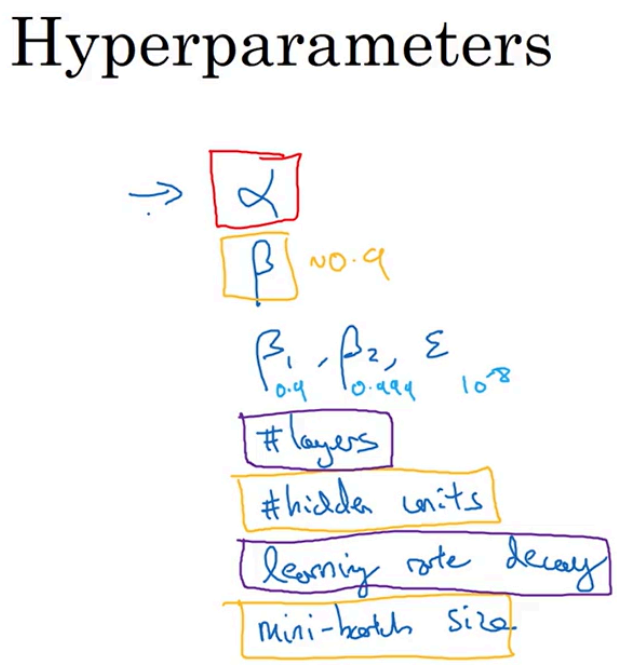
Week 3: Improve Deep Neural Networks - Hyperparameter Tuning, Regularization and Optimization

Tuning Process



This image shows the most common **hyperparameters** in deep learning, ranked by their importance for tuning. Hyperparameters are the settings you choose *before* training a model, as they control the learning process itself.

Tier 1: Highest Priority

- α **(Alpha) - The Learning Rate**: This is highlighted as the **most important hyperparameter**.
    - **What it is:** It controls how big of a step the optimization algorithm (like Gradient Descent) takes during each iteration.

- **Why it's important:** If α is too large, you might overshoot the optimal solution. If it's too small, training will be very slow. Finding the right value is crucial for efficient training.

Tier 2: High Priority

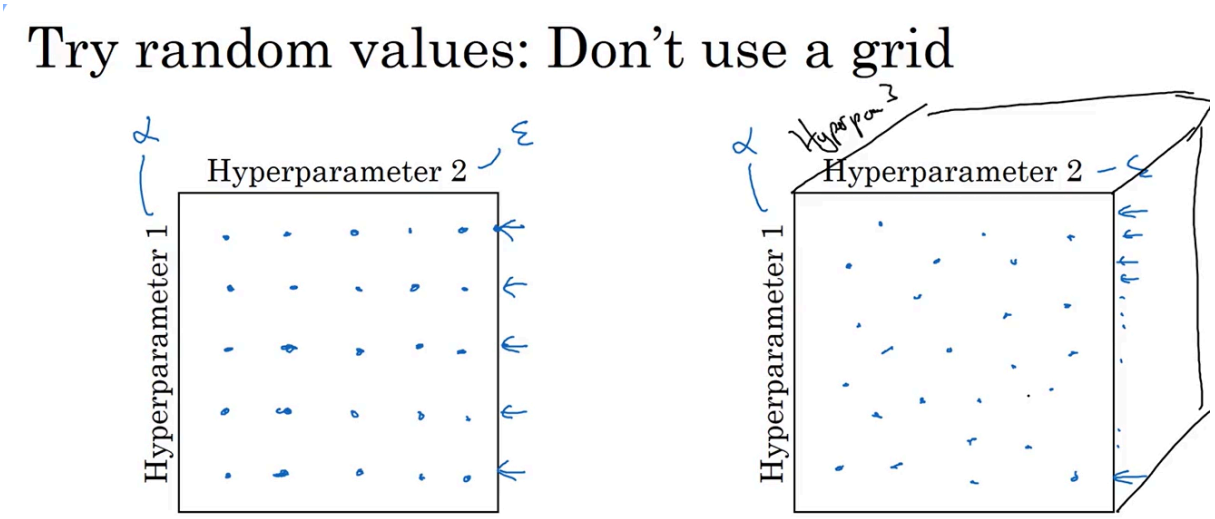
- β **(Beta) - Momentum**: This is typically the momentum term in optimizers like "Gradient Descent with Momentum."
  - **What it is:** It helps to smooth out the learning process. A common value is **0.9**.
  - **Why it's important:** It helps the optimizer accelerate in the correct direction and dampens oscillations, often leading to faster convergence.
- `# hidden units`: The number of neurons in each hidden layer of your network.
  - **What it is:** This determines the "width" of your layers.
  - **Why it's important:** It defines the learning capacity of your model. Too few units might lead to underfitting (the model can't learn the pattern), while too many can lead to overfitting and increased computational cost.
- `mini-batch size`: The number of training examples used in a single iteration.
  - **What it is:** Your training data is divided into small batches, and this number defines their size (e.g., 32, 64, 128).
  - **Why it's important:** It affects the speed and stability of training. Larger batches are computationally efficient, but smaller batches can provide a regularizing effect and help the model generalize better.
- `# layers`: The number of hidden layers in your network.
  - **What it is:** This determines the "depth" of your model.
  - **Why it's important:** Deeper networks can learn more complex, hierarchical features from the data. However, adding too many layers can make the model harder to train (due to issues like vanishing gradients).

Tier 3: Lower Priority (Often Left as Default)

- `learning rate decay`: A strategy to gradually reduce the learning rate α as training progresses.
  - **What it is:** You start with a larger α and make it smaller over time.

- **Why it's important:** It allows the model to take big steps at the beginning of training and smaller, more precise steps as it gets closer to the best solution, helping to fine-tune the weights.

- **Adam Optimizer Parameters ($\beta1$, $\beta2$, ε):** These are specific to the **Adam optimization algorithm**.
  - **What they are:** $\beta1$ is the exponential decay rate for the first moment estimate (like momentum), $\beta2$ is for the second moment estimate, and ε (epsilon) is a tiny number to prevent division by zero.
  - **Why they are lower priority:** The authors of the Adam paper found that the default values of $\beta1=0.9$, $\beta2=0.999$, and $\varepsilon=10^{-8}$ work exceptionally well for most problems. You usually don't need to change them.



This image explains why you should use **random search** instead of **grid search** when tuning hyperparameters. The key idea is that random search is more efficient at finding good hyperparameter values because it explores the search space more effectively.

Grid Search (The Inefficient Method)

The image on the left illustrates **grid search**.

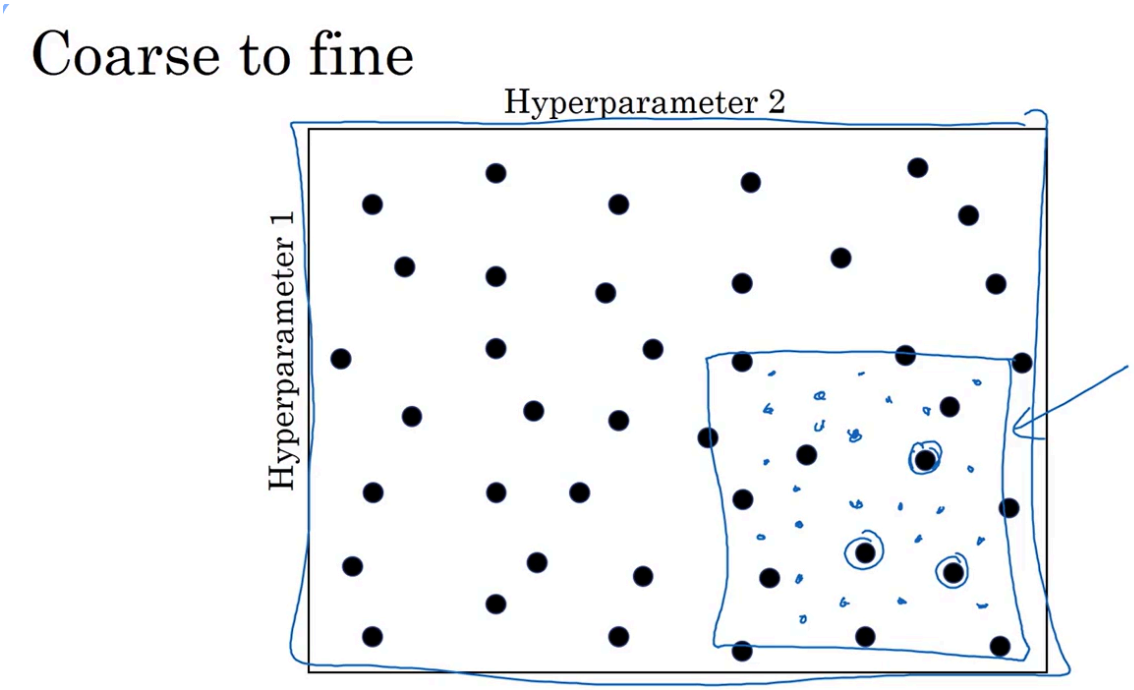
- **How it works:** You pre-define a grid of values for each hyperparameter. For example, you might test learning rates $\alpha = [0.1, 0.01, 0.001]$ and momentum values $\beta = [0.8, 0.9]$. Grid search then systematically tries every single combination of these values.
- **The problem:** This approach is inefficient because you don't know which hyperparameters are the most important beforehand. In the example, **Hyperparameter 1**

(α) might be very important, while **Hyperparameter 2 (ε)** might have almost no effect. Grid search wastes a lot of time testing different values for the unimportant hyperparameter (ε) at each fixed value of the important one (α). As you can see on the left, you're only testing a few distinct values for α .

Random Search (The Better Method)

The image on the right illustrates **random search**.

- **How it works:** Instead of a fixed grid, you define a range for each hyperparameter (e.g., a learning rate between 0.0001 and 0.1) and then randomly sample a certain number of combinations from within that range.
- **The advantage:** With the same computational budget (the same number of dots), you test a much wider and more diverse set of values for each hyperparameter. You aren't stuck with a few pre-selected points. This significantly increases your chances of discovering a really good value for the most important hyperparameters, leading to a better-performing model.



This image illustrates the **"Coarse to Fine"** strategy for hyperparameter tuning, which is a methodical way to zoom in on the best settings for your model. It's typically a two-step process.

Step 1: Coarse Search (The Wide Search)

This is represented by the large box covering the entire search space.

- **What you do:** You begin by performing a **random search** over a very broad range of possible values for your hyperparameters.
- **The goal:** The aim here isn't to find the single best hyperparameter combination right away. Instead, you want to identify a **promising region** where the models perform generally well. You're basically getting a rough idea of where the "sweet spot" is.

Step 2: Fine Search (The Focused Search)

This is represented by the smaller, hand-drawn box that has been "zoomed in" on a promising area.

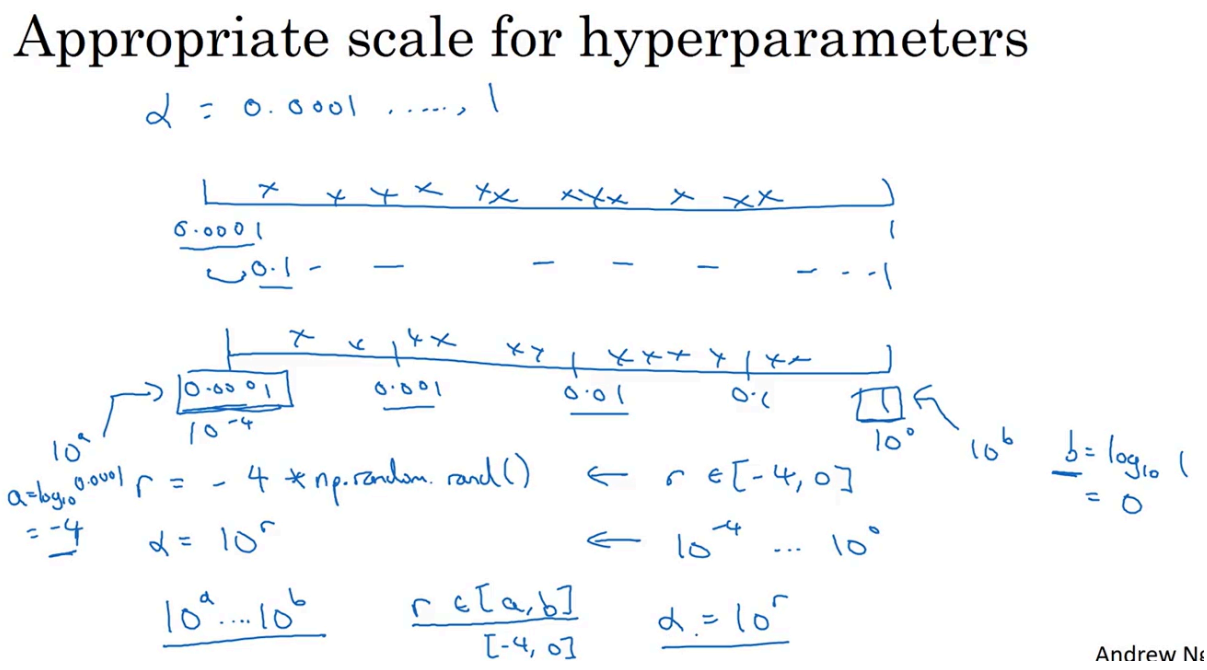
- **What you do:** After the coarse search, you analyze the results and identify the area that produced the best-performing models. You then conduct a second, more focused random search, sampling more densely **only within this smaller, promising region**.
- **The goal:** Now you dedicate your computational power to exploring this smaller space in greater detail to pinpoint the optimal values. The circled points in the image represent the best-performing models found during this focused phase.

Why Use This Approach?

The "Coarse to Fine" strategy is both **efficient and effective**:

- **Efficiency:** It saves computational resources by not wasting time on unpromising regions of the hyperparameter space. 🗺️
- **Effectiveness:** It increases your chances of finding a great set of hyperparameters by combining broad exploration (coarse search) with focused exploitation (fine search).

Using an Appropriate Scale to pick Hyperparameters.



1. Concept

When tuning hyperparameters (e.g., **learning rate α**), it is important to choose the right **scale** for searching values.

Instead of searching linearly (e.g., `0.0001, 0.0002, 0.0003...`), we often search **logarithmically** (e.g., `10^-4, 10^-3, 10^-2, ..., 1`).

2. Why Logarithmic Scale?

- Many hyperparameters (like α) vary **over several orders of magnitude**.
- A **linear search** wastes trials because most useful values are clustered in small regions.
- A **logarithmic search** gives equal importance across ranges.

Example:

- $\alpha \in [0.0001, 1]$
- Better to search values like `10^-4, 10^-3, 10^-2, 10^-1, 1` than evenly spacing them.

3. Sampling α (Learning Rate) Randomly

Instead of trying fixed values, we **sample randomly** on a log scale:

- Pick

  $r \sim U[-4, 0]$

- Then set

  $\alpha = 10^r$

👉 This gives α values uniformly distributed in log-space between 10^{-4} and 10^0 = 1.

---

4. General Formula

For any hyperparameter between 10^a and 10^b:

1. Sample

   $r \sim U[a, b]$

2. Set

   $\alpha = 10^r$

This ensures better coverage of the search space.

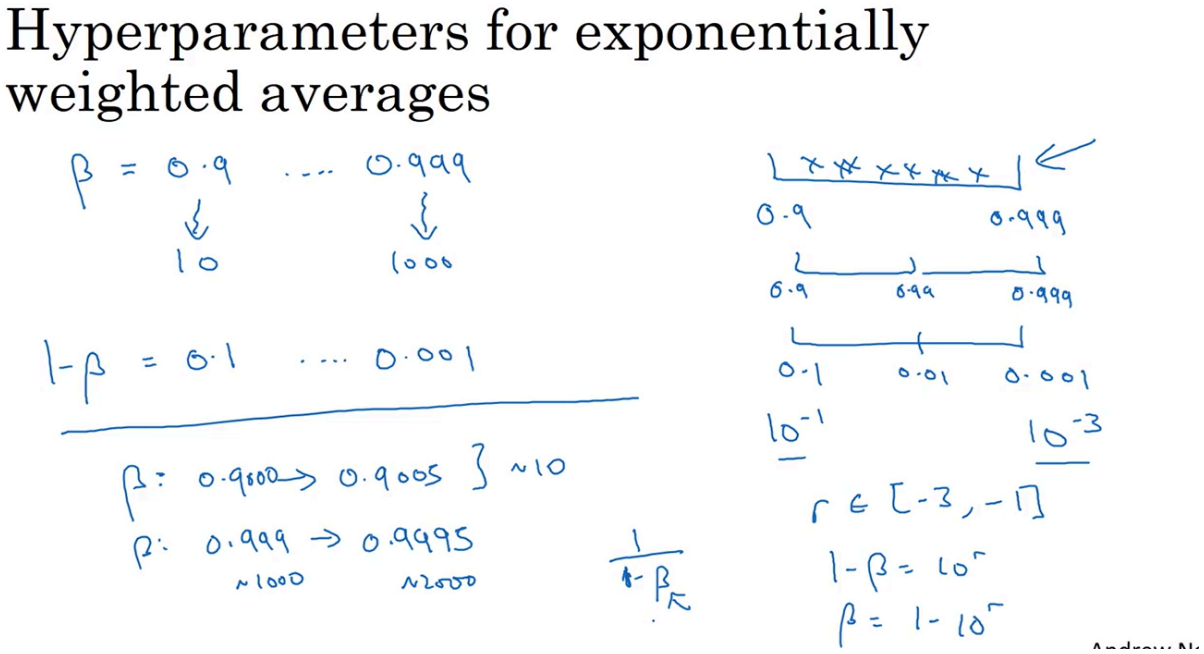
---

5. Intuition

- Think of hyperparameter tuning as "zooming in" on useful ranges.
- Logarithmic scaling avoids wasting effort where hyperparameters are unlikely to work.
- Example ranges:
  - **Learning rate (α):** $10^{-4} \rightarrow 1$
  - **Regularization (λ):** $10^{-5} \rightarrow 10^2$

✅ **Key takeaway:**

Always search **logarithmically** for hyperparameters that span many magnitudes, especially learning rate and regularization terms.



Got it 👍 Let's write clean notes for this second image in the same structured way.

---

1. Concept

In algorithms like **momentum** or **RMSProp**, we use **exponentially weighted averages** with parameter β.

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

- β controls how much weight is given to the **past vs. present**.
- Common values: $\beta \in [0.9, 0.999]$.

---

2. Equivalent Time Constant

- Effective window size ≈ $\frac{1}{1-\beta}$
- Examples:
  - β=0.9 → window size ≈ 10 steps
  - β=0.999 → window size ≈ 1000 steps

This tells us how much past data is remembered.

---

3. Better Search Strategy

Instead of searching β directly (0.9, 0.91, 0.92 …), tune **1−β** on a **logarithmic scale**:

- $1-\beta \in [0.1, 0.001]$
- Expressed as:

  $$1 - \beta = 10^r, \quad r \sim U[-3, -1]$$
- Then compute:

  $$\beta = 1 - 10^r$$

---

4. Intuition

- When β is very close to 1, small changes (e.g., 0.999 → 0.9995) can drastically affect performance.
- That's why we sample **logarithmically** in 1−β, not linearly in β.

---

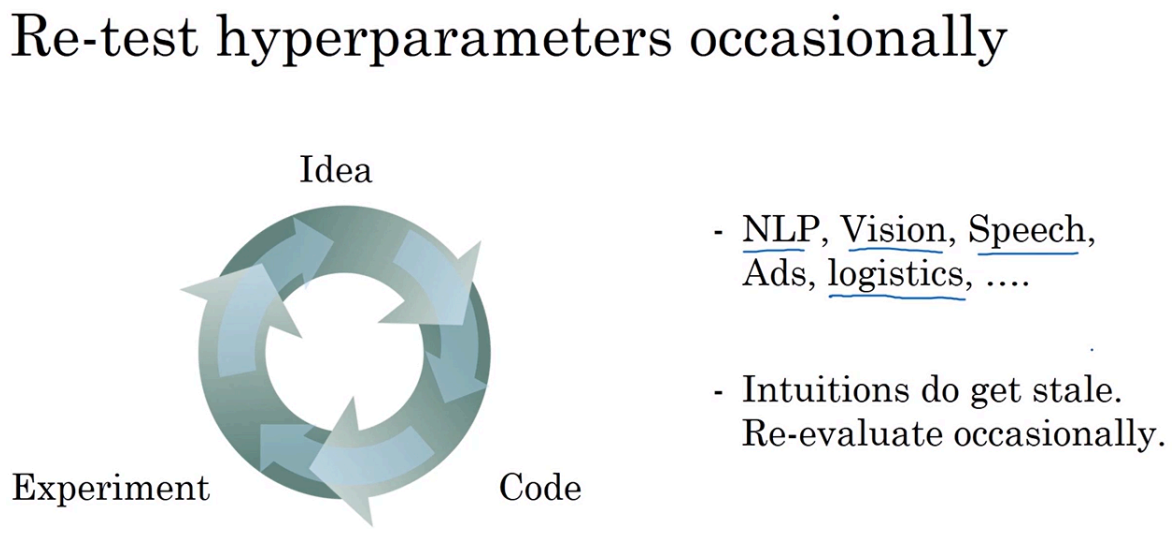
5. Typical Ranges

- **Momentum β**: 0.9 → remembers ~10 steps
- **Adam β1**: 0.9
- **Adam β2**: 0.999

✅ **Key takeaway:**

Tune **1−β** on a logarithmic scale (not β directly), because useful values are clustered near β ≈ 1.

Hyperparameters Tuning in Practice: Pandas vs. Caviar



1. Concept

- Hyperparameters that worked well once may **not remain optimal** over time.
- As tasks, data, and domains evolve, we must **re-evaluate** hyperparameter choices.

---

2. Domains Affected

- **NLP**
- **Computer Vision**
- **Speech Recognition**
- **Ads / Recommendation Systems**
- **Logistics / Operations**
- (and many other ML applications)

---

3. Why Re-test?

- **Intuitions get stale** → What worked in one setup may not work after data distribution shifts or model architecture changes.
- **Regular re-testing** ensures the model adapts to:
  - New datasets
  - Changing input distributions
  - Advances in algorithms

---

4. Process (Iteration Cycle)

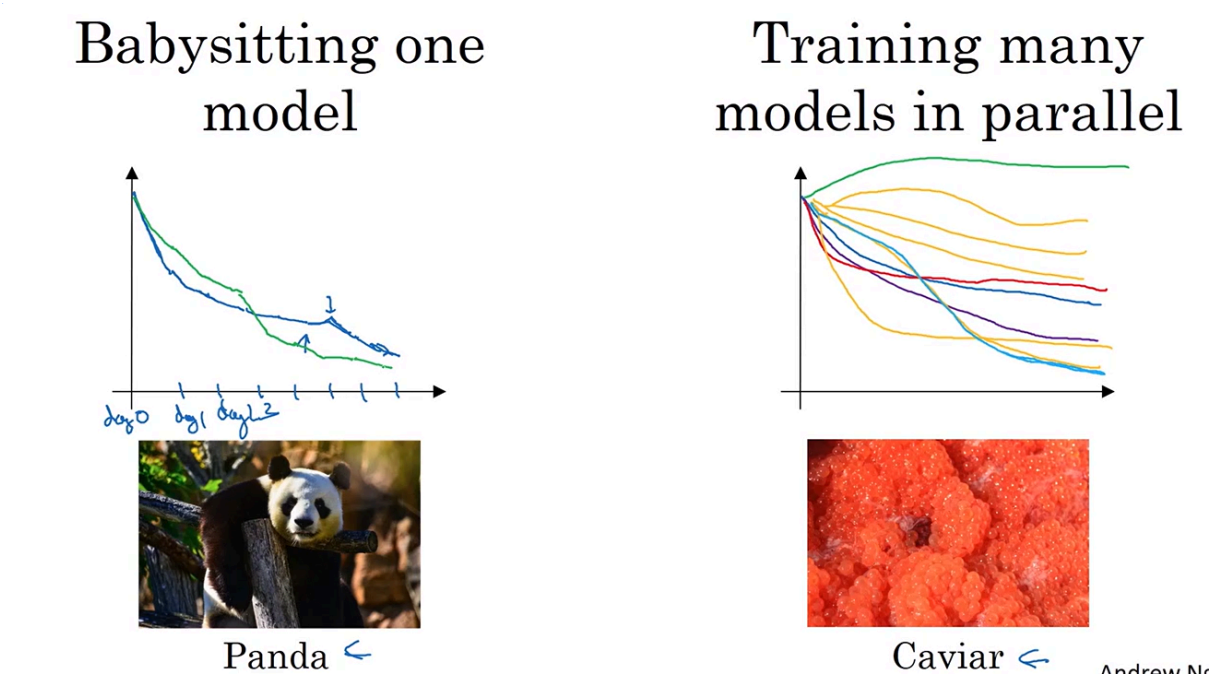
The workflow follows an **Idea → Code → Experiment → Idea** cycle:

- **Idea**: Generate a hypothesis (e.g., “maybe smaller learning rate will work”).
- **Code**: Implement change.
- **Experiment**: Run and observe results.
- Loop back to refine.

✅ **Key takeaway:**

Even if you’ve found good hyperparameters once, **don’t stick with them forever**.

Re-test occasionally to stay aligned with evolving data and problem domains.



1. Babysitting One Model (🐼 Panda)

- Train **one model** at a time, monitoring its learning curve closely.
- If it doesn't perform well → tweak hyperparameters → retrain again.
- **Drawback:** Very slow, since you're waiting days/weeks for feedback from just one experiment.
- Analogy: Like "babysitting a panda" — requires constant attention, progress is slow.

2. Training Many Models in Parallel (🍣 Caviar)

- Instead of one model, train **many models simultaneously** with different hyperparameters.
- Compare learning curves, quickly see which hyperparameters work.
- Much **faster experimentation**, leading to better results in less time.
- Analogy: Like "caviar" — many small eggs together, easier to sample broadly.

3. Intuition

- **Parallel training** scales better with modern compute (GPUs/TPUs, cloud).

- Encourages **exploration** of hyperparameter space rather than “betting on one”.
- Critical for finding good learning rates, regularization terms, optimizers, etc.

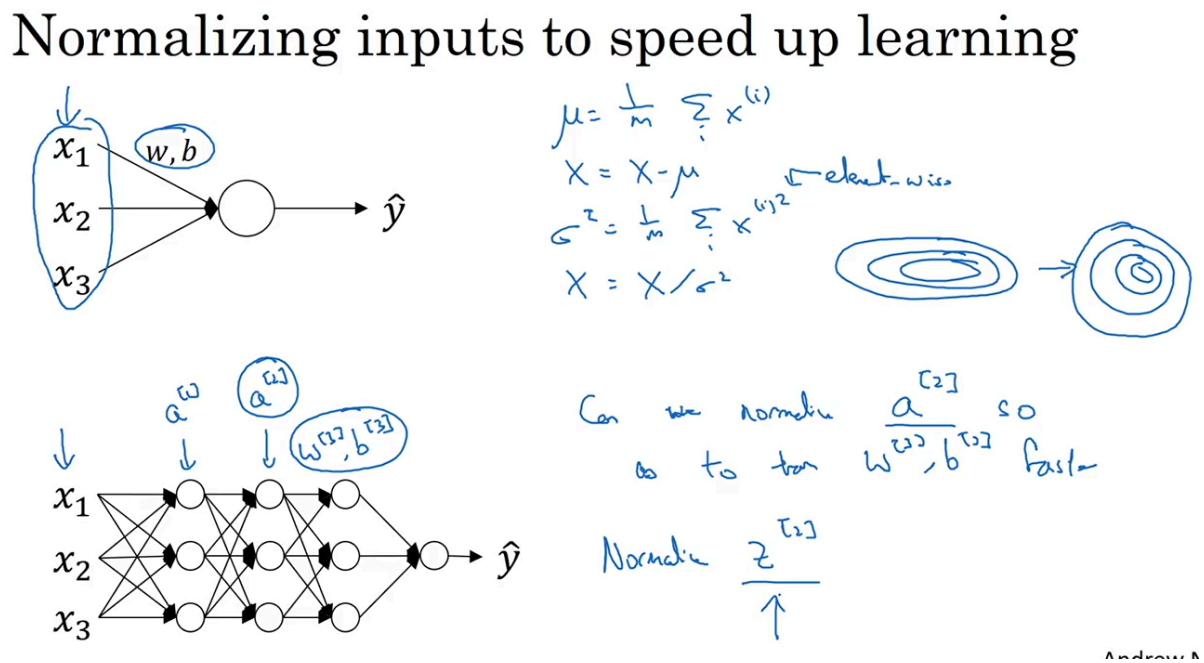
---

4. Key Takeaway

👉 Don’t “babysit one panda” (one model).

👉 Instead, train **multiple models in parallel** (like caviar) to speed up discovery of good hyperparameters.

Normalizing Activations in a Network



1. Motivation

- Neural networks train faster when inputs are normalized.
- Without normalization, cost contours can be **elongated ellipses** → gradient descent zig-zags, slows convergence.
- With normalization, contours become **more circular** → smoother and faster optimization.

---

2. Normalization Steps

For each feature x:

1. **Mean normalization**

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

   $x \leftarrow x - \mu$

2. **Variance scaling**

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

   $x \leftarrow \frac{x}{\sigma}$

Where:

- μ = mean of feature
- σ^2 = variance
- σ = standard deviation of feature

3. Effect on Neural Networks

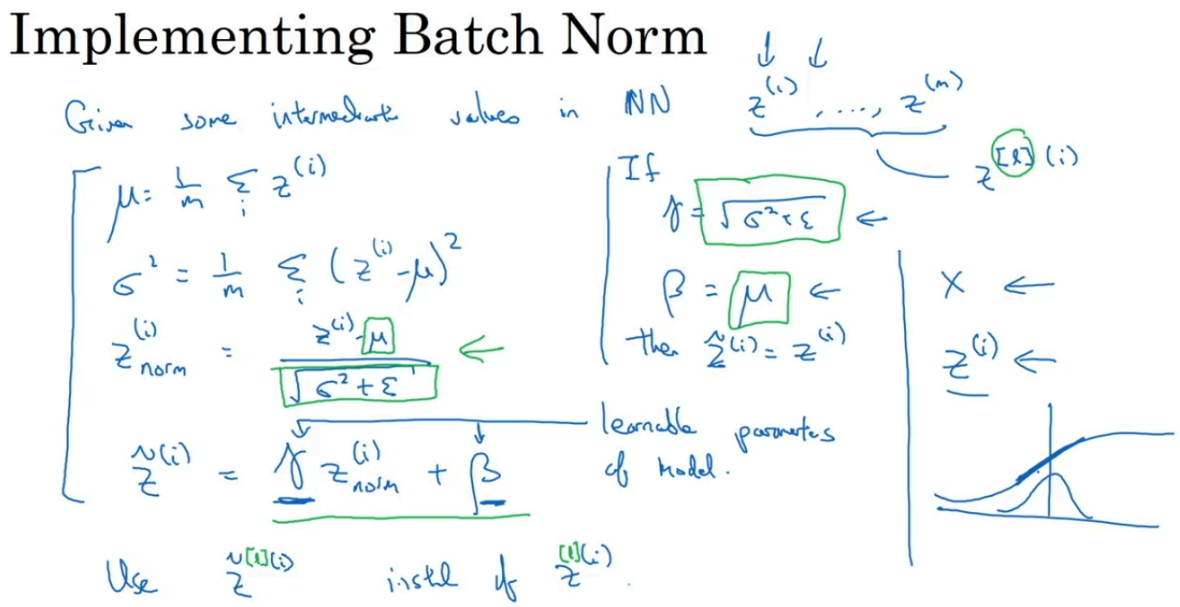
- Applies to **input layer features** $x_1, x_2, \ldots$.
- Leads to faster optimization of parameters w, b.
- Idea extends to hidden activations too (this motivates **Batch Normalization**).

---

4. Key Insight

- Normalization makes training **faster** and more **stable**.
- Instead of struggling with poor conditioning (long narrow valleys), gradient descent makes steady progress.

✅ **Key takeaway:** Always normalize inputs before training — it can significantly improve convergence speed.



📝 Notes: Implementing Batch Normalization

1. Goal

- Normalize intermediate values in a neural network to stabilize and accelerate training.
- Done at each layer (not just input).

---

2. Batch Norm Steps

For activations $z^{(i)}$ in a mini-batch:

1. **Compute mean**

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

1. **Compute variance**

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (z^{(i)} - \mu)^2$$

1. **Normalize**

$$z^{(i)}_{\text{norm}} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

(ϵ is a small constant to avoid division by zero).

1. **Scale and shift (learnable parameters)**

$$\tilde{z}^{(i)} = \gamma \cdot z^{(i)}_{\text{norm}} + \beta$$

Where:

- γ (scale) and β (shift) are **learnable parameters**, allowing the model to undo normalization if needed.

---

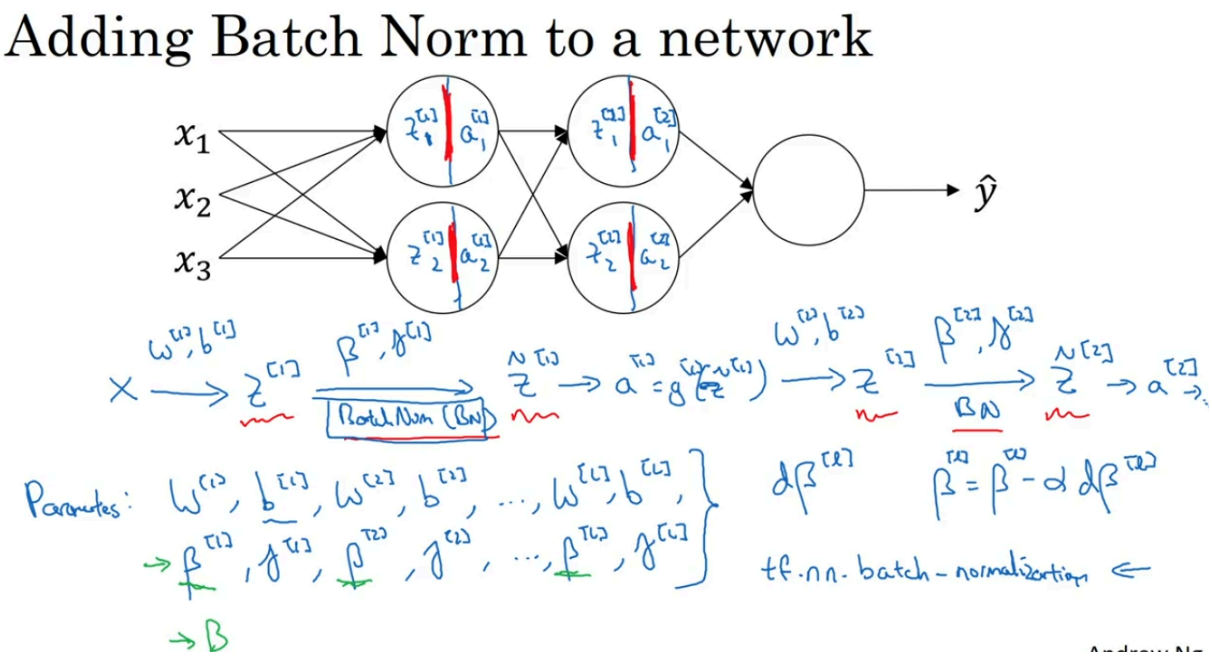
3. Usage

- Instead of using$z^{(i)}$ directly, use $\tilde{z}^{(i)}$ in forward propagation.
- Helps reduce **internal covariate shift** (distribution of activations changing during training).
- Results in **faster convergence** and sometimes allows **higher learning rates**.

👉 Intuition:

- Normalization keeps the activations well-behaved (mean ~0, variance ~1).
- But if the network needs a different distribution, γ and β let it adjust.

Fitting Batch Norm into a Neural Network



🔹 Motivation

- Speeds up training by normalizing intermediate values in the network.
- Reduces **internal covariate shift** (changes in distribution of activations during training).
- Allows **higher learning rates** and improves stability.

🔹 Adding Batch Norm to a Network

For each layer:

$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$

Instead of directly using $z^{[l]}$, we normalize it:

1. **Batch Normalization step**:

$$\tilde{z}^{[l]} = \frac{z^{[l]} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

2. **Scale & Shift** (learnable parameters):

$$\hat{z}^{[l]} = \gamma^{[l]} \tilde{z}^{[l]} + \beta^{[l]}$$

3. Apply activation:

$$a^{[l]} = g(\hat{z}^{[l]})$$

🔹 Parameters in the Network

Now the network has **two sets of parameters** per layer:

- Usual weights and biases: $W^{[l]}, b^{[l]}$
- Batch norm params: $\gamma^{[l]}, \beta^{[l]}$

So total parameters:

$\{W^{[L]}, b^{[L]}, \gamma^{[1]}, \beta^{[1]}, ..., \gamma^{[L]}, \beta^{[L]}\}$

🔹 Backpropagation

- Gradients also update $\gamma^{[l]} \quad and \quad \beta^{[l]}$.
- Example update:
  $\beta^{[l]} := \beta^{[l]} - \alpha \, d\beta^{[l]}$

🔹 Implementation

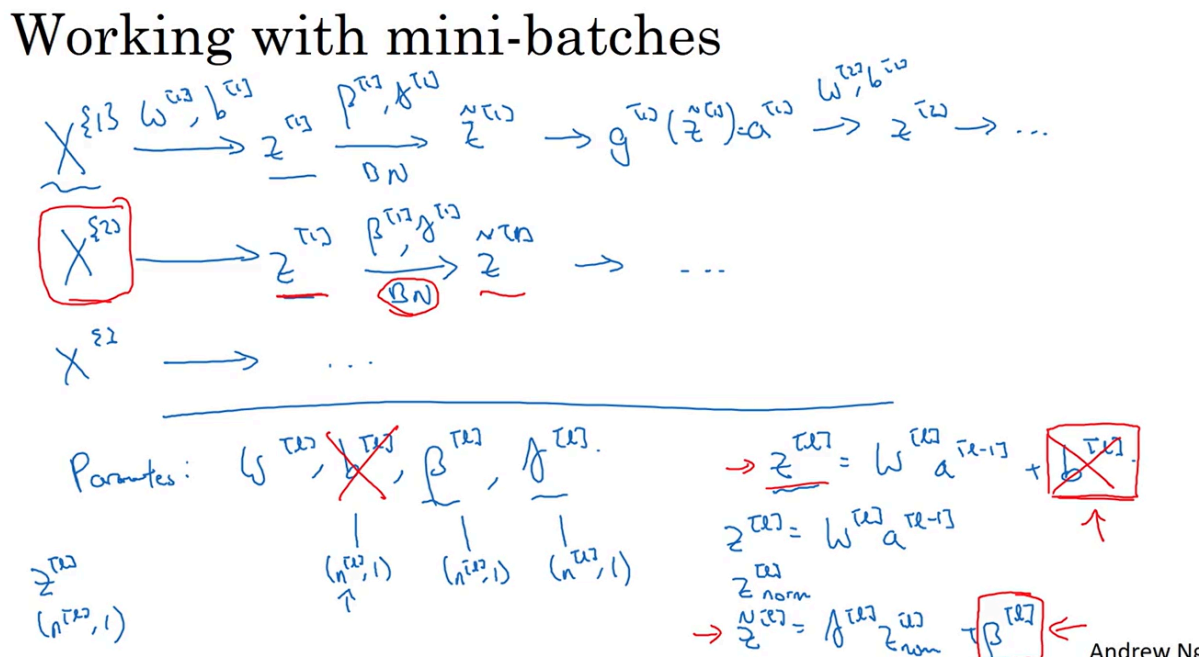
- TensorFlow:

```
tf.nn.batch_normalization()
```

✅ **Key Takeaway**:

Batch Norm normalizes activations at each layer, introduces new learnable parameters (γ, β), improves training stability, and allows faster convergence.

Working with mini-batches



🧠 Batch Normalization with Mini-Batches

🔹 Idea

- Instead of normalizing across the **entire dataset**, Batch Norm is applied **per mini-batch** during training.
- Each mini-batch computes its own **mean** and **variance** for normalization.

🔹 Forward Propagation with Mini-Batches

For mini-batch $X^{\{i\}}$:

1. Compute linear activation:

$$z^{[l](i)} = W^{[l]} a^{[l-1](i)}$$

2. Apply Batch Norm:
    - Compute batch mean:

$$\mu = \frac{1}{m_{\text{mini}}} \sum_{i=1}^{m_{\text{mini}}} z^{[l](i)}$$

    - Compute batch variance:

$$\sigma^2 = \frac{1}{m_{\text{mini}}} \sum_{i=1}^{m_{\text{mini}}} (z^{[l](i)} - \mu)^2$$

    - Normalize:

$$z_{\text{norm}}^{[l](i)} = \frac{z^{[l](i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

3. Scale & shift:

$$\hat{z}^{[l](i)} = \gamma^{[l]} z_{\text{norm}}^{[l](i)} + \beta^{[l]}$$

4. Apply activation:

$$a^{[l](i)} = g(\hat{z}^{[l](i)})$$

🔹 Parameters

- **Weights**: $W^{[l]}$
- **No bias**: $b^{[l]}$ is unnecessary (crossed out in the diagram) since Batch Norm introduces its own shift ($\beta^{[l]}$).
- **Batch Norm params**: $\gamma^{[l]}, \beta^{[l]}$

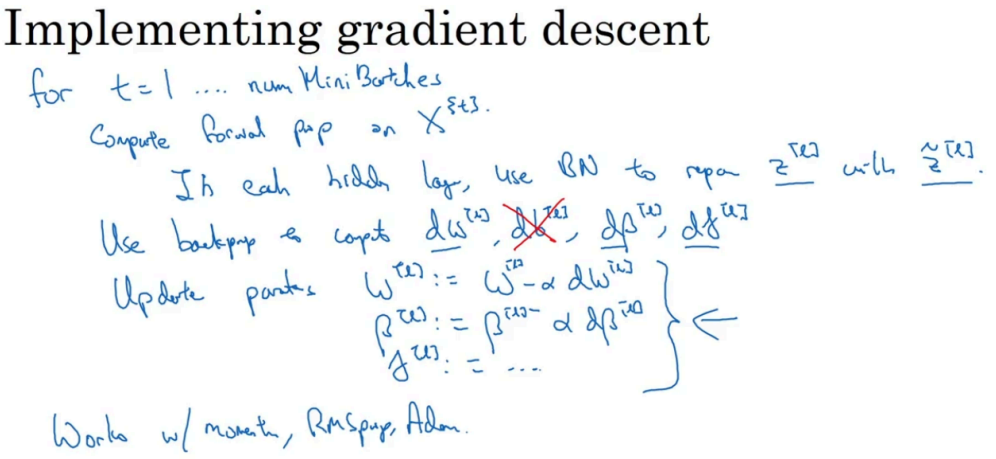
🔹 Dimensions

- $z^{[l]} : (n^{[l]}, 1)$
- $\gamma^{[l]}, \beta^{[l]} : (n^{[l]}, 1)$

✅ **Key Takeaway**:

- With Batch Norm, **bias terms are redundant** (we use $\beta^{[l]}$ instead).
- Each mini-batch maintains its own mean and variance, which makes training more stable and efficient.

Implementing Gradient Descent



🧠 Implementing Gradient Descent with Batch Normalization

🔹 Training Loop

For each mini-batch $t = 1 \dots \text{numMiniBatches}$:

1. **Forward Propagation** on mini-batch $X^{\{t\}}$
   - At each hidden layer, replace:

     $$z^{[l]} \quad \rightarrow \quad \hat{z}^{[l]} \quad \text{(using Batch Norm)}$$
2. **Backpropagation**
   - Compute gradients:

     $$dW^{[l]}, \quad d\gamma^{[l]}, \quad d\beta^{[l]}$$
   - Note: $db^{[l]}$ is unnecessary (bias is redundant in BN).
3. **Parameter Updates**
   - For weights:

     $$W^{[l]} := W^{[l]} - \alpha\, dW^{[l]}$$
   - For Batch Norm parameters:

$$\beta^{[l]} := \beta^{[l]} - \alpha\, d\beta^{[l]}$$
$$\gamma^{[l]} := \gamma^{[l]} - \alpha\, d\gamma^{[l]}$$

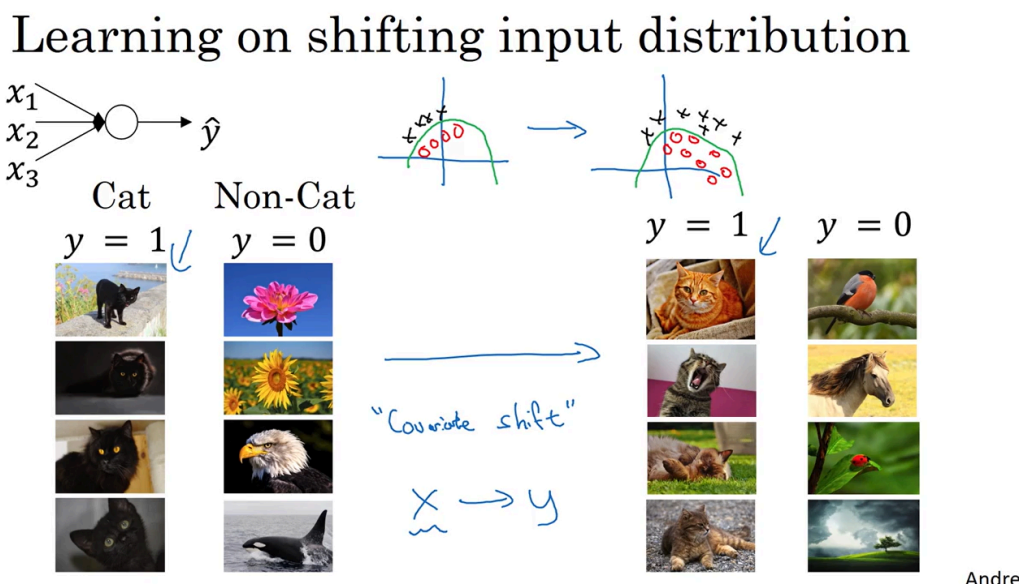
🔹 Optimizers

- Works seamlessly with:
  - **Momentum**
  - **RMSprop**
  - **Adam**

✅ **Key Takeaway**:

Batch Norm modifies the forward pass (normalizing z), but gradient descent and optimizers still work as usual — now with extra parameters γ, β.

Why does Batch Norm Work?



🧠 Learning on Shifting Input Distributions

🔹 Problem Setup

- Binary classification: Cat vs Non-Cat
  - y = 1: Cat
  - y = 0: Non-Cat

- Training set and test set may have **different input distributions**.

🔹 Covariate Shift

- Training data may come from one distribution, e.g.:
  - Cats: black cats
  - Non-cats: flowers, whales, eagles
- Test data may come from a different distribution, e.g.:
  - Cats: orange/brown cats
  - Non-cats: birds, horses, landscapes

📌 The input distribution $P(x)$ **changes** between training and testing,

but the conditional distribution $P(y|x)$ remains the same.

🔹 Impact

- The model trained on the first dataset may not generalize well to the new dataset.
- Example: Classifier trained on black cats struggles with orange cats.

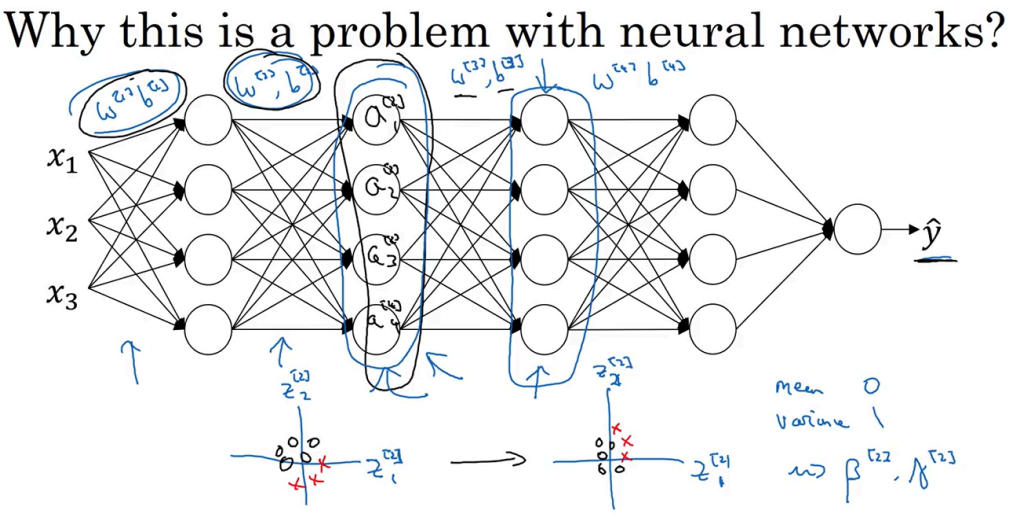
🔹 Key Idea

- **Covariate shift** = when the distribution of inputs x shifts,

  but the task (mapping $x \rightarrow y$) is still the same.

✅ **Key Takeaway**:

Batch Normalization helps mitigate covariate shift inside the network by keeping activations in a stable distribution — making training more robust to input distribution changes.



This image is about **why internal covariate shift is a problem in neural networks**.

1. Neural Network Forward Pass

- Input layer: $x_1, x_2, x_3 \rightarrow$ first hidden layer.
- Each layer computes:

  $z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$

  $a^{[l]} = g(z^{[l]})$

So outputs of one layer become inputs to the next.

---

2. The Problem (Internal Covariate Shift)

- The **distribution** of $z^{[l]}$ (inputs to a hidden layer) keeps changing during training because weights $W^{[l]}$ and biases $b^{[l]}$ get updated.
- For example, below the network you see plots of $z_1^{[2]}$ and $z_2^{[2]}$.
  - Initially, centered around zero (nice distribution).
  - After training updates, the distribution shifts (red crosses).

This forces the next layer to constantly adapt to new distributions → slows down training.

---

3. Why is this bad?

- If inputs to each layer are not **well-scaled** (mean ≈ 0, variance ≈ 1), activations can:
  - Saturate (for sigmoid/tanh → gradients vanish).
  - Explode (large values → unstable training).

This means:

- Gradient descent takes **longer to converge**.
- Sometimes it **fails to converge**.

---

4. Solution: Normalize Activations (Batch Normalization)

- Before applying activation, normalize:

$$\hat{z}^{[l]} = \frac{z^{[l]} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

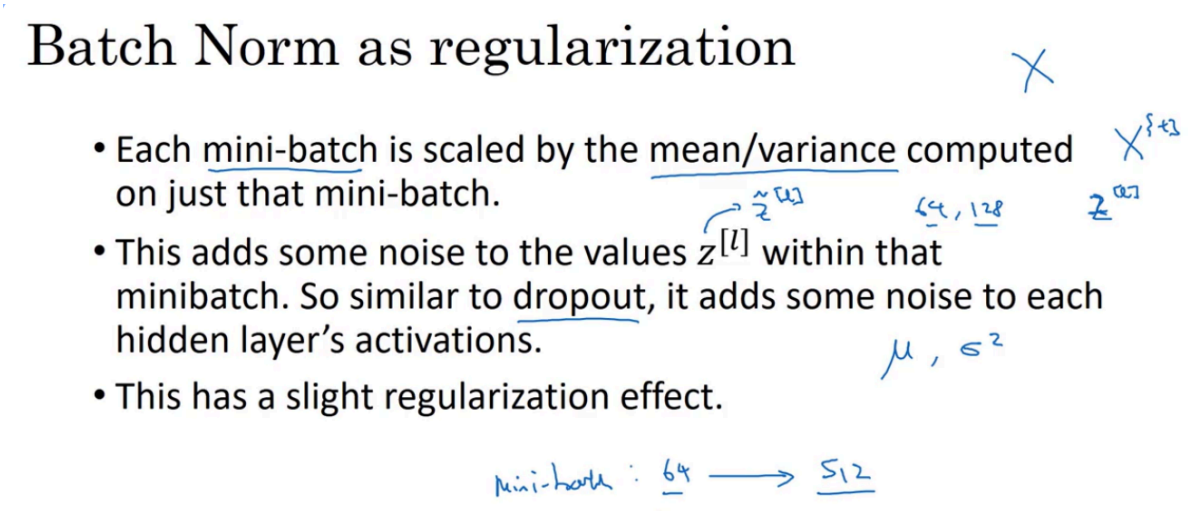
  where mean $\mu = 0$, variance $\sigma^2 = 1$.

- Then scale & shift with learnable parameters:

$$z^{[l]}_{norm} = \gamma \hat{z}^{[l]} + \beta$$

This keeps distributions stable across layers → faster, more reliable training.

✅ So, this diagram is showing **how distributions of hidden layer inputs keep shifting** (the scatterplots below), and why we need **batch normalization** to fix it.



Batch Norm as Regularization

- **Key idea**:

  Each **mini-batch** is normalized using the **mean (μ)** and **variance (σ²)** computed from just that mini-batch.

- **Effect**:
  - Adds some **noise** to activations $z^{[l]}$, since different mini-batches will have slightly different statistics.
  - This noise is **similar to dropout** → introduces randomness in hidden activations.
  - Hence, Batch Norm provides a **slight regularization effect**.
- **Takeaway**:

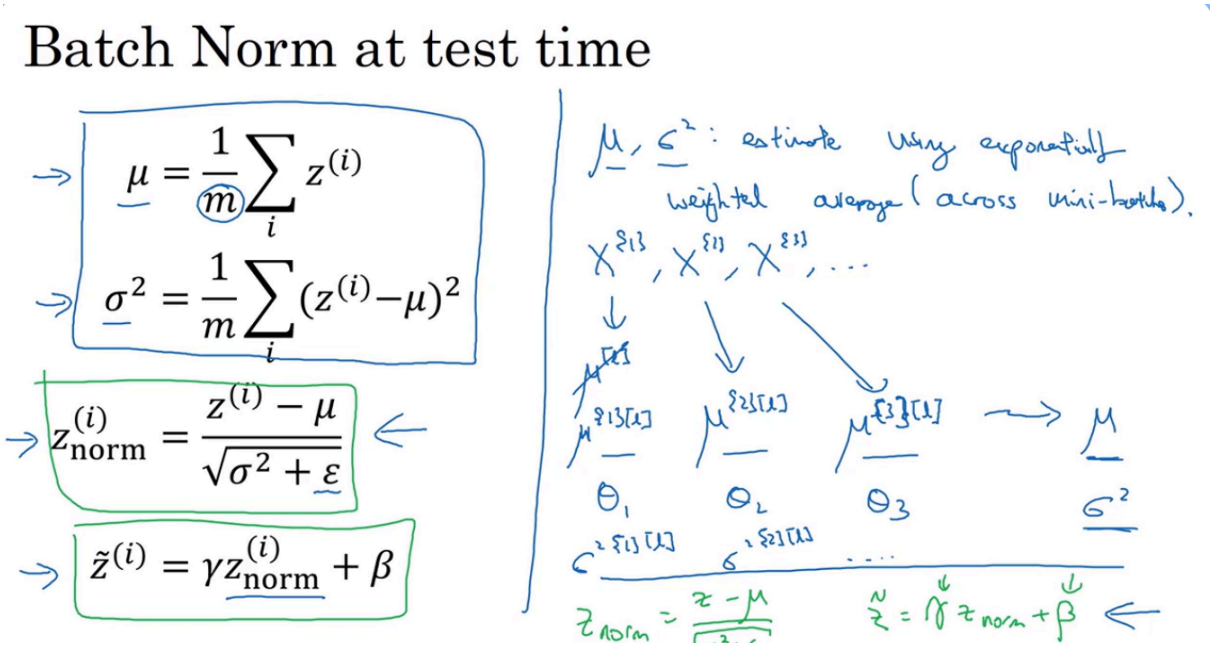
  Batch Norm not only speeds up training & improves convergence,

  but also acts like a **regularizer** (though weaker than dropout or L2).

⚡ Intuition:

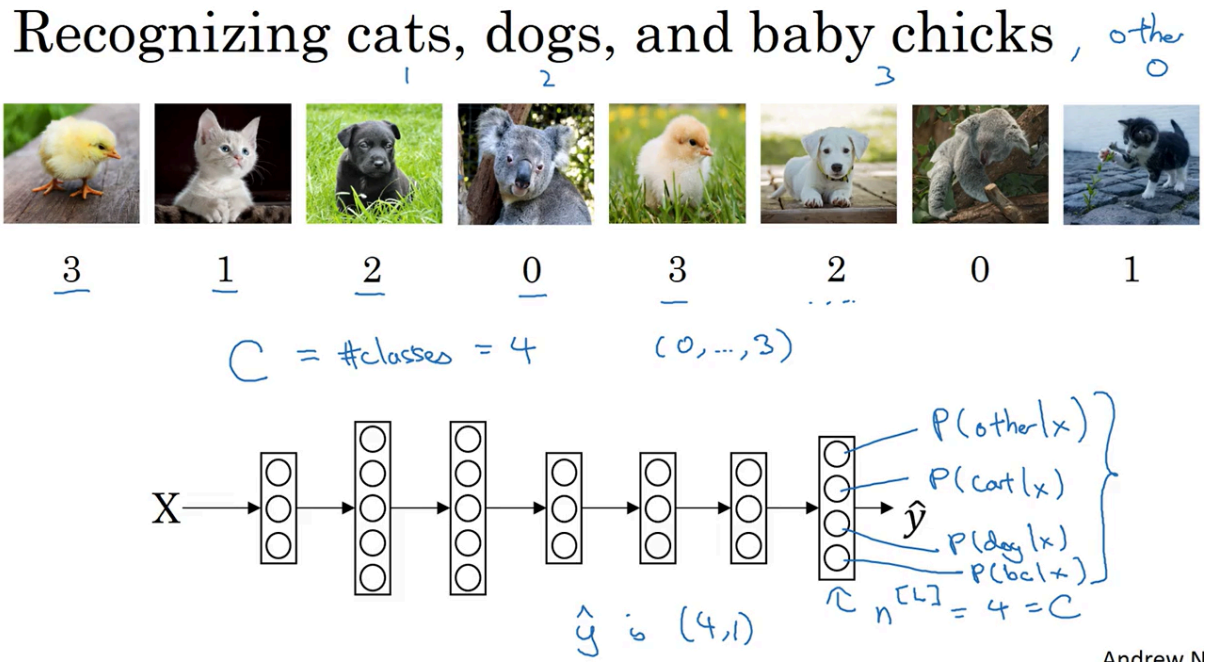
Because mini-batch statistics vary, the network cannot rely too much on any specific hidden unit's absolute scale → reducing overfitting slightly.

Batch Norm at Test Time



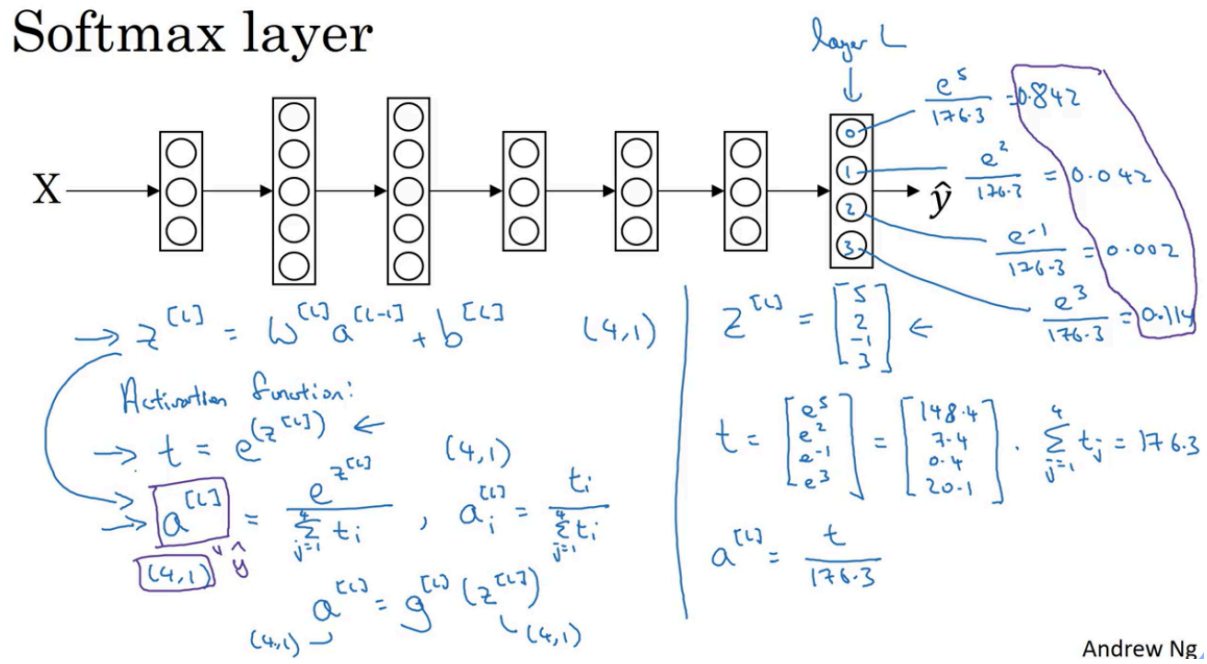
Softmax Regression

→ Multi- class classification



Andrew N

Softmax Layer



Andrew Ng

- **Final layer (multi-class classification):**
  - Linear step:

    $z^{[L]} = W^{[L]}a^{[L-1]} + b^{[L]}$

  - Compute exponentials:

    $t_i = e^{z_i^{[L]}}$

- Normalize to probabilities:

  $$a_i^{[L]} = \frac{e^{z_i^{[L]}}}{\sum_{k=1}^{K} e^{z_k^{[L]}}} \qquad \text{for } i = 1, \ldots, K$$

  where K = number of classes.

- **Example (4 classes):**

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}, \quad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$$

  - Denominator:

    $\sum_{i=1}^{4} t_i = 176.3$

  - Softmax output:

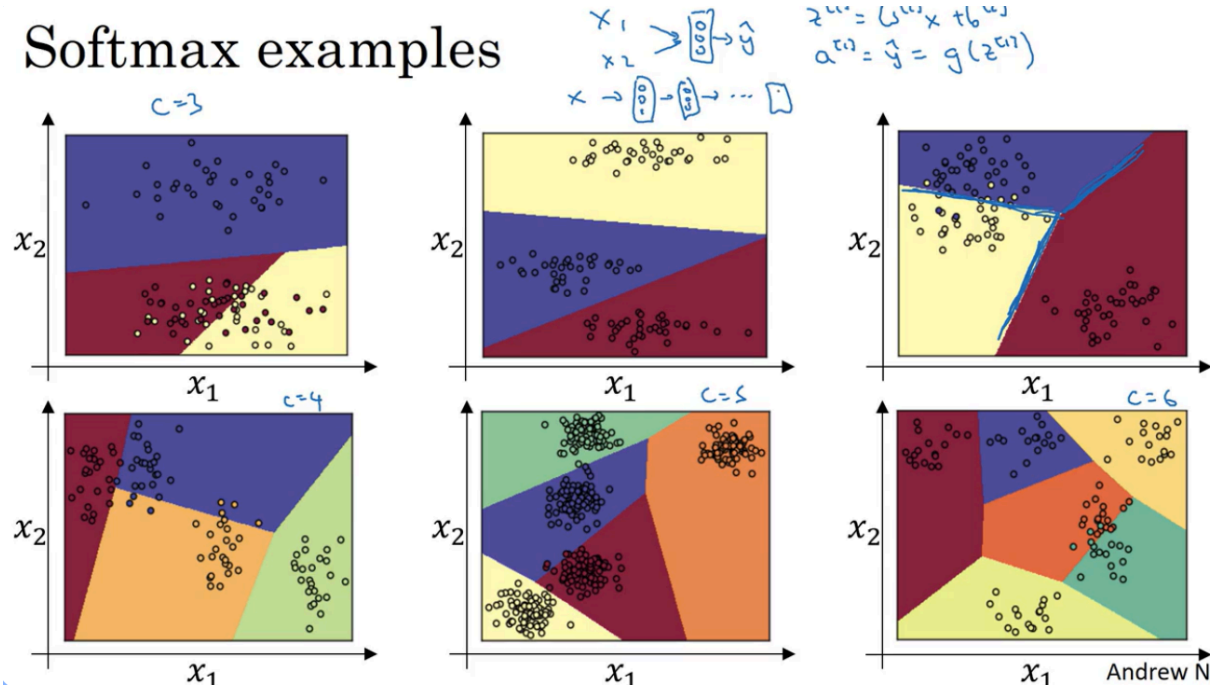
    $$a^{[L]} = \frac{1}{176.3} \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

⚡ **Key Points**:

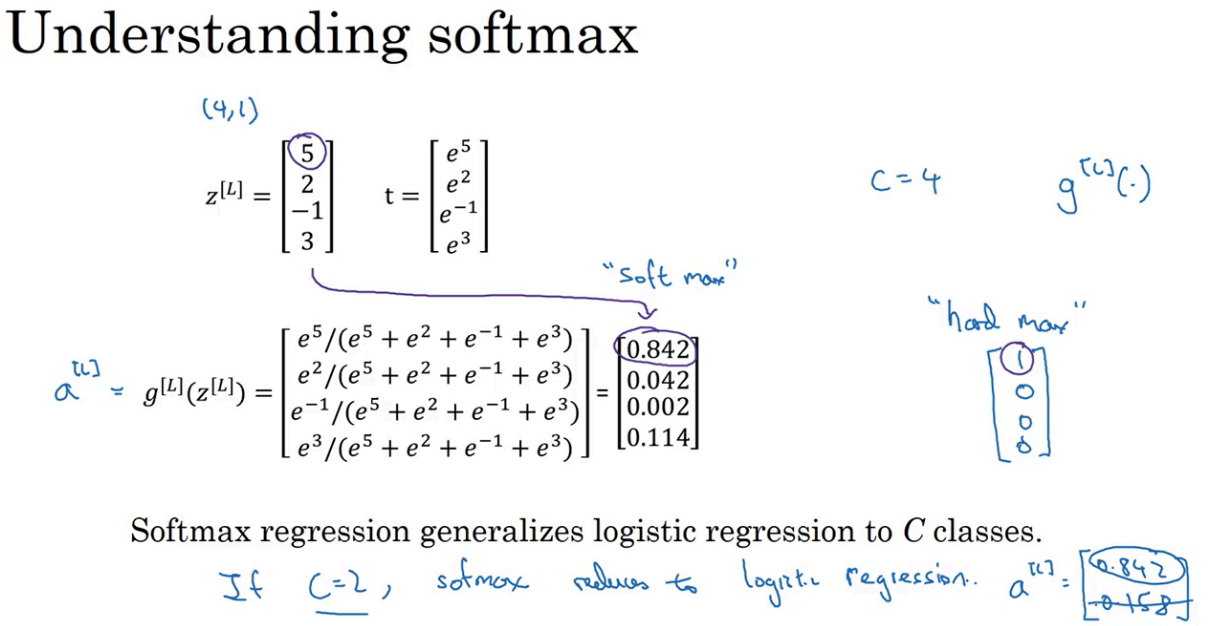
- Softmax converts logits z[L]z^{[L]}z[L] into a **probability distribution**.
- Outputs are always between **0 and 1** and **sum to 1**.
- Used in the **output layer of multi-class classification networks**.

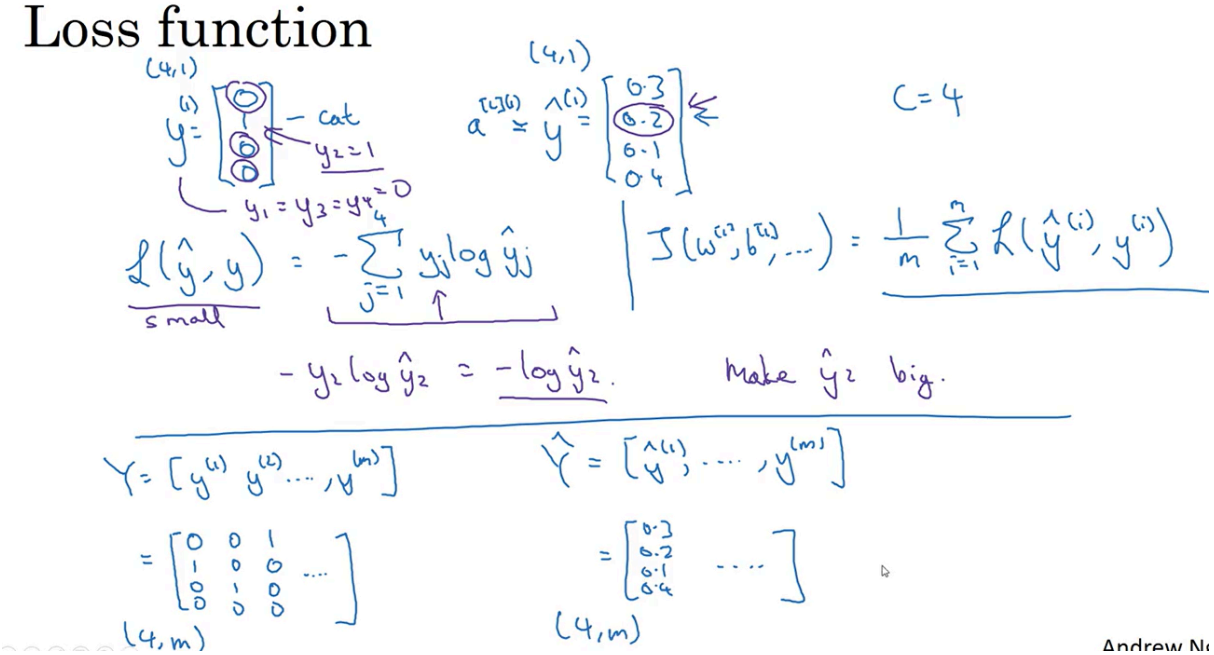
✅ Correction of the mistake you noticed:

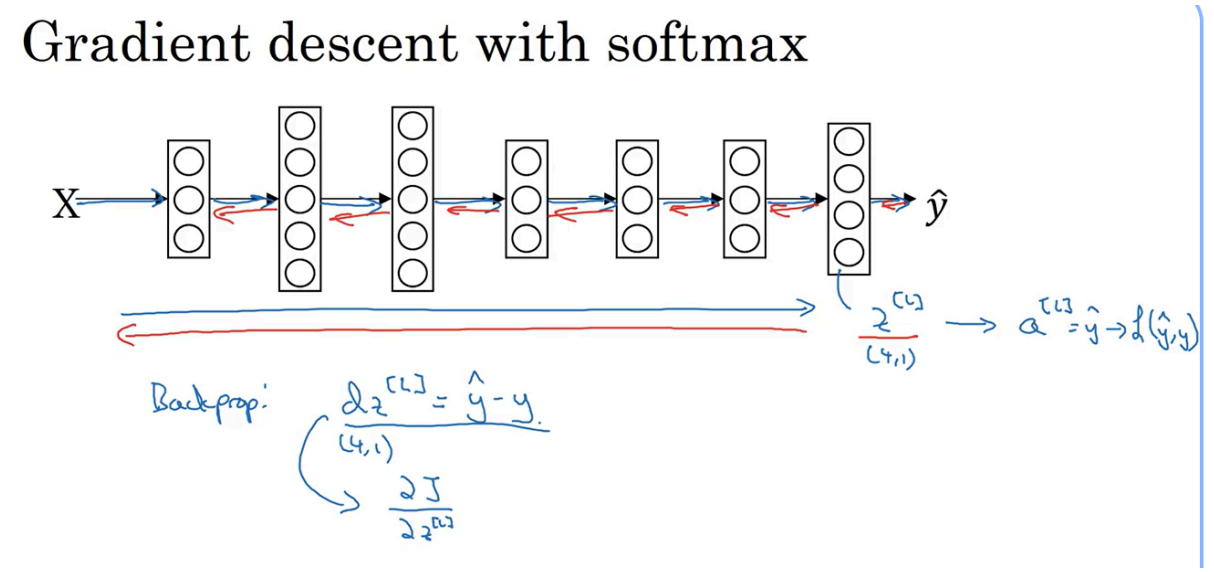
- The denominator must sum over **i (all classes)**, not **j**.



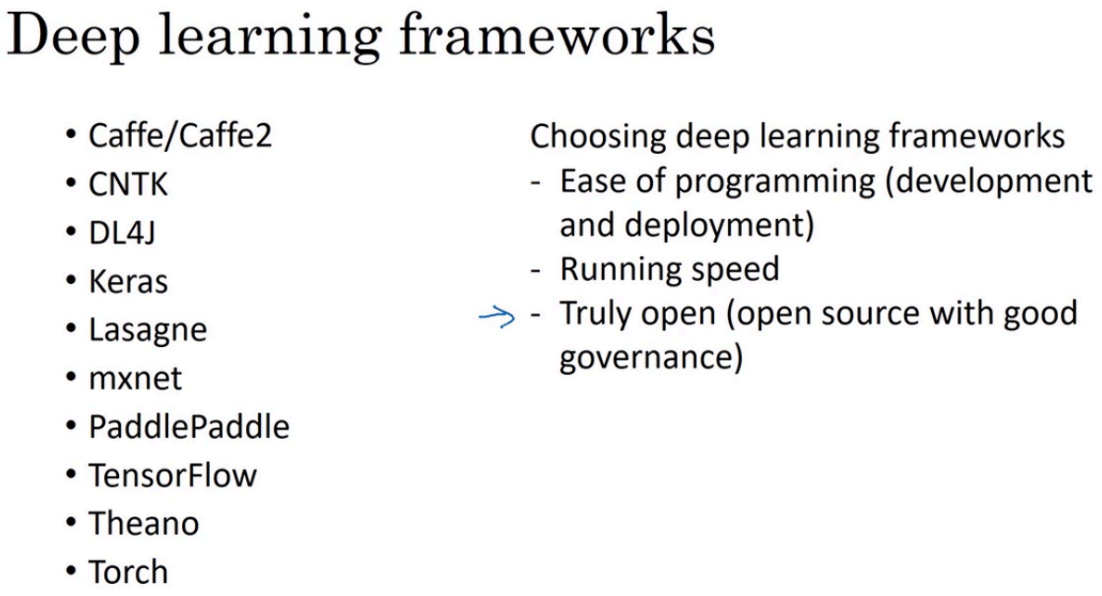
Training a Softmax Classifier







Deep Learning Frameworks



TensorFlow

```
import numpy as np
import tensorflow as tf
```

```
W = tf.variable(0, dtype = tf.float32)
optimizer = tf.keras.optimizers.Adam(0.1)

def train_step():
    with tf.GradientTape() as tape:
        cost = W ** 2 - 10 * W + 25
    trainable_variables = [W]
    grads = tape.gradient(cost, trainable_variables)
    optimizer.apply_gradients(zip(grads, trainable_variables))
print(W)
```

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=0.0>
```

```
train_step()
print(W)
```

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=0.099999308586
1206>
```

```
for i in range(1000):
  train_step()
print(W)
```

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=5.000000953674
316>
```