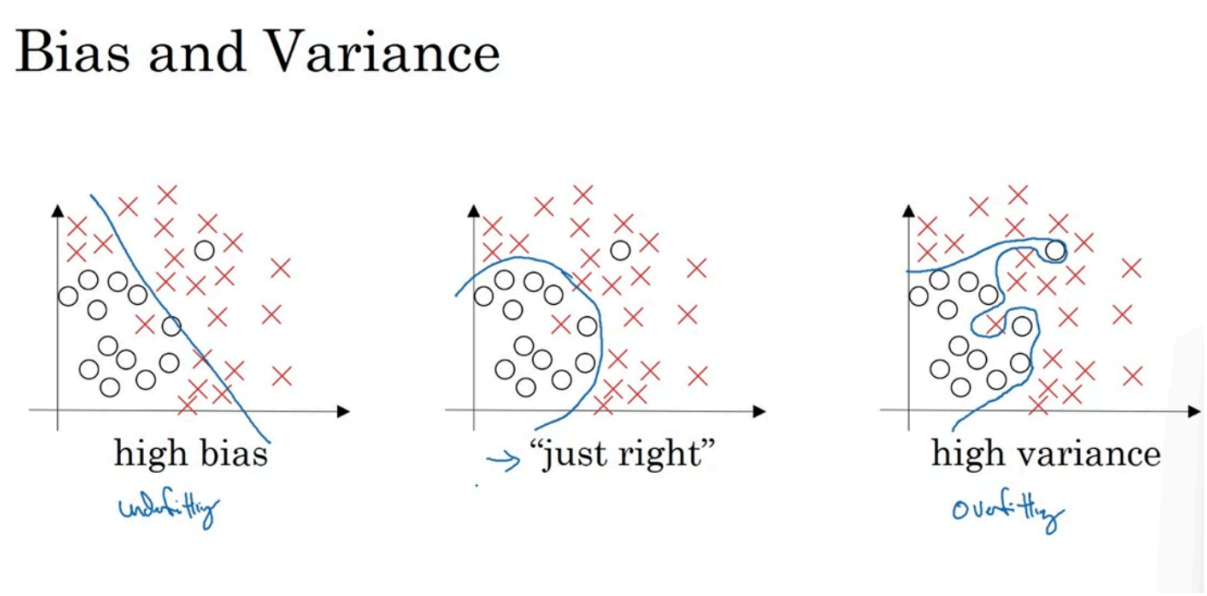
The test set is just to check how well you generalized.

3. **Avoid data leakage** (don't let test info slip into training).

✅ **Summary in one line:**

- **Train set → learn weights**
- **Dev set → choose model & hyperparameters**
- **Test set → final evaluation of generalization**

Bias / Variance



🔷 1. What is **Bias**?

- **Bias** = error from **wrong assumptions** in the learning algorithm.
- It's when the model is **too simple** to capture the underlying patterns.
- Example: Trying to fit a **straight line** to curved data.

👉 **High Bias → Underfitting**

- Both **Training error** and **Dev error** are high.

🔷 2. What is **Variance**?

- **Variance** = error from model being **too sensitive** to training data.

- The model learns noise instead of just patterns.
- Example: A very complex **high-degree polynomial** curve fitting just a few points.

👉 **High Variance → Overfitting**

- **Training error** is low, but **Dev/Test error** is high.

🔷 3. Bias-Variance **Trade-off**

- Traditionally, people said there's a trade-off:
  - If you reduce bias (make model more complex), variance increases.
  - If you reduce variance (simplify model), bias increases.

⚡ But Andrew Ng highlights in **modern deep learning**:

- With enough data and proper regularization, **you can often reduce both bias and variance**.
- That's why deep learning is so powerful!

🔷 4. Diagnosing with Train/Dev Error

Here's how you use **Train vs Dev set performance** to detect bias/variance:

Case	Training Error	Dev Error	Problem
High	High	High	**High Bias (Underfitting)**
Low	High	Much higher Dev error	**High Variance (Overfitting)**
High	Higher	Both high but gap exists	**High Bias + High Variance**
Low	Low	Low	✅ Good fit

Bias and Variance

Cat classification

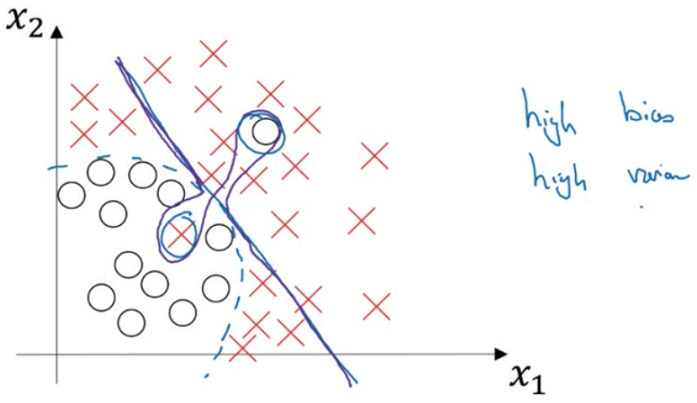
Train set error:	1%	15%	15%	0.5%
Dev set error:	11%	16%	30%	1%
	high variance	high bias	high bias & high variance	low bias low variance

Human: ~0%

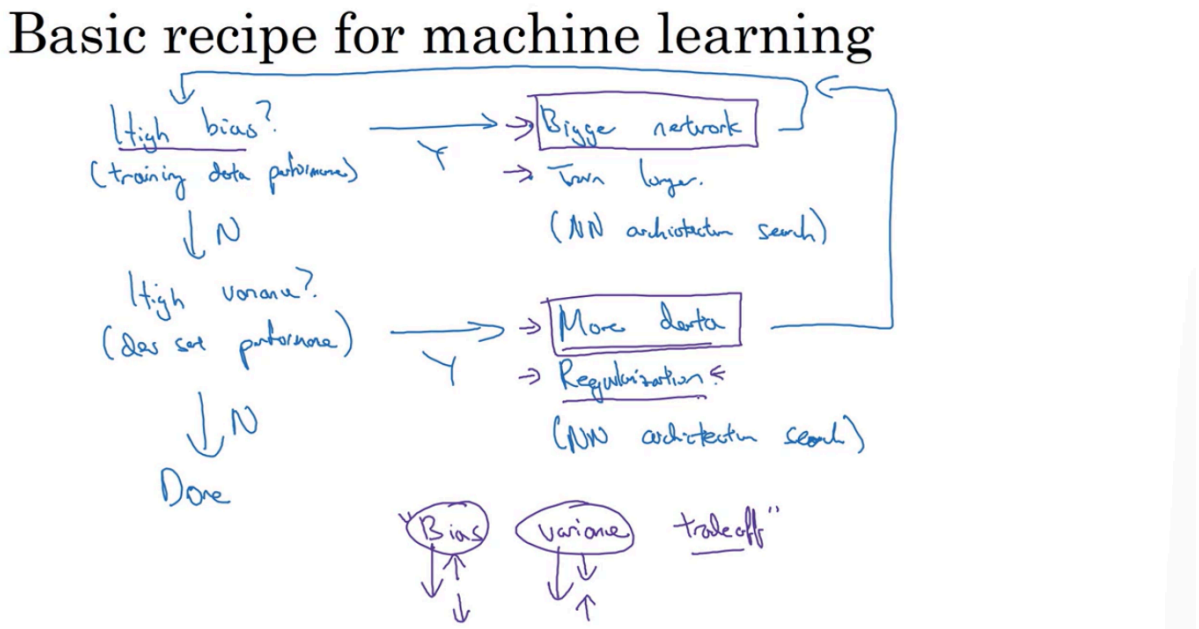
Optimal (Bayes) error: ~~0%~~ 15%

Blurry images

High bias and high variance



Basic Recipe for Machine Learning



- **High Bias (Underfitting)** → make model more powerful
  - Bigger network
  - Train longer
  - Better features/architecture
- **High Variance (Overfitting)** → improve generalization
  - More training data
  - Regularization (L2, dropout)
  - Data augmentation

✅ **In short**:

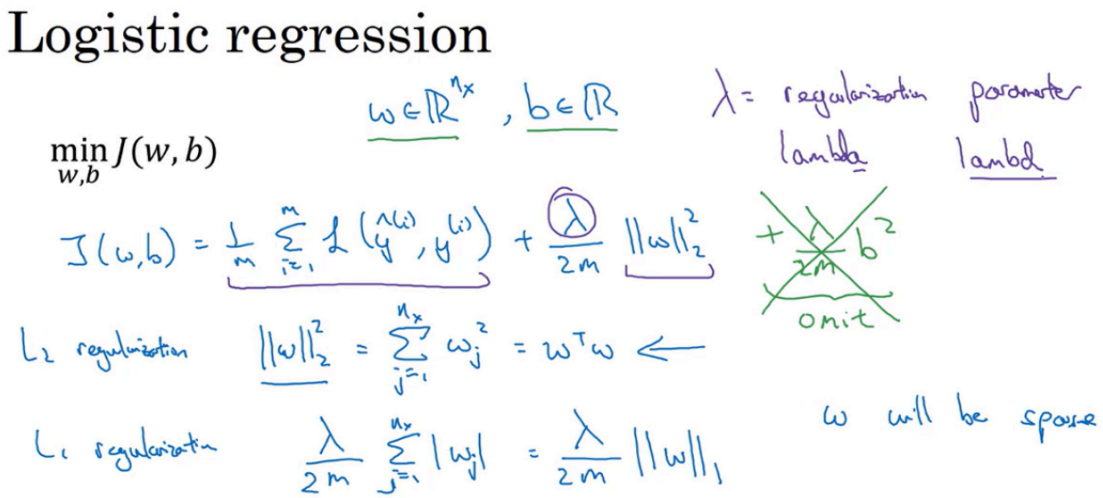
- **Bias** → model too simple → underfits.
- **Variance** → model too complex/sensitive → overfits.
- Deep learning often lets us **reduce both** by scaling data + models.

keep eye on this formula

Frobenius norm formula should be the following:

$$||w^{[l]}||^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{i,j}^{[l]})^2$$

Regularization



🔷 1. Why Regularization?

- In deep learning, big networks with many parameters can **overfit** (memorize training data instead of generalizing).
- **Regularization** is a set of techniques to **reduce overfitting (high variance)**.

🔷 2. L2 Regularization (Weight Decay)

Cost function without regularization:

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

With L2 regularization:

$$J = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^{L} ||W^{[l]}||^2$$

- $\lambda \rightarrow$ regularization parameter (controls strength of penalty).
- $||W^{[l]}||^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{i,j}^{[l]})^2$ (Frobenius norm squared).

👉 This **discourages large weights**. Large weights mean complex models → high variance.

🔷 3. How L2 Regularization Affects Gradient Descent

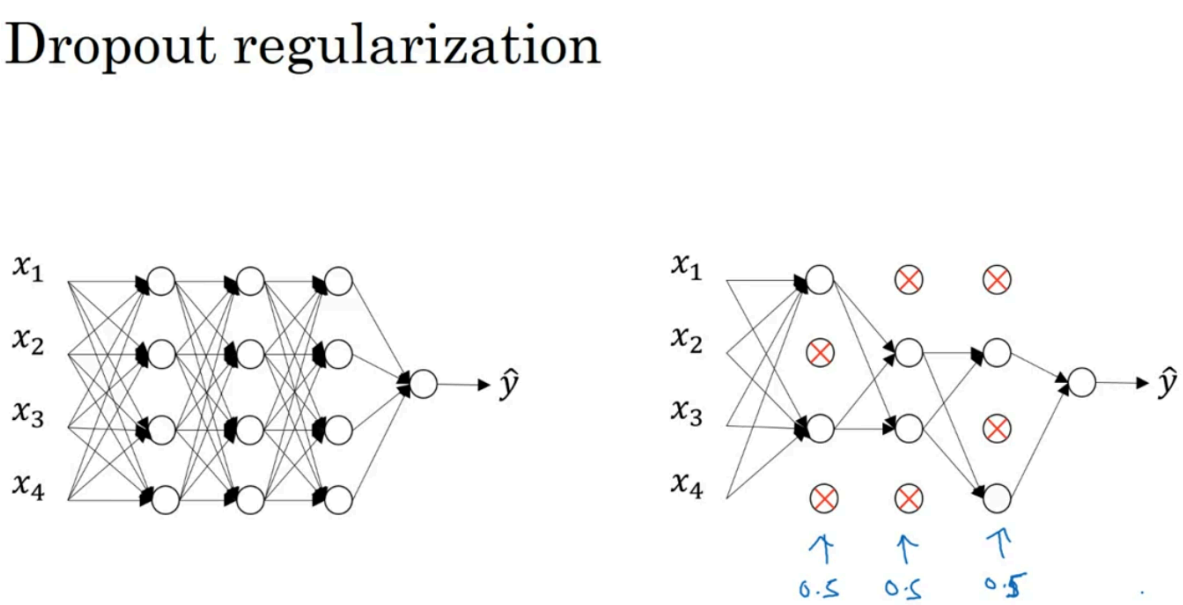
When computing gradients, the weight update rule becomes:

$$W^{[l]} := W^{[l]} - \alpha \left(dW^{[l]} + \frac{\lambda}{m} W^{[l]} \right)$$

- Extra term $\frac{\lambda}{m} W^{[l]}$ "shrinks" weights.
- So weights don't grow too large → smoother, less complex model.

- The factor $\left(1 - \alpha\frac{\lambda}{m}\right)$ is **slightly less than 1.**
- This means every update, the weights are **scaled down (shrunk)** a little, even before subtracting the gradient.
- So the weights **decay** over time → hence the name **weight decay.**

🔷 4. Dropout Regularization



- Randomly **"drop out" neurons** (set activations to 0) during training.
- Example: keep probability p=0.8 → each neuron has 80% chance of staying, 20% chance of being dropped.

Why it works:

- Prevents neurons from co-adapting too much.
- Forces network to spread learning across neurons.
- Works like training many smaller networks and averaging them.

🔷 5. Other Regularization Techniques

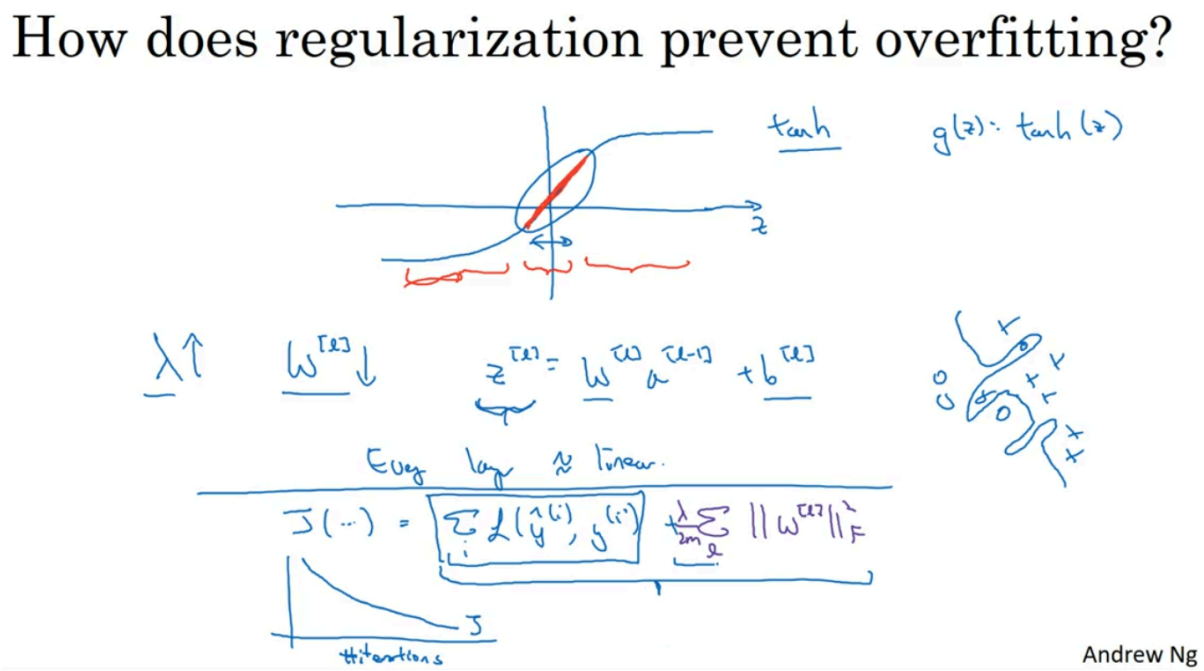
- **Data Augmentation** → enlarge training set with variations (rotations, flips, crops).
- **Early Stopping** → stop training when Dev error stops decreasing.
- **L1 Regularization** → adds $||W||_1$ instead of squared weights (leads to sparsity).
- **Max Norm / Gradient Clipping** (advanced, less used in basic course).

🔷 6. Key Intuition

- Regularization makes the hypothesis **simpler** and helps the model **generalize** better.
- If you have **high variance (overfitting)** → try regularization.
- If you have **high bias (underfitting)** → regularization won't help, you need a bigger model or better training.

✅ **Summary in one line:**

Regularization = "**don't let the network memorize the data** — keep it simple enough to generalize."



🔷 1. The Main Idea

When we apply **L2 regularization**, the regularization parameter λ increases → weights $W^{[l]}$ shrink (become smaller).

Smaller weights → smaller $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$.
So activations like **tanh(z)** or **sigmoid(z)** stay closer to the **linear region around 0**.

🔷 2. Effect on Activation Functions (Top diagram)

- The **tanh curve** is drawn.
- The **red oval** highlights the **linear region near 0**.
- If weights are small → z stays small → activations operate in this **linear region**.

👉 This makes the network behave **more like a linear/logistic regression model**, which is **simpler** and **less likely to overfit**.

🔷 3. Effect on Decision Boundaries (Right diagram)

- Without regularization: weights can grow large → very **wiggly/complex decision boundaries** that fit noise.
- With regularization: weights are smaller → boundaries are **smoother and simpler**, capturing the true structure, not the noise.

👉 This prevents overfitting by **simplifying the decision boundary**.

🔷 4. Cost Function with Regularization (Middle bottom)

$$J = \frac{1}{m}\sum_{i} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l} ||W^{[l]}||^2$$

- The first term = original loss.
- The second term = **regularization penalty** (Frobenius norm of weights).
- This penalty encourages smaller weights.

🔷 5. Training Curve (Bottom left)

- The plot shows cos J decreasing with iterations.
- Even with regularization, the algorithm still learns.
- But the weights are shrunk each step (weight decay), leading to smoother convergence.

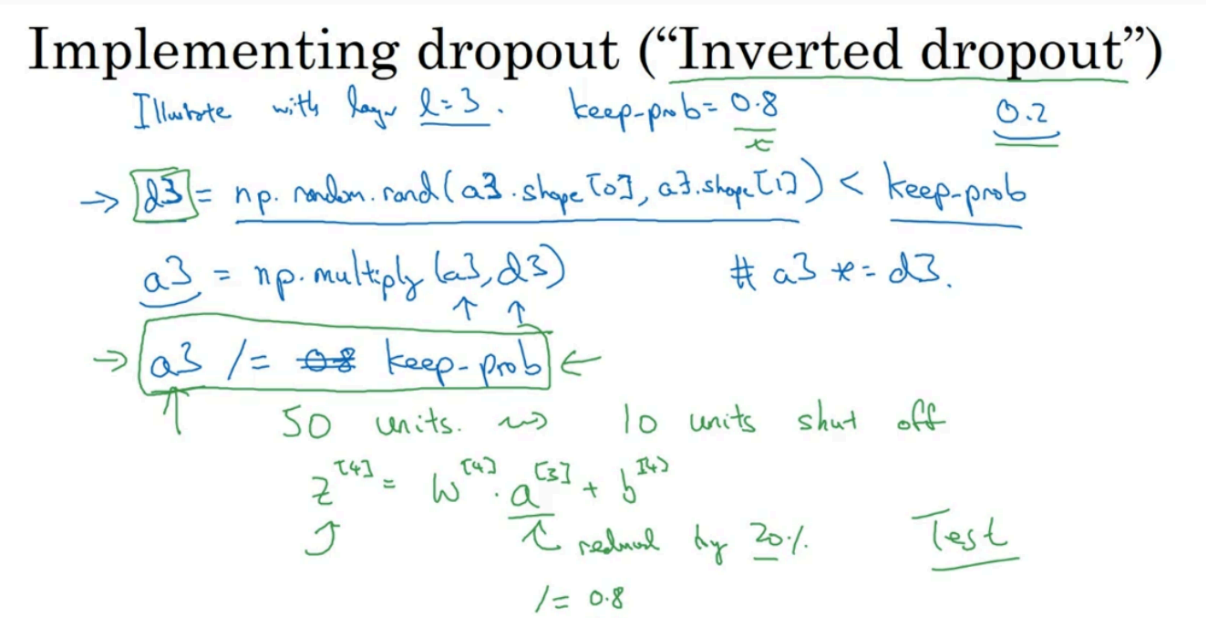
🔷 6. Summary

- If λ is too large → weights shrink too much → **underfitting (high bias)**.
- If λ is too small → little effect → **overfitting (high variance)**.
- With the right λ, regularization makes the neural network **simpler, more linear, smoother decision boundaries** → prevents overfitting while still fitting well.

✅ **One-line takeaway (Andrew Ng style):**

Regularization works by keeping weights small, which keeps activations in the linear region, making the network simpler and less likely to overfit.

Dropout Regularization



🔷 1. What is Dropout?

Dropout is a **regularization technique** where, during training, we randomly turn off (set to 0) some neurons in each layer with a certain probability.

- Example: If a layer has **50 units** and we use **keep-prob = 0.8**, then on average 10 units (20%) will be shut off in each forward pass.

- This prevents neurons from **relying too much on each other**, forcing the network to learn more robust features.

🔷 2. Step-by-Step Implementation (shown in the image)

(a) Generate dropout mask

$$d^{[3]} = \text{np.random.rand}(a^{[3]}.shape[0], a^{[3]}.shape[1]) < \text{keep-prob}$$

- `np.random.rand` creates random numbers between 0 and 1.
- Compare each random number with `keep-prob` (e.g., 0.8).
- If less → neuron is kept (1), else → neuron is dropped (0).
- So `d3` is a binary **mask matrix** (same shape as activations $a^{[3]}$.

👉 Example: if `keep-prob = 0.8`, ~80% of neurons are kept, ~20% are shut off.

(b) Apply mask to activations

$$a^{[3]} = a^{[3]} \times d^{[3]}$$

- Multiplying by the mask → dropped neurons become **0**, kept neurons stay as they are.
- Effectively shuts off 20% of neurons here.

(c) Scale activations (Inverted Dropout)

$$a^{[3]} = \frac{a^{[3]}}{\text{keep-prob}}$$

- After dropout, the **expected value of activations decreases** (since some are zeroed).
- To fix this, we divide by keep-prob (e.g., divide by 0.8).
- This rescales activations so their expected value stays the same.

👉 This is why it's called **Inverted Dropout** — we scale during training, so at test time we don't need to scale anything.

🔷 3. Example from the diagram

- Suppose layer 3 has **50 units**.
- With `keep-prob = 0.8`, about **10 units are shut off randomly**.
- So only ~40 units remain active per forward pass.
- The activations that survive are scaled up by $1/0.8 = 1.25$, so their average remains correct.

🔷 4. Why Dropout Works

- Each mini-batch sees a **different random subset of neurons**.
- Forces the network to learn redundant representations (neurons can't "co-adapt" too strongly).
- Acts like training an **ensemble of many smaller networks**, then averaging them at test time.
- Helps reduce **overfitting** significantly.

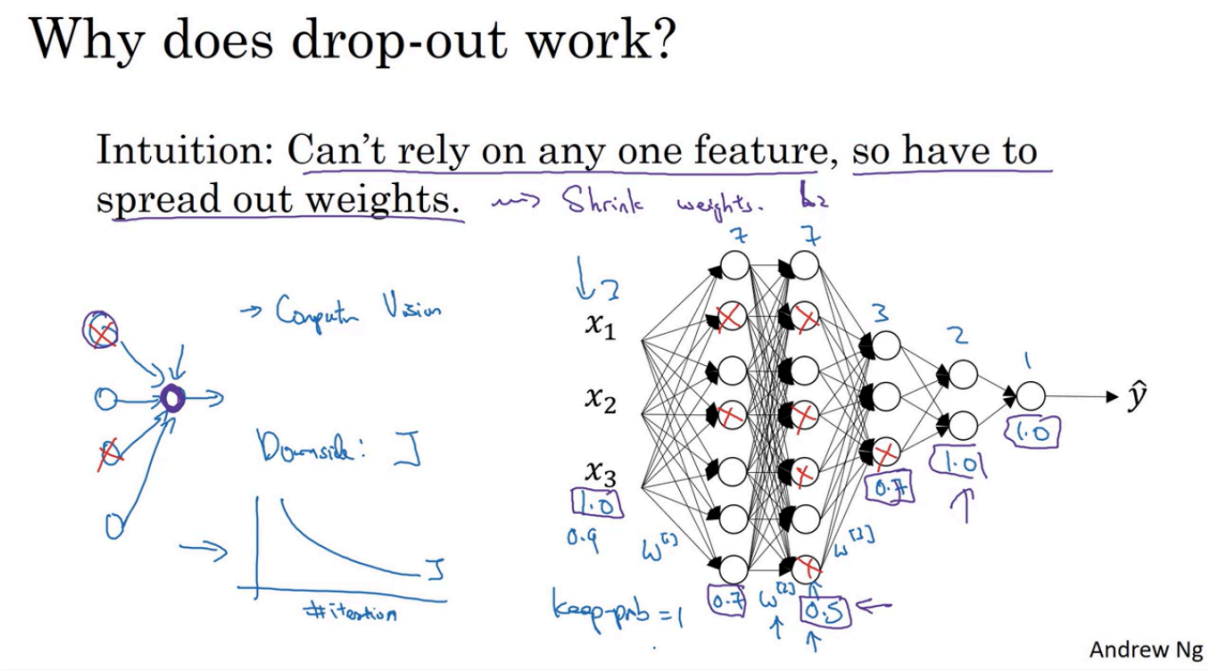
🔷 5. Key Point (from Andrew Ng)

- **Training phase** → Apply dropout + scaling (as shown).
- **Testing phase** → Do NOT apply dropout. Just use the full network as-is.

This ensures predictions are stable at test time.

✅ **One-line Andrew Ng style summary:**

Dropout randomly turns off neurons during training and rescales the rest, preventing over-reliance on specific neurons and making the network more robust against overfitting.



Why does dropout work?

- **Intuition:**

  If neurons know they might get randomly "dropped" (turned off), the network **can't rely on any single feature**.

  👉 It must **spread out the weights** across many neurons.

- **Effect:**
  - This prevents some neurons from becoming too dominant.
  - Encourages the network to learn more **robust, distributed representations**.

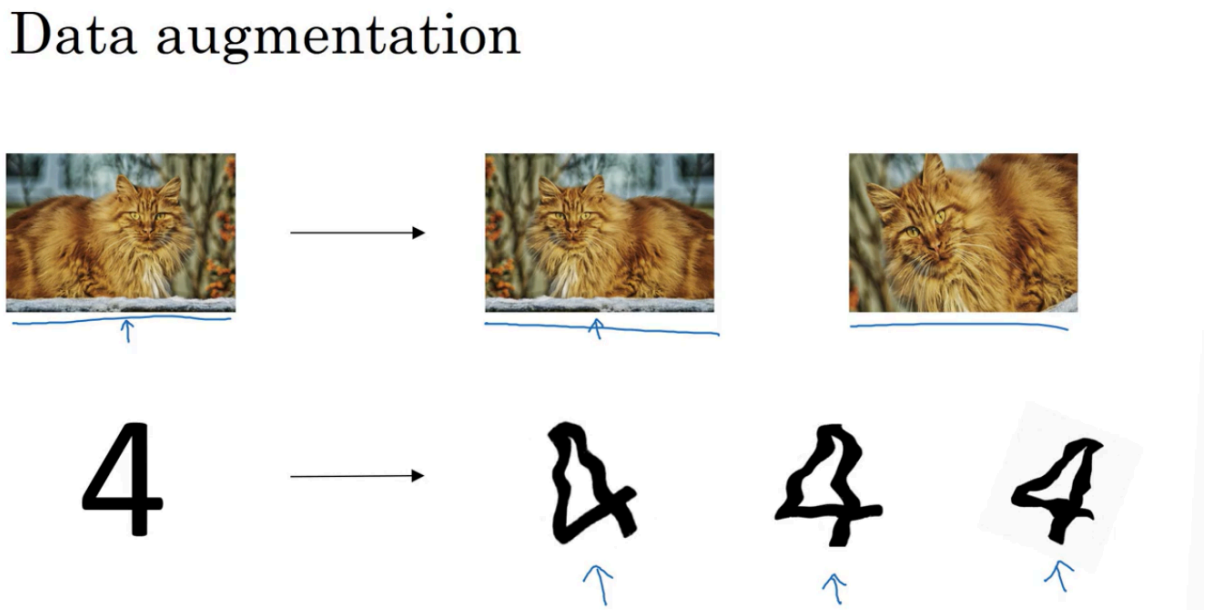
- Works like **regularization (similar to L2 shrinkage)**.

- **Example in the diagram:**
  - Keep probability (keep_prob) varies (e.g., 0.5, 0.7, 1.0).
  - Neurons with red crosses are “dropped” during training.
  - This forces the others to pick up the slack and generalize better.
- **Downside:**
  - Slows convergence (cost J decreases more slowly).
  - But **final generalization is much better**.

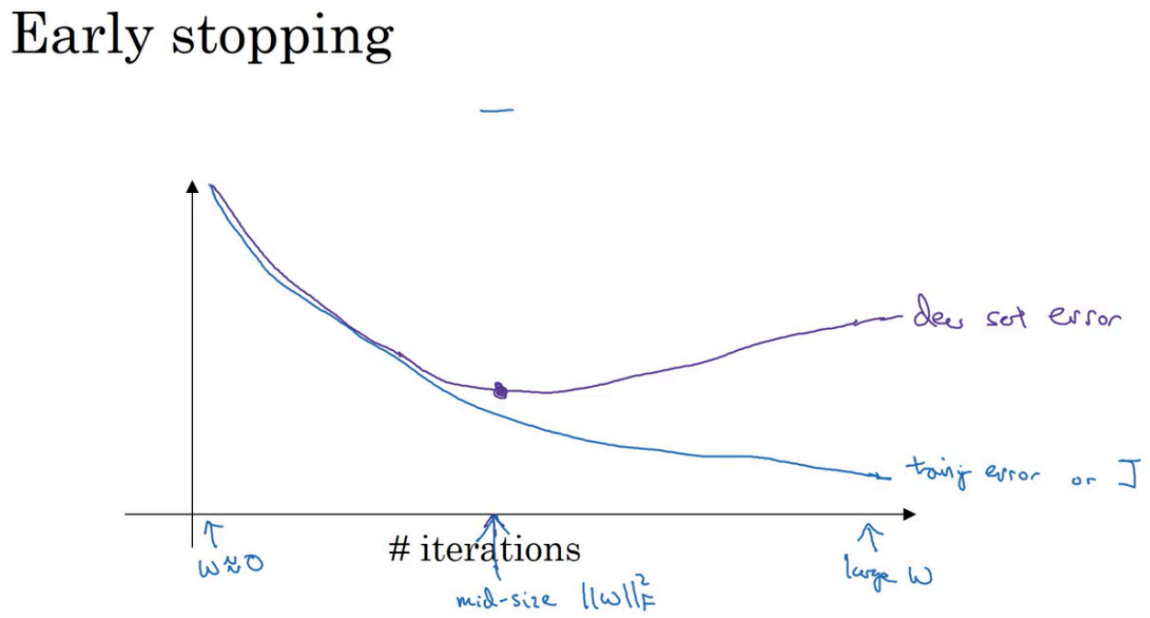
👉 In short: **Dropout reduces overfitting by preventing co-adaptation of neurons and spreading weights across the network.**

Other Regularization Methods

1. Data augmentation



2. Early stopping



🌟 Early Stopping

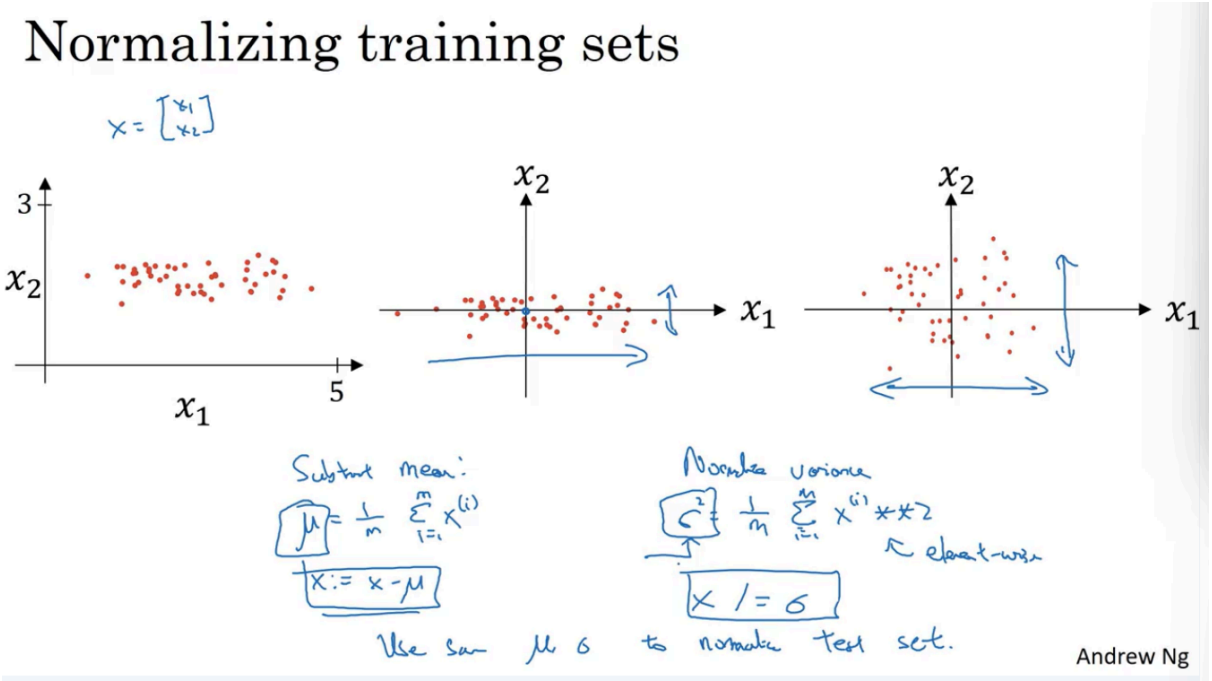
- **Idea**: Stop training when validation error starts increasing, even if training error keeps decreasing.
- **Why?** Prevents overfitting by not letting the model "memorize" noise.
- **How?**
  1. Split into training & validation sets.
  2. Train model → monitor validation error.
  3. When validation error increases (after decreasing), stop training.

🔑 Key Points

- **Bias-Variance tradeoff**: Stops at the sweet spot (low bias, low variance).
- **Alternative to L2**: Works like regularization by preventing large weights.
- **Downside**: Training process becomes dependent on validation set monitoring.

👉 One-liner: *Early Stopping is a form of regularization that halts training before the model starts overfitting.*

Normalizing Inputs



Perfect buddy 🙌 Let's make **Notion-style notes** for this image (Normalizing Training Sets).

📊 Normalizing Training Sets

📌 Why Normalize?

- Speeds up training ⚡.
- Ensures features are on a **similar scale** → avoids one feature dominating.
- Makes gradient descent converge faster.

🧮 Steps

1. **Subtract the Mean** (Centering)

$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$

$x := x - \mu$

- Shifts data to be centered around **0**.

---

2. **Normalize the Variance** (Scaling)

$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)})^2$

$x := \frac{x}{\sigma}$

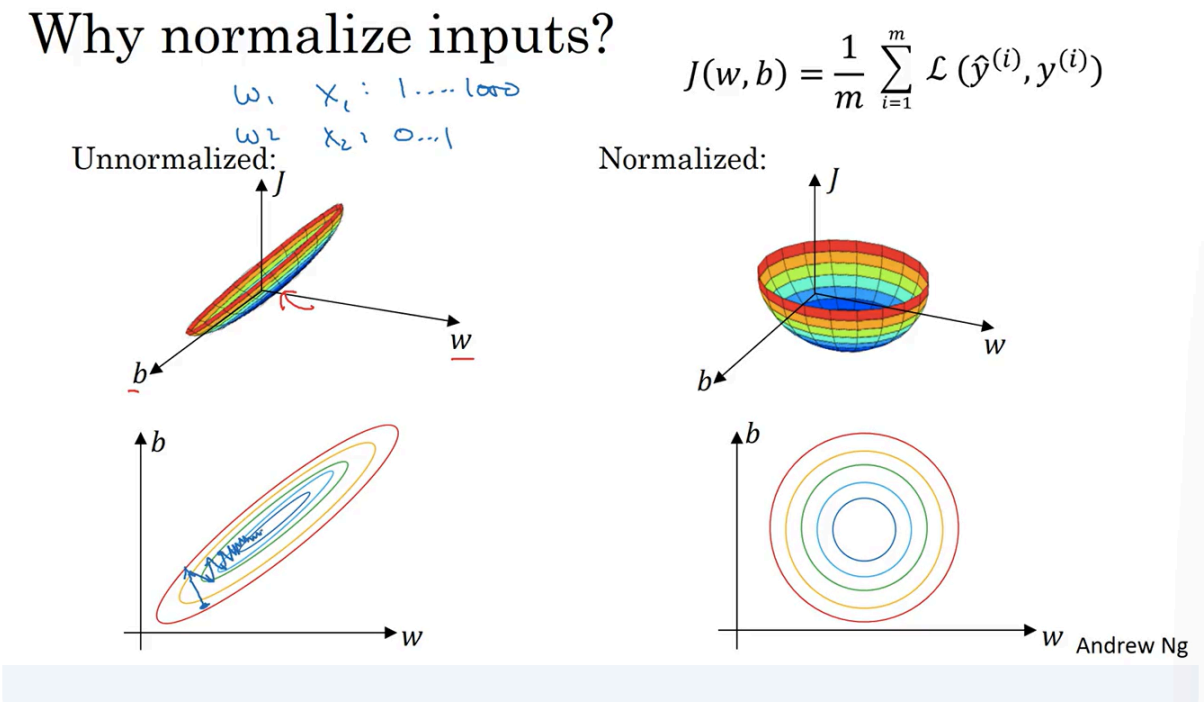
- Scales features so that variance ≈ **1**.

✅ Final Effect

- Input features spread evenly around **0**, with **unit variance**.
- Test set must also use **same μ and σ** from training set for consistency.

⚖️ **In short:**

- Step 1: Center data (mean = 0).
- Step 2: Scale data (variance = 1).
- Helps gradient descent converge **much faster** 🚀.



3d representations above images

contor lines below images

⚖️ Why Normalize Inputs?

📌 Intuition

- If features are on **very different scales** (e.g., $x_1 \in [1, 1000], x_2 \in [0, 1]$),

  → cost function contours become **elongated ellipses**.

  → Gradient Descent takes a **zig-zag path** → **slow convergence** ⏳.
- After **normalization**:

  → features on **similar scales** → cost contours become **circles**.

  → Gradient Descent can take **direct, efficient steps** 🚀.

🧮 Cost Function

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

- Without normalization → “stretched bowl” shape.
- With normalization → “round bowl” shape.

🔍 Visualization

- **Unnormalized (Left):**
  - Cost contours are **elliptical**.
  - GD path is **zig-zag & slow**.
- **Normalized (Right):**
  - Cost contours are **circular**.
  - GD converges **faster & smoother**.

✅ Key Takeaway

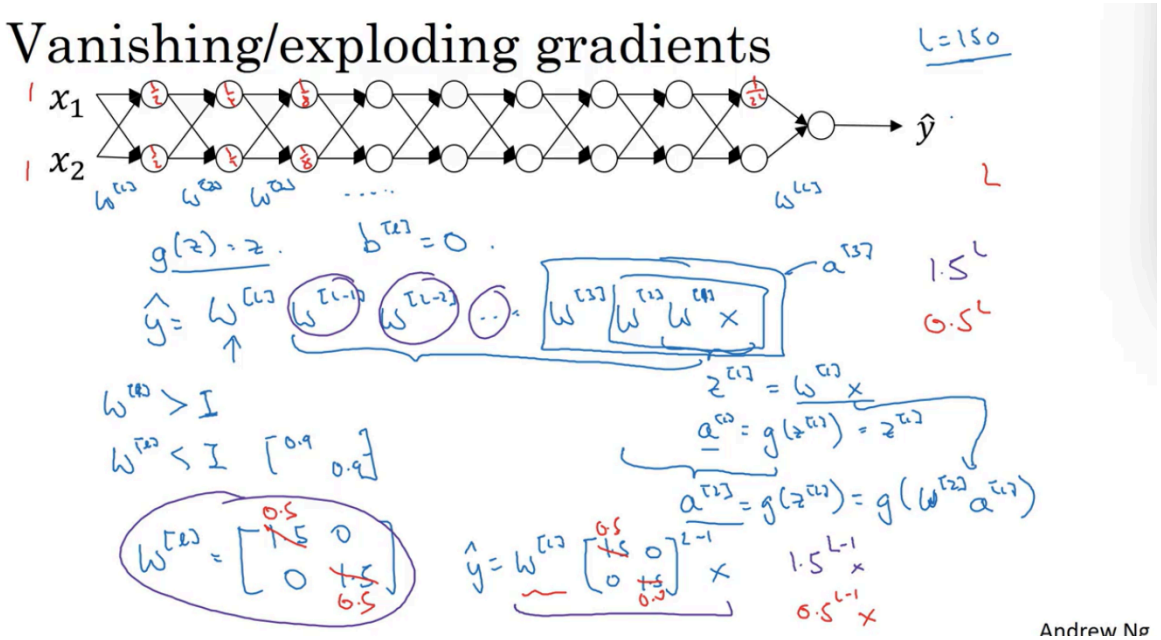
- Normalization makes optimization **faster & more stable**.

- Always normalize features before training a model.

⚡ **In short:**

Normalize → transforms cost function from a **stretched ellipse** to a **circle**, making gradient descent converge much faster.

Vanishing / Exploding Gradients



- In **deep neural networks** (e.g., with L=150L = 150 layers as shown), gradients can become extremely **small (vanish)** or **large (explode)** during backpropagation.
- This happens because gradients are propagated by repeatedly multiplying by weight matrices across layers.

Key Idea in the Image

1. **Linear Activation Example**
    - To simplify, Andrew Ng assumes activation g(z)=zg(z) = z (linear identity function).

- The network output:
  $\hat{y} = W^{[L]}W^{[L-1]} \dots W^{[1]}x$

2. **Effect of Weights**
   - If weights WW are **greater than 1** (e.g., 1.5), repeated multiplication gives:
     $(1.5)^L$
     → grows exponentially (**exploding gradients**).
   - If weights WW are **less than 1** (e.g., 0.5), repeated multiplication gives:
     $(0.5)^L$

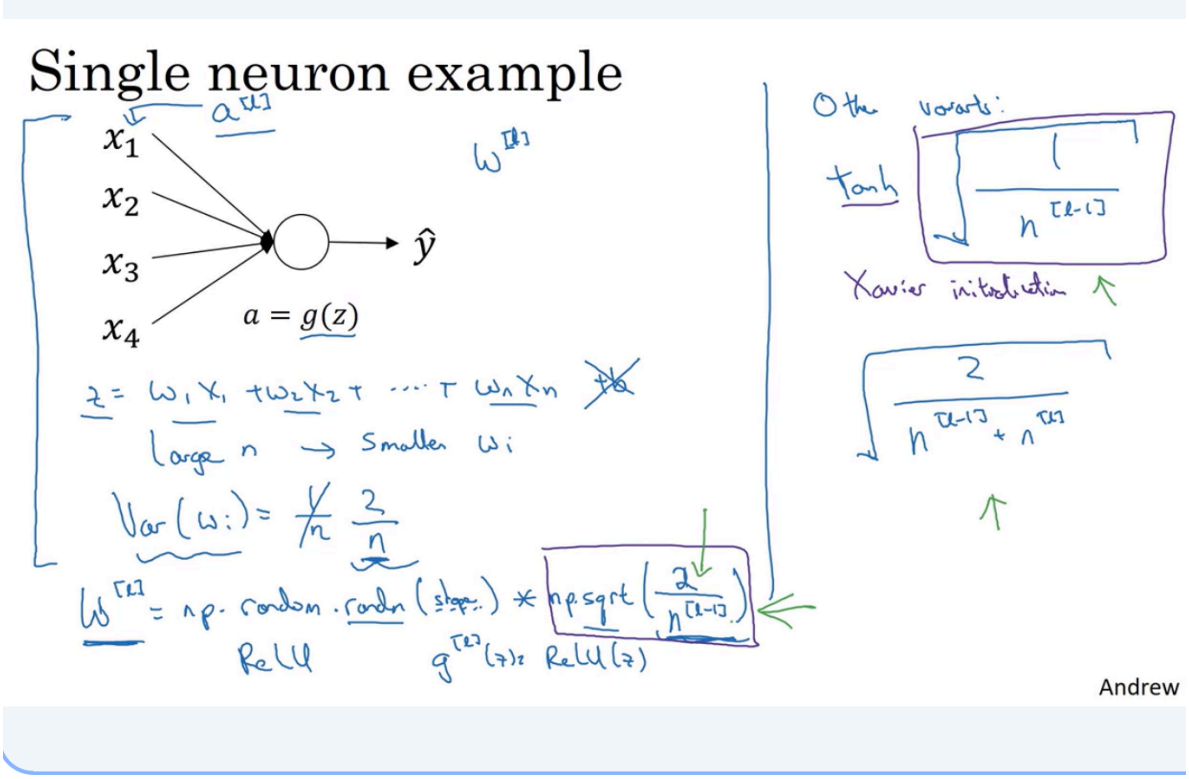
→ shrinks exponentially (**vanishing gradients**).

1. **Example Matrix**
   - A simple diagonal weight matrix is used:
     $W = [1.5000.5] W = \begin{bmatrix} 1.5 & 0 \\ 0 & 0.5 \end{bmatrix}$
   - After many layers, one component grows as 1.5^L, while the other shrinks as 0.5^L .

Conclusion

- **Vanishing gradients** → network learns very slowly, weights stop updating.
- **Exploding gradients** → unstable training, weights blow up.
- This is why techniques like **better weight initialization (Xavier, He)**, **Batch Normalization**, and **careful activation choices (ReLU instead of sigmoid/tanh)** are used.

Weight Initialization for Deep Networks



Report an issue

Weight Initialization for Deep Networks

Save note

Single Neuron Example

We have one neuron with inputs:

x_1, x_2, x_3, x_4

Neuron computes:

$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$

$a = g(z)$

Problem

- If the number of inputs n is **large**, and we pick **random weights** w_i without scaling,

  then z can become **too large** or **too small**.

- This leads to **vanishing/exploding activations** across layers.

Variance Scaling Idea

We want the variance of z to stay stable.

If w_i are random variables:

$Var(w_i) = \frac{1}{n}$

or sometimes:

$Var(w_i) = \frac{2}{n}$

This ensures activations don't grow or vanish as they pass through layers.

Initialization Rules

- For **ReLU** activations (He initialization):

$w^{[l]} \sim \mathcal{N}\left(0, \frac{2}{n^{[l-1]}}\right)$

- For **tanh / sigmoid** activations (Xavier initialization):

$w^{[l]} \sim \mathcal{N}\left(0, \frac{1}{n^{[l-1]}}\right)$

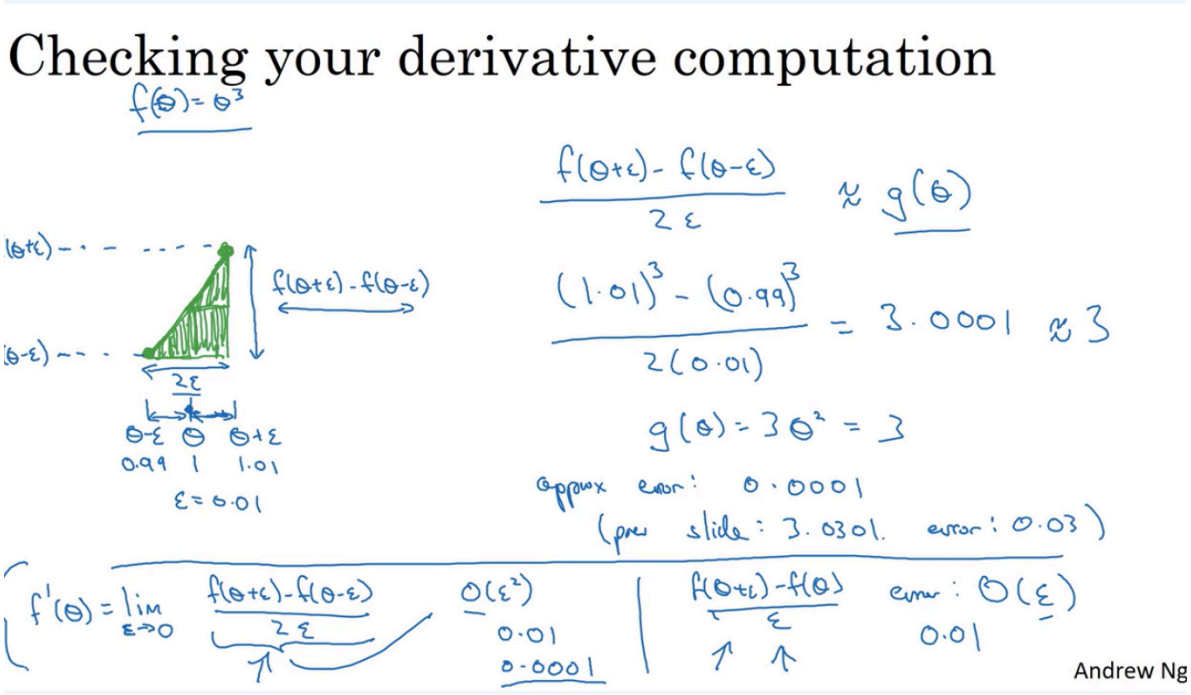
or more generally:

$w^{[l]} \sim \mathcal{N}\left(0, \frac{2}{n^{[l-1]}+n^{[l]}}\right)$

Andrew Ng's Key Insight

- The goal is to **keep variance of activations and gradients stable across layers**.
- If weights are too big → exploding activations.
- If weights are too small → vanishing activations.
- Good initialization (He, Xavier) helps networks train much faster.

Numerical Approximation of Gradients



1. Setup

We want to check if our backprop derivative is correct.

Take a simple function:

$f(\theta) = \theta^3$

At $\theta = 1\theta = 1$, the true derivative is:

$f'(\theta) = 3\theta^2 = 3$

---

2. Numerical Approximation (Central Difference)

We approximate the derivative with a small ϵ:

$f'(\theta) \approx \frac{f(\theta+\epsilon) - f(\theta-\epsilon)}{2\epsilon}$

Example: with $\theta = 1$ and $\epsilon = 0.01$:

$\frac{(1.01)^3 - (0.99)^3}{2(0.01)} = \frac{1.030301 - 0.970299}{0.02} = 3.0001$

Which is very close to the true derivative 3.

---

3. Error Analysis

- Using **central difference** (above), the error is order:

$O(\epsilon^2)$

For $\epsilon = 0.01$, error ≈ 0.0001.

- If we had used **forward difference**:

$\frac{f(\theta+\epsilon)-f(\theta)}{\epsilon}$

Error would be **much larger**, order:

$O(\epsilon)$

For $\epsilon = 0.01, error \approx 0.01$.

---

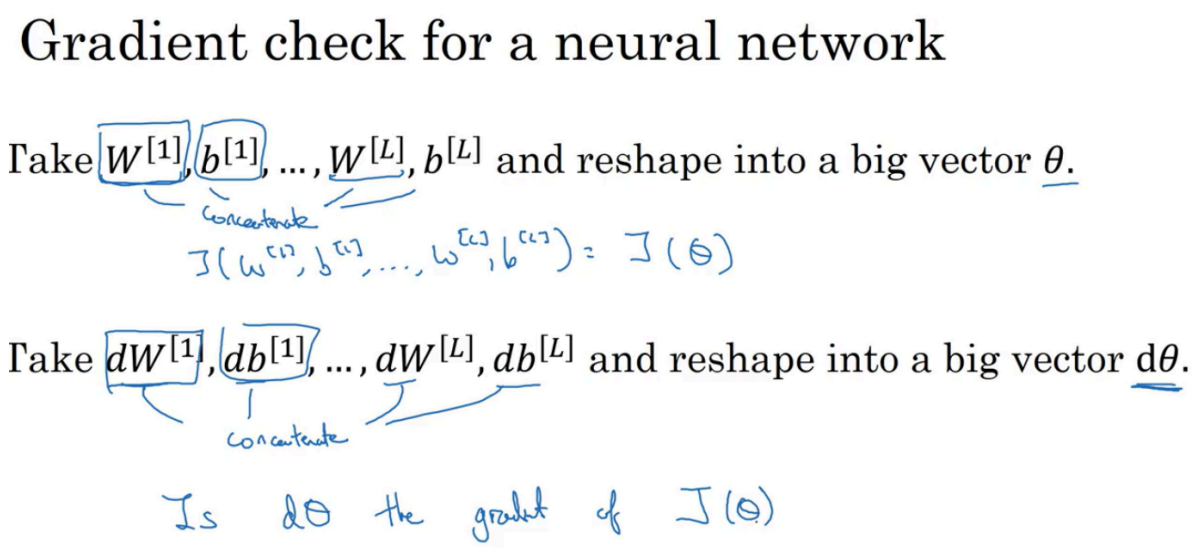
4. Andrew Ng's Key Insight

- Use **central difference** for gradient checking (much more accurate).
- Choose ϵ small enough (like 10^{-4}) but not too small (to avoid floating-point issues).
- Gradient checking is only used for **debugging**, not in actual training.

✨ That's the intuition Andrew Ng is teaching here:

👉 Numerical gradient checking helps us **verify backprop**, and central difference is the best choice.

Gradient Checking



Numerical gradient approximation:

$$\frac{\partial J}{\partial \theta_i} \approx \frac{J(\theta_1, \ldots, \theta_i + \epsilon, \ldots, \theta_n) - J(\theta_1, \ldots, \theta_i - \epsilon, \ldots, \theta_n)}{2\epsilon}$$

Compact vector form:

$$\nabla_\theta J(\theta) \approx \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon}$$

Comparison check

$$d\theta \overset{?}{\approx} \nabla_\theta J(\theta)$$

Gradient check ratio:

$$\text{difference} = \frac{\|d\theta_{\text{approx}} - d\theta\|_2}{\|d\theta_{\text{approx}}\|_2 + \|d\theta\|_2}$$

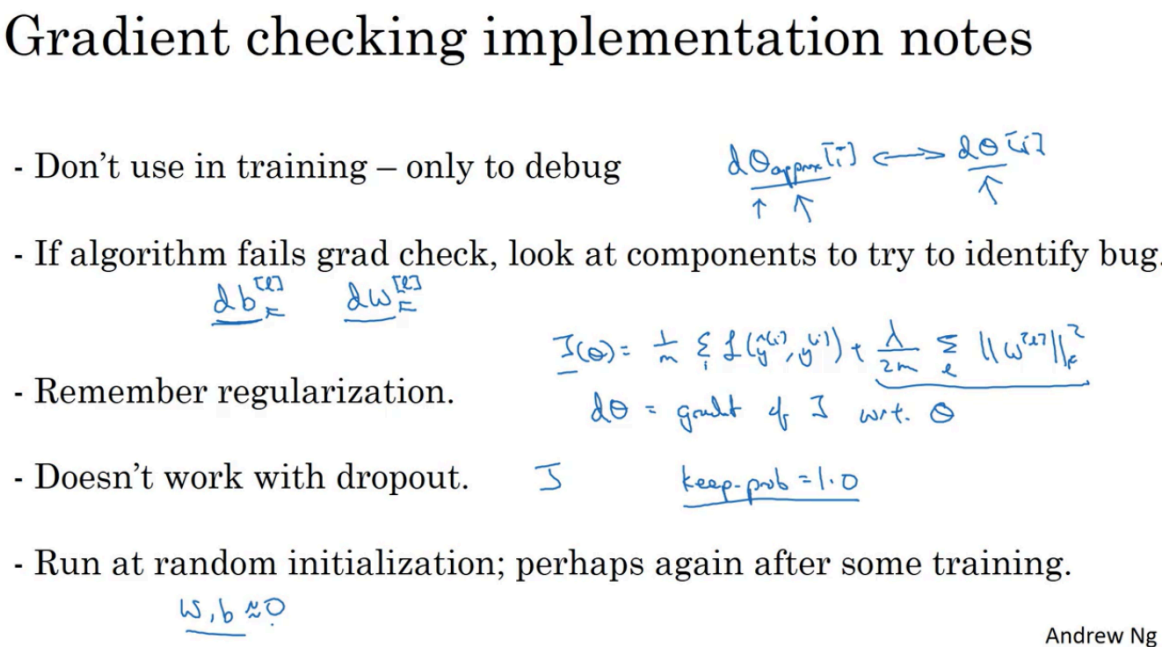
Where:

- dθ = gradient from backprop
- $d\theta_{\text{approx}}$ = gradient from numerical approximation

If this $difference \leq 10^{-7} \rightarrow backprop$ is correct ✅

If it's larger → bug in implementation 🐞

Gradient Checking Implementation Notes



Graded Assignment

⚡ Key Takeaways

- **Small datasets** → 60/20/20 split
- **Dev & Test sets** → same distribution
- **High variance** → more data, regularization
- **High bias** → bigger model
- **Dropout** → reduces variance
- **λ (L2)** ↑ → more regularization
- **keep_prob** ↓ → more regularization
- **Normalization** → center & scale inputs

Yoshua Bengio Interview

Summary

This is an interview with deep learning pioneer Yoshua Bengio. He recounts his journey into the field, starting from reading science fiction and neural net papers in the mid-1980s. He discusses the evolution of his thinking, from early

work on distributed representations and tackling the curse of dimensionality to his group's foundational work on deep learning (e.g., word embeddings, RBMs, autoencoders, GANs, attention mechanisms).

Bengio reflects on surprises (like the effectiveness of ReLU), mistakes (believing smooth nonlinearities were essential), and the ongoing inspiration he draws from neuroscience and biological plausibility. The conversation shifts to the future, where he expresses most excitement for unsupervised and reinforcement learning as paths to machines that can truly understand the world through observation and interaction. He emphasizes the importance of "science" in deep learning—asking *why* things work—and advises newcomers to focus on first principles and hands-on implementation rather than just using high-level frameworks.

Important Points

1. Historical Journey & Key Contributions

- **Motivation:** Entered the field in 1985, inspired by science fiction and the connectionist idea of distributed representation (vs. symbolic AI).
- **Foundational Work:** His research group is behind many pillars of modern deep learning:
    - Studying **vanishing gradients** and **long-term dependencies**.
    - **Word embeddings** (from work on tackling the curse of dimensionality with distributed representations).
    - **Stacks of Autoencoders and RBMs** for deep learning.
    - **Denoising Autoencoders**, **GANs** (Generative Adversarial Networks), and the **Attention mechanism** (crucial for modern machine translation).

2. Evolution of Thinking & Surprises

- **Biggest Mistake:** Believing smooth nonlinearities (like sigmoids) were necessary for training. The success of **ReLU** was a major surprise.
- **Shift in Perspective:** Attention mechanisms changed his view of neural nets from simple vector-to-vector mappers to systems that can handle complex data structures.

3. Relationship to Neuroscience & Biology

- **Core Inspiration:** The brain's **distributed representation** of information remains a foundational insight.
- **Current Research Interest:** Exploring how the brain could implement something like **backpropagation**, seeking biologically plausible credit assignment methods inspired by Jeff Hinton's earlier ideas.

4. The Future: Unsupervised & Reinforcement Learning

- **Critical Limitation:** Current AI relies on supervised learning and human-labeled data.
- **Grand Goal: Unsupervised learning** is key to building AI that can, like a child, discover how the world works through observation and interaction (**intuitive physics**).
- **Most Exciting Direction:** Combining unsupervised learning with **reinforcement learning**, where agents learn by exploring and trying to control their environment. This is seen as the path to true understanding and common sense.

5. Philosophy of Research & Advice

- **"Toy Problems":** Advocates for working on simple, controllable problems to quickly understand failures and accelerate the research cycle.
- **The "Science" of Deep Learning:** Stresses the importance of asking **"why"** algorithms work, not just benchmarking performance. The goal is deeper understanding, not just better results.
- **Advice for Newcomers:**
    - **Don't just use frameworks:** Implement algorithms from scratch to truly understand them.
    - **Derive from first principles:** This builds a much stronger foundation.
    - **Required Background:** A solid base in **computer science, probability, linear algebra, calculus, and optimization** is essential.
    - **It's accessible:** With the right background, one can become proficient in a matter of months.