Practical Aspects of Deep Learning | Train | Devl | Test

I. The core philosophy: Applied ML is an iterative process.
The key to success in a real-world deep learning projects is recognizing that it's a highly iterative cycle. It's almost impossible to guess the hyperparameters on the first try.

The work flow is:

1. Idea: Start with a model architecture and hyper parameters
2. Code: Implement your idea
3. Experiment: Run the code and get results
4. Refine: Analyze the results, refine your idea, and repeat the cycle

Goal: Make this cycle as fast and efficient as possible. The most crucial step for efficiency is how you set up your data.

II. Setting Up Your Data: Train, Dev, and Test sets
Properly splitting your data is critical for measuring progress.

- Train set: The data your model learns from.
- Development (Dev) set: Used to tune hyperparameters. You test many differences on the dev set to see which performs best
- Test set: Used only once at the very end to get a final, unbiased performance score of your chosen model.

III. Data Split Ratios: The Shift from Traditional to "Big Data" Era
The recommended splits have changed as datasets have grown.

A. Traditional Era (Smaller Datasets: < 100k examples)

- Common splits were 70/30 or 60/20/20
- These ratios are fine for smaller amounts of data

B. Big Data Era (Larger Datasets: > 1M examples)

- The Dev and Test sets can be a much smaller percentage of the total data. Reasoning: The absolute number of examples is more important than the percentage.
- The Dev set only needs to be larger enough to reliably compare two different models

- The test set only need to be large enough to give a confident estimate of your final models performance.
- Example splits for 1,000,000 examples
  - Train: 980,000 (98%)
  - Dev: 10,000 (1%)
  - Test: 10,000 (1%)

IV: Handling Mismatched Data Distributions

Increasingly common for your training data to come from a diff source

- Scenario: You build a cat-photo app.
  - Training set: High-resolution, professional cat pictures downloaded from the internet
  - Dev/Test sets: Blurry, lower-resolution cat pictures uploaded by your app's users
- The most important rule of thumb:
  Make sure your Dev and Test sets come from the same distribution.

Why: Its acceptable to have miss matched training sets.

V. Is a Test set always always necessary?

Answer: No * purpose of test set is to provide a final, unbiased performance estimate

- If you don't need perfectly unbiased estimate

Basic Recipe for machine Learning

I love comsci
I love comsci

After training initial model...

High Bias? (training data performance)
- Bigger network
- Train longer
(NN architectures search)

↓ N

High variance? (dev set performance)
- More data
- ☆ Regularization
(NN architecture search)

↓

Done

"Bias Variance Trade Off"

↕ ↓ — pre deep learning era did not have many

Regularization

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

Why regularization?

- prevents overfitting, reduces model complexity without needing more data
- Common in logistic regression and neural networks

L2 ~~Cost function~~ Regularization (Weight Decay)

Cost function (Logistic Regression)

- $\|w\|^2 = \text{sum}(w_j^2) \rightarrow$ Euclidean norm (L2 norm)

$J(w,b) = \text{Original cost} + (\lambda / 2m) * \|w\|^2$

Usually do not recognize bias (b) → its just 1 value

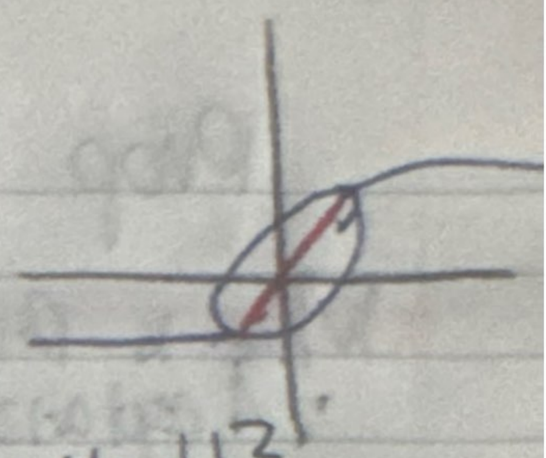
Key benefits

- Shrinks weights towards 0 → reduces overfitting
- Encourage simpler models
- Still uses all features (not sparse)

Scanned with CamScanner

Regularization and Overfitting

Why Regularization Helps with overfitting



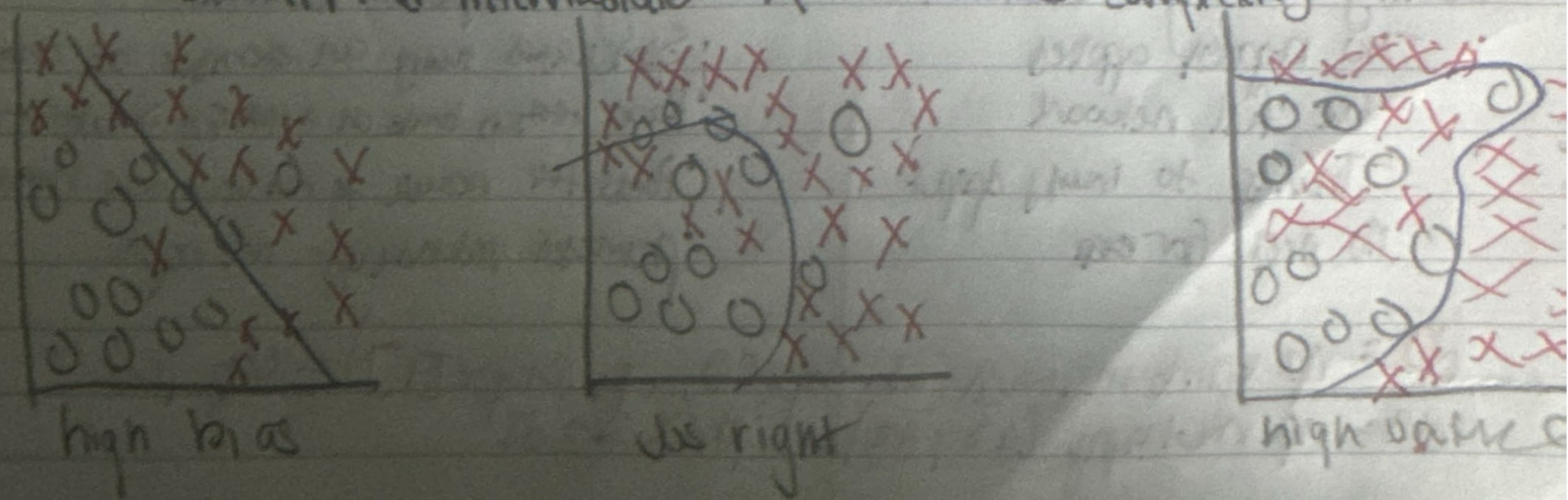
- L^2 Regularization (Weight Decay):
  - cost functions becomes:
    - $J(w,b) = \text{Original cost} + (\lambda/2m) * ||W||^2$
  - Penalizes large weights ($||W||^2$ = forbids nan)
  - Encourages smaller parameter values → reduced model complexity
- Intuition
  - Larger λ → stronger penalty → weights shrink towards zero
  - Shrinking weights reduces hidden units influence
  - Effectively simplifies the network → behaves like a smaller ne
  - Too much regularization → underfitting (high bias)
  - Optimal λ → "just right" model with low bias & variance

Impact on Neural Network Behavior

- Smaller W → smaller $Z = wx + b$
- Using tanh (or sigmoid) activation:
  - If Z stays small → tanh operates in linear region
  - Deep neural network becomes almost linear overall.
  - Linear networks can't model complex boundaries → less overfitting
- Result: Model becomes more stable and less sensitive to noise

Bias Variance Trade Off

- No Regularization
- Too much regularization
- Goal: find intermediate λ to balance complexity



Drop Out Regularization

What is Dropout?

- A regularization technique to reduce overfitting in neural networks
- During training, randomly "drop out" (disable) some neurons
- Each training example uses a different "thinned" network
- At test time, the full network is used (no dropout applied)

Dropout Mechanics

During Training:

- for each neuron in a layer, flip biased coin:
  - Keep it with probability = keep_prob
  - Drop with probability = 1 - keep_prob

Inverted Dropout

1. Forward Pass for layer l:

```
d[l] = np.random.rand(a[l].shape) < keep_prob
a[l] * d[l]
a[l] /= keep_prob
```

- Ensures expected value of activations remains unchanged
- d[l]: dropout mask; 0s & 1s
- Works even if a[l] is vectorized

2. Back propagation:
   - Reuse the same d[l] mask to zero out gradients
   - This is applied both in forward and backward passes

During Testing (Prediction)

- No dropout applied
- Use full network
- Trained to handle dropout
- No need for mask

Why Dropout Works

- Simulates training an ensemble of smaller networks
- Each iteration trains on a different subset of nodes
- Forces the network to not rely too heavily
- promotes redundancy in reps

```
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
a3 = np.multiply(a3, d3)   # a3 *= d3
```

Understanding Dropout

Why Dropout Works (Expanded Intuition)

- Dropout randomly disables neurons → each iteration trains a different sub network
- feels like training a smaller network, which naturally recognizes
- from a single neurons perspective
  - His inputs may be good → can rely on any feature
  - forces the neuron to distribute weights across inputs
  - Prevents over-reliance on any one input → encourages weight shrinking

Similar effect to L2 regularization

- In fact, Dropout ≈ adaptive L2 regularization
- But L2 penalty differs per weight, based on size & corresponding activations.

Per-Layer Dropout Control

Layer Specific keep_prob values

			keep_prob
Input Layer	3 inputs	Usually kept intact	0.9 or 1.0
Hidden Layer	7x7 W2	High parameter = overfit most powerful layer	0.5
Smaller Layers	3x7	Lower risk of overfitting	0.7 - 1.0

Implementation Reminder

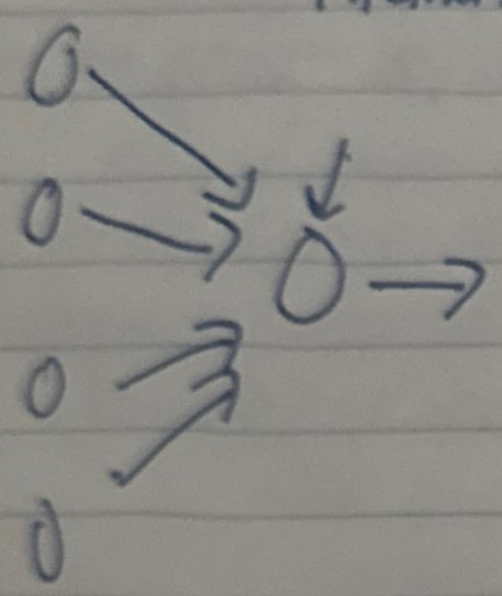
forward Pass (Training Mode)

```
d[l] = np.random.rand(a[l].shape <, keep_prob
d[l] *= d[l]
d[l] /= keep_probs
```

- do not apply dropout when evaluating or predicting

• lecture notes

Intuition: can't rely on any one feature so have to spread out weights → shrink weights



Dropout has the same effect as L2 regularization.

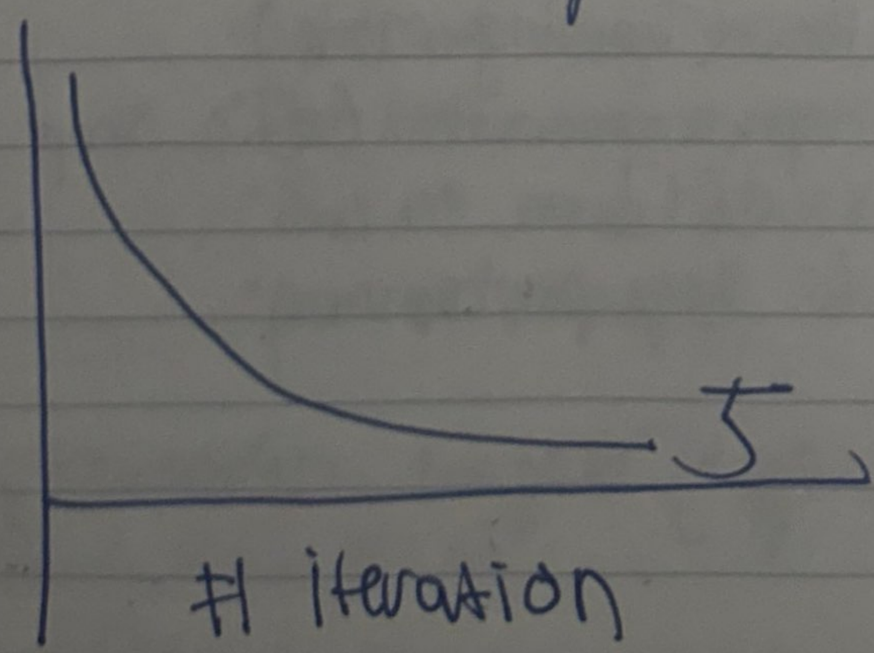
Keep-prob $W^{[2]}$ 7x7
0.5

cases of successful drop out → ☆ Dropout

Drop out is a regularization technique that prevents overfitting.

Downside: cost function J no longer well defined

less well defined.



Other Regularization methods Beyond L2 & Dropout

Data Augmentation

What is it?

- Artificial increase training data by applying transformations
- Helps reduce overfitting when new data is harder expensive to collect.

Common Image Transformations:

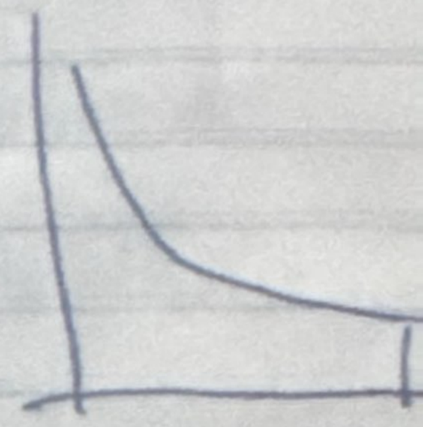
Transformation	Effect
Horizontal flip	cat remains a cat
Random crop/zoom	focus on different parts
Small rotations	
Subtle distortion	Keeps image labels intact

Early Stopping - What is it?

Stops training early based on dev set error to prevent over fitting

Intuition ★

As you run gradient descent →



- Early training, small weights (w) → less overfitting
- Too many iterations, weights grow, more overfit
- Stop when dev error starts increasing

Visualization ★

Iterations → Training time ↓ Dev Error ↓ then ↑

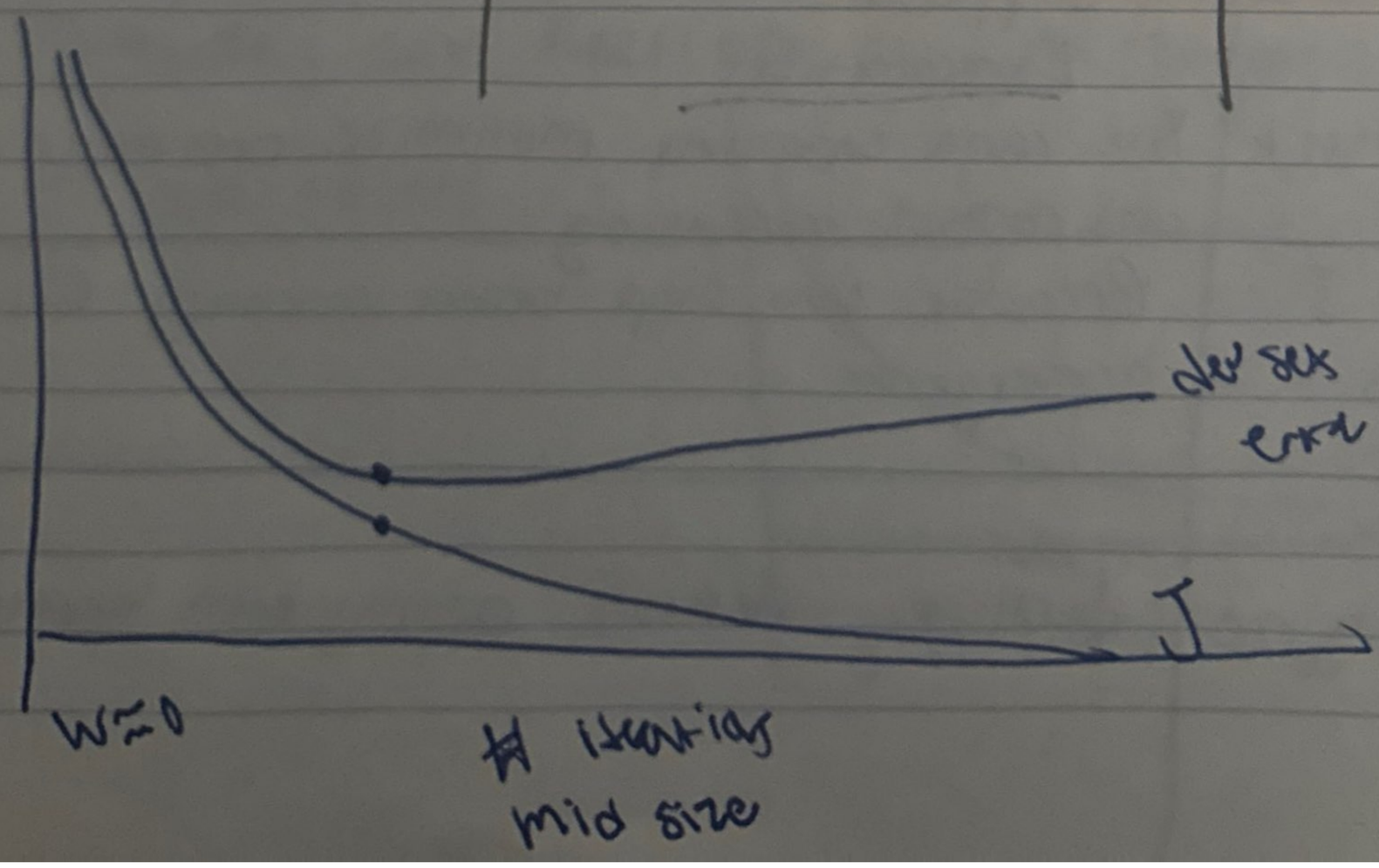
Downsides of Early Stopping

Issue	Explanation
Couples optimization & regularization	You can't separately minimize cost function J and control overfitting
Cost function J not minimized	Because you stop before reaching full convergence
Hard to debug	↓ Hard always decreasing early can't learn

Best Practices

If you use	Notes
Data Augmentation	Greater for vision tasks, low cost, common in CV
Early Stopping	Efficient, reduce compute needed, but loss clean
L2 Regularization	Easier hyper parameter tuning

	Early Stop	L2
Need to tune lambda?	X	✓
Slows training?	X	✓
Keeps optimization quality?	X	✓
Preferred by Researchers?	X	✓



Normalize inputs

What is Input Normalization?

A preprocessing step that ensures each input feature has zero mean and unit variance, making training more efficient and stable.

Two-Step Normalization Process

1. Zero-Mean Centering

   Remove the mean from each feature $\mu = (1/m) * \sum X_i$

   $X := X - \mu$

2. Normalize Variance

   Scale each feature have unit variance

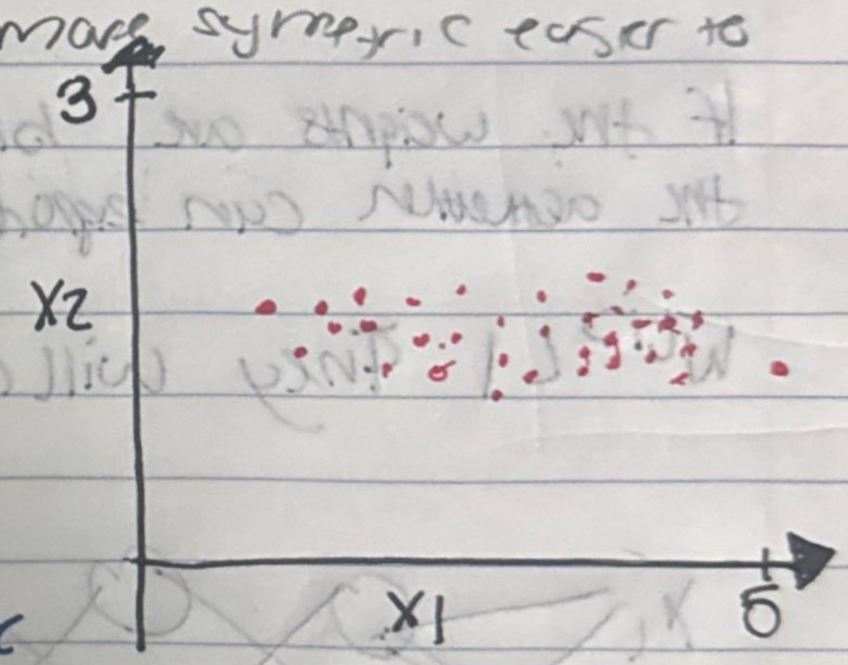
   $\sigma^2 = (1/m) * \sum (X_i)^2 \qquad X := X / \sigma$

Why normalize Input?

When features vary drastically in scale, gradients → slow convergence

Normalization makes $J(w,b)$ more symmetric easier to optimize $X = \begin{bmatrix} x1 \\ x2 \end{bmatrix}$

1.



1. Subtract mean:

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

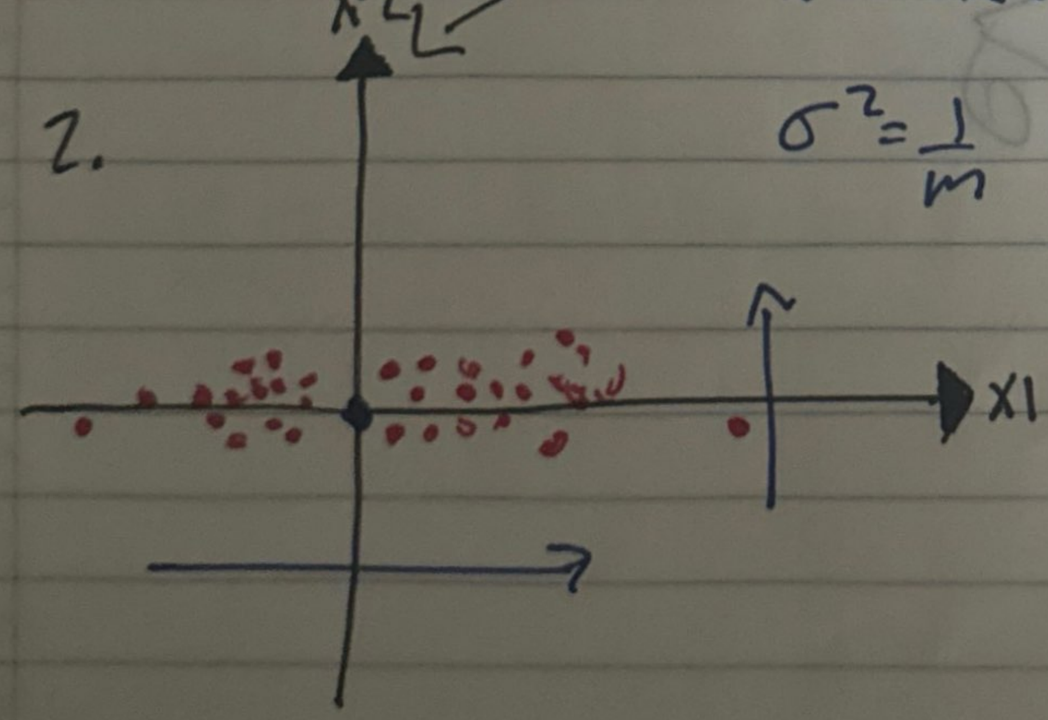
$$\underline{X := X - \mu}$$

use to normalize test set

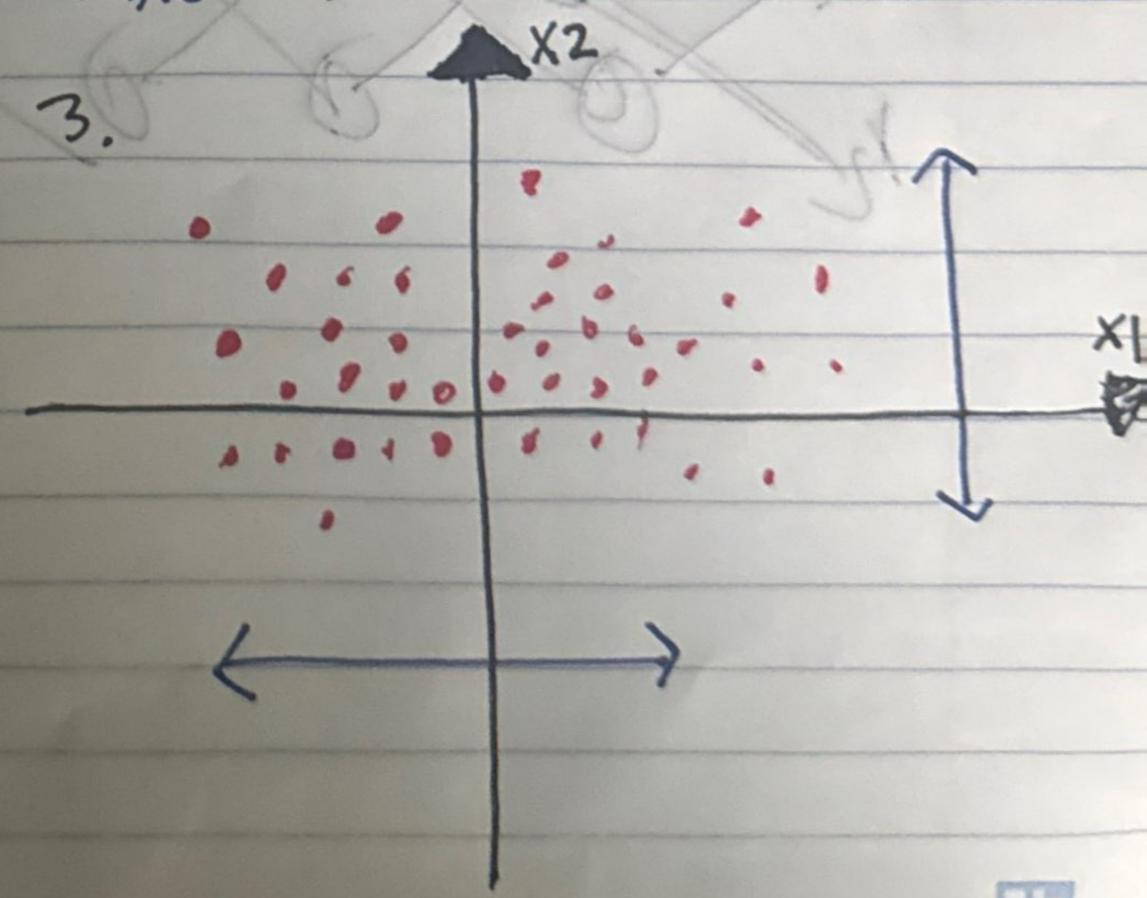
Normalize variance

$$\sigma^2 = \frac{1}{m}$$

2.



3.



if you used unnormalized features looks flat.

Vanishing and / Exploding Gradients

$y(\vec{x}) = 2 \qquad b^{[l]} = 0$

$y = w^{[L]} w^{[L-1]} w^{[L-2]} \cdots w^{[2]} \underbrace{w^{[2]}}$

$z^{[2]} = w^{[2]} x$

$\underbrace{a^{[2]} = g(z^{[2]})} = z^{[2]}$

$a^{[L]} = g(z^{[L]}) = g(w^{[L]} x)$

1.5^{L-1}

0.5^{L-1}

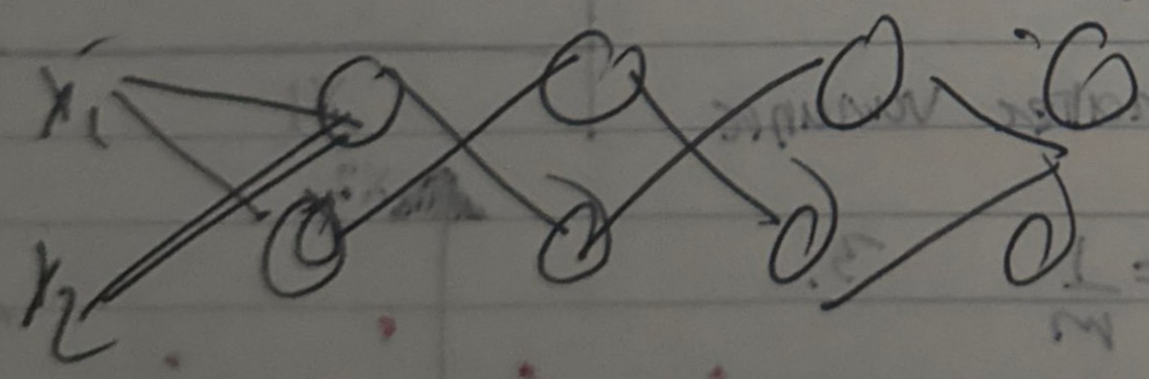
$w^{[1]} = \begin{bmatrix} \cancel{1.5}\ 0.5 & 0 \\ 0 & \cancel{1.5}\ 0.5 \end{bmatrix}$

$\hat{y} = w^{[L]} \begin{bmatrix} \cancel{1.5}\ 0.5 & 0 \\ 0 & \cancel{1.5}\ 0.5 \end{bmatrix}^L$

X_1

If the weights are bigger than 1 with a very deep network the activation can explode

$w^{[l]}$ l They will decrease exponentially



Week 2(d)

Batch vs. mini-batch gradient descent

Vectorization allows you to efficiently compute on m examples

$X = [\underbrace{x^{(1)}\ x^{(2)}\ x^{(3)}}_{X^{\{1\}}} \cdots x^{(1000)} \mid x^{(1001)} \cdots x^{(2000)} \mid \cdots \mid x^{(m)}]$

$Y = [y^{(1)}\ y^{(2)}\ y^{(3)} \cdots y^{(1000)}]$

What if $m = 5{,}000{,}000$?

5,000 mini-batches of 1,000 each

mini-batch t: $X^{\{t\}}, Y^{\{t\}}$

$Z^{[1]}$

$Y^{\{t\}}, Y^{\{t\}}$

for $t = 1 \ldots 5{,}000$:

forward prop on $X^{\{t\}}$

$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$

$A^{[1]} = g^{[1]}(Z^{[1]})$

$\vdots$

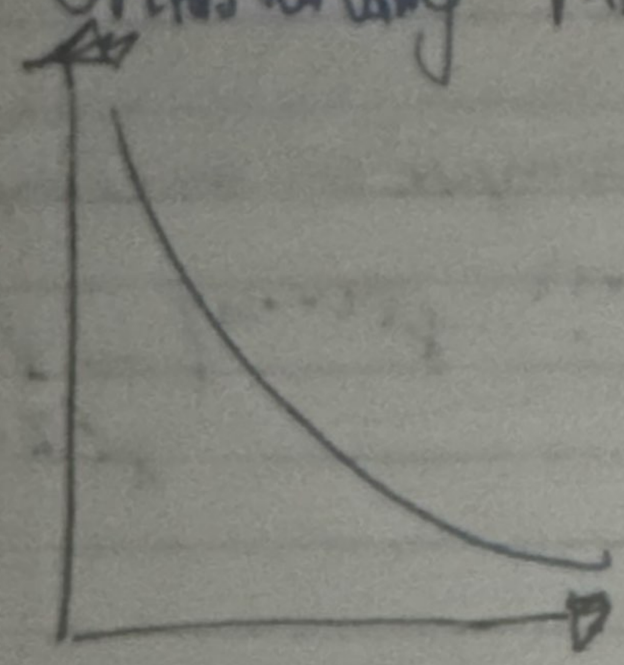
$A^{[L]} = g^{[L]}(Z^{[L]})$

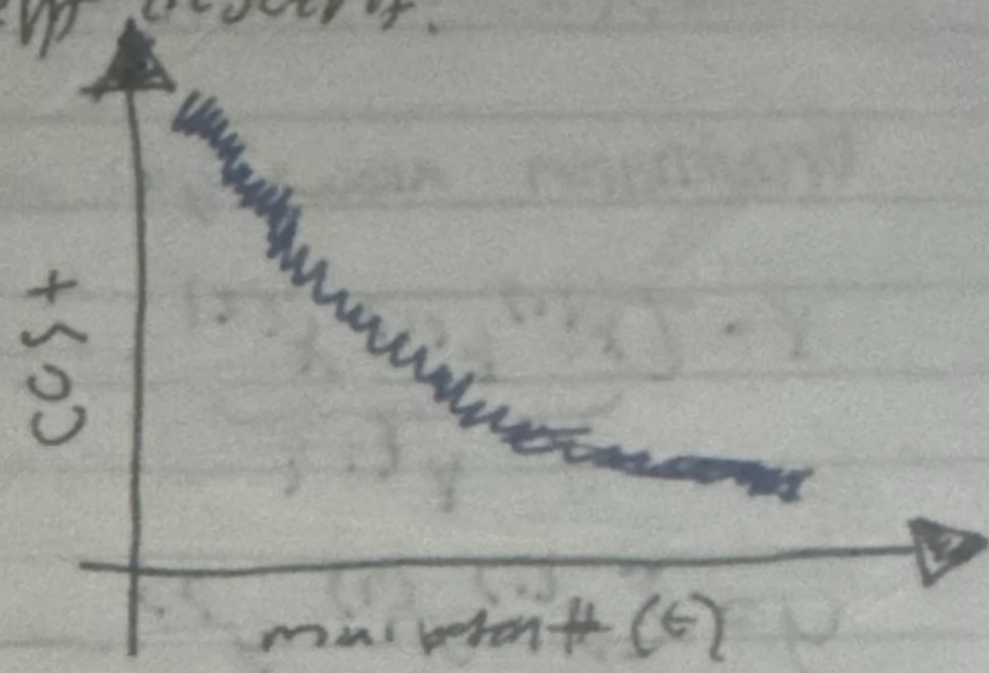
Vectorized implementation (1000 examples)

Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \ldots$

Back prop to compute gradients for $J^{\{t\}}$ (using $\{X^{\{t\}}, Y^{\{t\}}\}$)

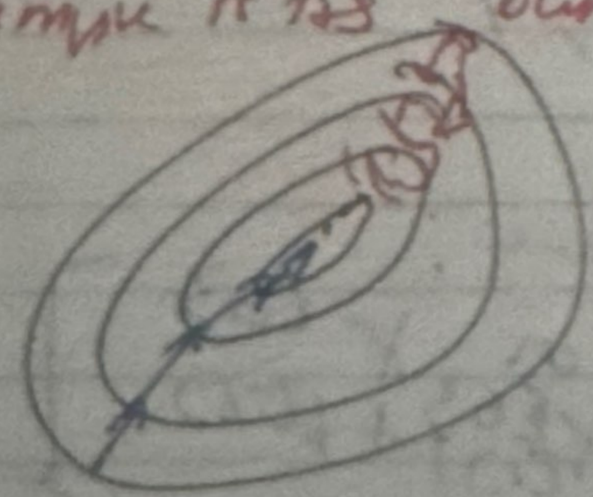
Understanding mini batch gradient descent.

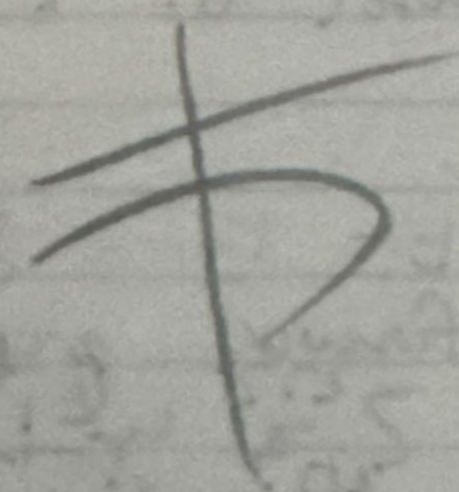




If mini batch size = m: Batch gradient descent

If you mini batch size = 1: Stochastic gradient descent
Every training example is its own





In practice: some where in-between 1 and m

Batch
gradient descent
(mini batch size = m)
↓
Too long
per iteration

In between
(mini batch size
not too big/small)
↓
faster learning
vectorization
Make progress

If small training set: Use batch gradient descent
($m \leq 2000$)