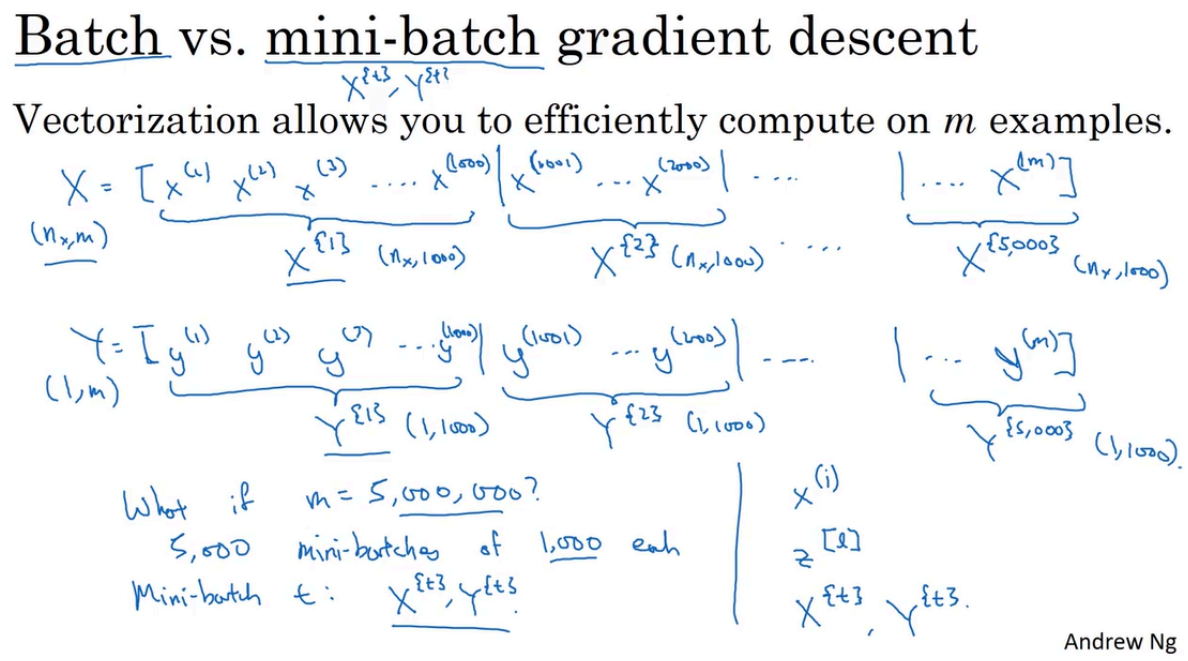
WEEK 2: Optimization Algorithms

Batch vs. Mini-Batch Gradient Descent



Key Concept

- **Vectorization** allows efficient computation on **m examples** at once.
- Instead of processing one example at a time, we use **matrix operations** for speed.

Batch Gradient Descent

- **All training examples** (m examples) are processed together.
- Dataset representation:
  - **Inputs**:

    $$X = [x^{(1)}, x^{(2)}, \dots, x^{(m)}] \quad \in \mathbb{R}^{(n_x, m)}$$

  - **Outputs**:

    $$Y = [y^{(1)}, y^{(2)}, \dots, y^{(m)}] \quad \in \mathbb{R}^{(1, m)}$$

- Very efficient with vectorization, but can be **slow** for very large datasets.

Mini-Batch Gradient Descent

- Instead of processing all examples at once, the dataset is split into **smaller batches**.
- Example: If $m = 5,000,000$:
  - We get **5000 mini-batches**.
- Each mini-batch:
  $X^{\{t\}}, Y^{\{t\}}$
  where $X^{\{t\}} \in \mathbb{R}^{(n_x,1000)}$, $Y^{\{t\}} \in \mathbb{R}^{(1,1000)}$.

Advantages of Mini-Batch

✅ Faster convergence compared to batch gradient descent.

✅ Makes better use of vectorization.

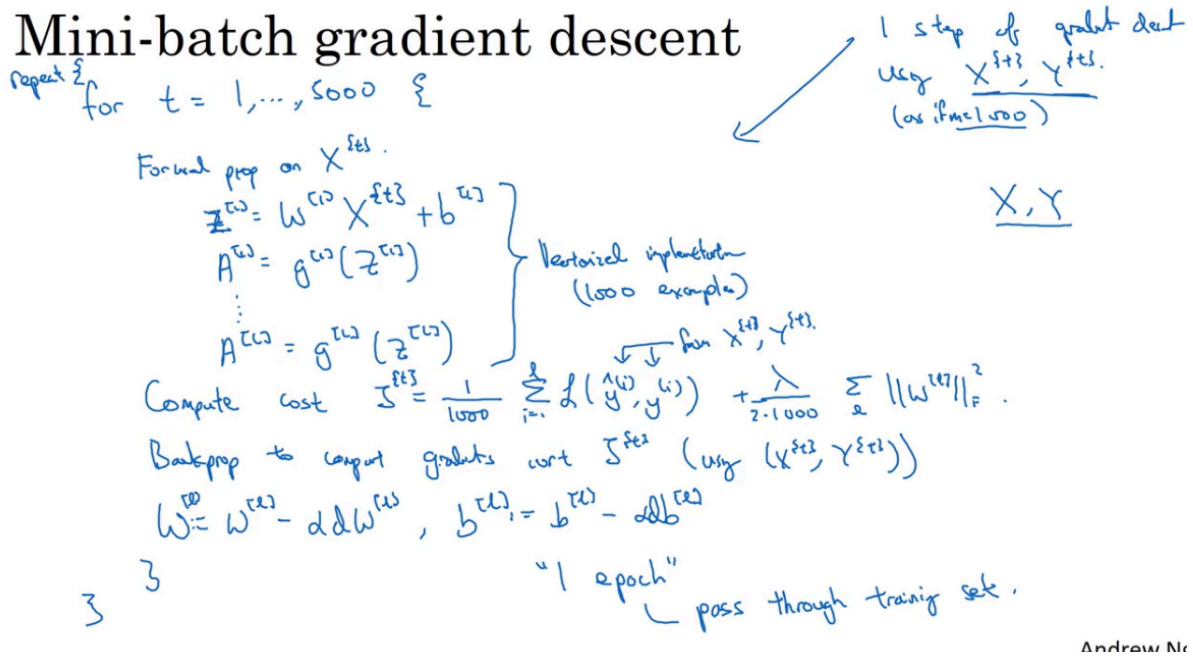
✅ Works well with large datasets.

✅ Reduces memory usage (processes smaller chunks).

Summary

- **Batch GD**: Uses the **whole dataset** per iteration (good for small datasets).
- **Mini-Batch GD**: Uses **subset of dataset** per iteration (good for large datasets, more efficient).
- Typical mini-batch size: **64, 128, 256, 512, 1024** (powers of 2 for efficiency).

Mini-Batch Gradient Descent (Detailed Steps)



Algorithm

Repeat for each mini-batch $t = 1, 2, \ldots, 5000$:

1. **Forward Propagation** on mini-batch $X^{\{t\}}$:
   - Linear step:
     $Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$
   - Activation:
     $A^{[1]} = g^{[1]}(Z^{[1]})$
   - Continue layer by layer until:
     $A^{[L]} = g^{[L]}(Z^{[L]})$

   ⚡ Done using **vectorized implementation** (e.g., 1000 examples per batch).

---

1. **Compute Cost for Mini-Batch tt:**
   $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{1000} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2 \cdot 1000} \sum_l \|W^{[l]}\|_F^2$
   - $L(\hat{y}, y)$ = loss function
   - Regularization term added

---

1. **Backpropagation**
   - Compute gradients using the mini-batch$\left(X^{\{t\}}, Y^{\{t\}}\right)$.

---

1. **Parameter Update**

   For each layer l:

   $W^{[l]} := W^{[l]} - \alpha\, dW^{[l]}$

   $b^{[l]} := b^{[l]} - \alpha\, db^{[l]}$

   - Here, α = learning rate

Epoch Definition

- **1 Epoch** = **one full pass** through the entire training set.
- If dataset has m=5,000,000 examples and

  batch size = 1000 →**5000 mini-batches per epoch**.

Key Points

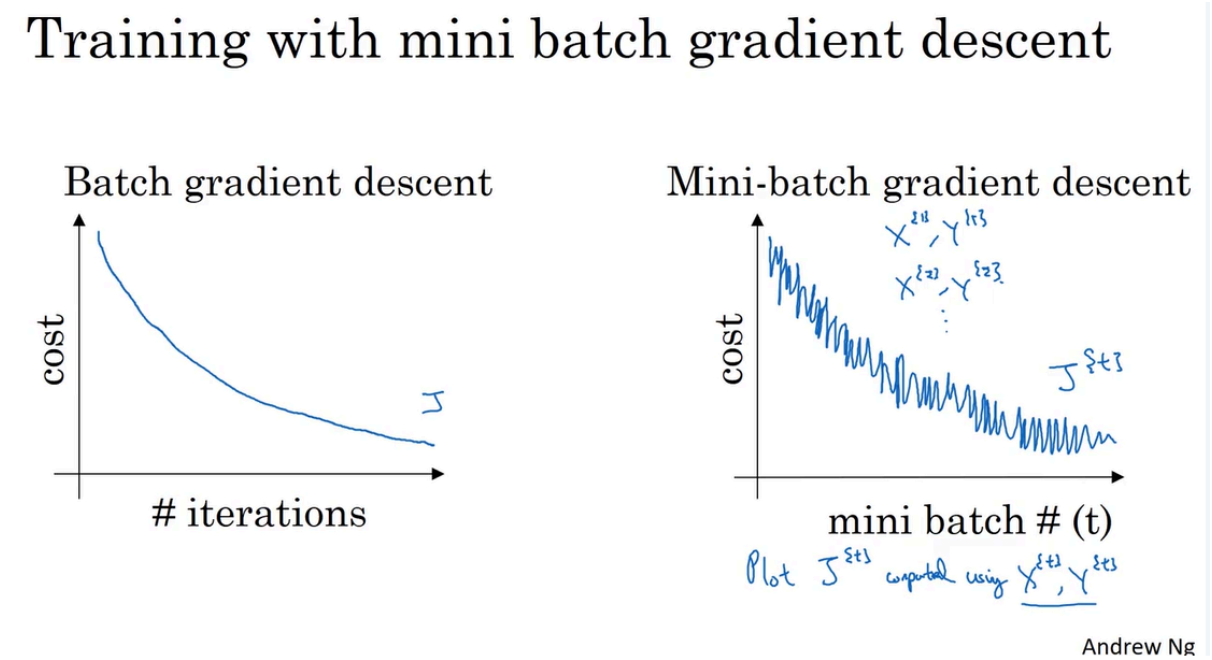
✅ Each mini-batch runs **1 step of gradient descent**.

✅ Cost function is computed on that mini-batch only.

✅ Reduces computation per step compared to batch GD.

✅ Helps smooth convergence compared to stochastic GD (which uses only 1 example).

Training with Mini-Batch Gradient Descent



1. Batch Gradient Descent

- Cost function J decreases **smoothly** over iterations.
- Each iteration computes gradient using the **entire dataset**.
- Curve is stable but can be **slow** for large datasets.

📉 Graph:

- **X-axis** → # iterations
- **Y-axis** → Cost
- Smooth downward slope.

---

2. Mini-Batch Gradient Descent

- Cost decreases **with oscillations**.
- Each mini-batch introduces **variance/noise** in the gradient.
- Faster updates since each step uses only a mini-batch.
- More efficient for large datasets.

📉 Graph:

- **X-axis** → Mini-batch number tt
- **Y-axis** → Cost

- Noisy but **overall trend is downward**.

Key Comparison

Aspect	Batch GD	Mini-Batch GD
Update per iteration	Uses **all m examples**	Uses **subset of examples** (batch size = 32, 64, 128, ...)
Cost curve	Smooth	Noisy but still decreases
Computation	Expensive for large datasets	More efficient, faster
Convergence	Stable, less variance	Faster, slight oscillations

Takeaway

- Batch GD → Good for **small datasets**.
- Mini-Batch GD → Best for **large datasets**, combines efficiency and convergence.
- Noise in mini-batch updates can help **escape local minima**.

Choosing Your Mini-Batch Size

1. Extreme Cases

- If **mini-batch size = m** → Equivalent to **Batch Gradient Descent**.
  - Very **stable** convergence.
  - But **too long per iteration** (computes over the entire dataset).
- If **mini-batch size = 1** → Equivalent to **Stochastic Gradient Descent (SGD)**.
  - Each example is its own mini-batch:
    $(X^{\{t\}}, Y^{\{t\}}) = (x^{(i)}, y^{(i)})$
  - Very **noisy updates**.
  - **Loses efficiency** from vectorization.

---

2. Practical Choice

- In practice, choose a mini-batch size **between 1 and m**.
- Benefits:

  ✅ **Fastest learning**

  ✅ **Efficient vectorization** (commonly 64, 128, 256, 512, 1024)

  ✅ **Good convergence** without processing entire dataset in one step

---

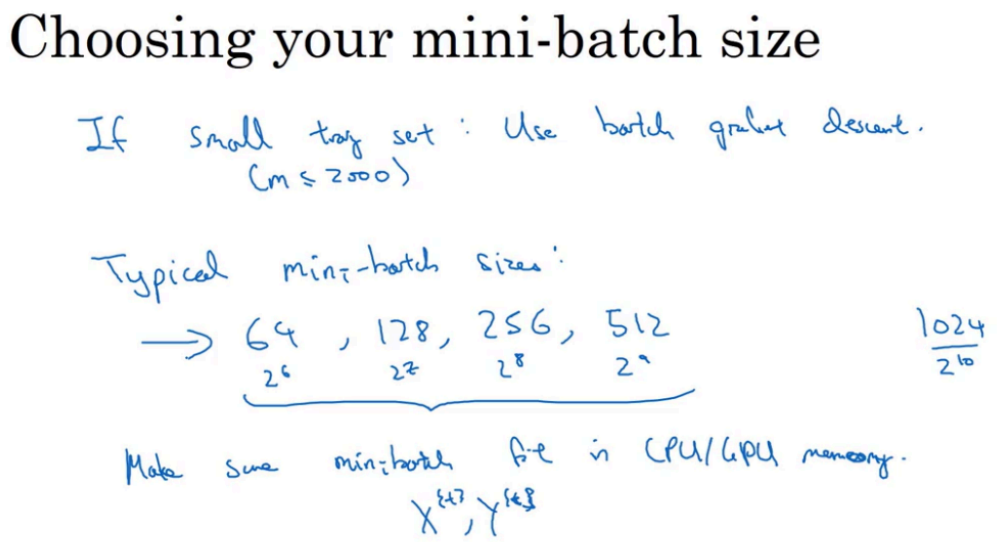
3. Visual Intuition

- **Batch GD (blue path)** → Smooth trajectory toward minimum, but slow.
- **Stochastic GD (purple path)** → Noisy zig-zag path, may take longer but sometimes escapes local minima.
- **Mini-Batch GD (green path)** → Balanced approach: fast convergence, some noise but efficient.

Summary

- **Batch GD** → Too slow for large datasets.
- **SGD** → Too noisy, inefficient without vectorization.
- **Mini-Batch GD** → Best of both worlds (balance between speed and accuracy).

Choosing Your Mini-Batch Size (Practical Tips)



1. If Dataset is Small

- If training set size is **small (m ≤ 2000)** → Use **Batch Gradient Descent** (process the entire dataset each step).

---

2. Typical Mini-Batch Sizes

- Common choices are powers of 2 (fit well in memory & parallelization):

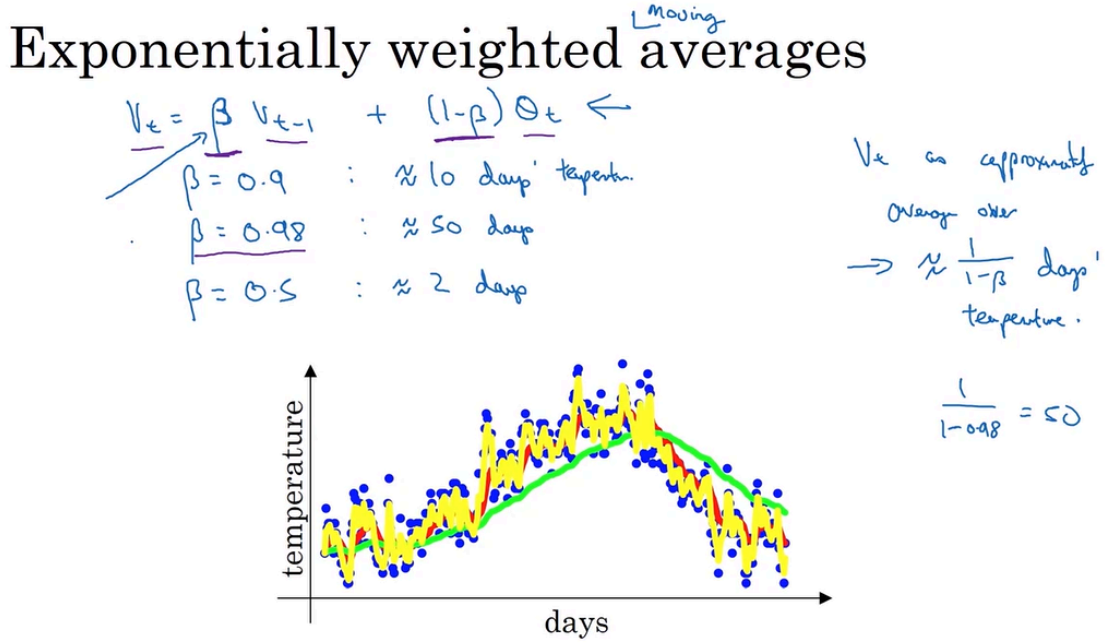
  64, 128, 256, 512, 1024

---

3. Practical Considerations

- **Make sure mini-batches fit in CPU/GPU memory**.
- Larger mini-batches use more memory, but too small batches lose efficiency.
- Usually, **128 → 512** is a good default range.

Exponentially Weighted Averages (EWA)



Formula

$V_t = \beta V_{t-1} + (1 - \beta)\theta_t$

- V_t: Smoothed value at time t
- θ_t: Observation at time t
- β: Weight factor (close to 1 = smoother curve, longer memory)

Effect of β\beta

- $\beta = 0.9$ $\Rightarrow$averages over ~10 days
- $\beta = 0.98$ $\Rightarrow$averages over ~50 days
- $\beta = 0.5$ $\Rightarrow$ averages over ~2 days

Interpretation

- V_t approximates the moving average over:

$$\frac{1}{1-\beta} days$$

- Example: $\beta = 0.98 \Rightarrow \frac{1}{1-0.98} = 50 days$

Graph Insight

- **Blue dots** = noisy daily temperature readings
- **Yellow line** = shorter smoothing (less stable)
- **Green line** = stronger smoothing (captures long-term trend)

📌 **Key Takeaway:**

Exponentially weighted averages smooth out noise in data. Larger β gives smoother trends but reacts slower to changes.

Why Use Exponentially Weighted Averages (EWA)

- **1. Smooths Noisy Data (Filtering Effect)**

  Instead of raw, fluctuating signals, we create a *smoothed estimate*:

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

  where:

  - v_t = smoothed value at time t
  - θ_t = actual observation at time t
  - $\beta \in [0, 1]$ = smoothing factor

- **2. Captures Recent Trends Without Forgetting the Past**
  - Recent data gets *more weight*.
  - Old data decays *exponentially*.

    Example expansion:

$v_t = (1 - \beta)\theta_t + (1 - \beta)\beta\theta_{t-1} + (1 - \beta)\beta^2\theta_{t-2} + \cdots$

→ Weight of observation k steps back: $(1 - \beta)\beta^k$.

- **3. Avoids Equal Weighting Problem in Simple Moving Averages**
  - In a simple moving average (SMA), all past values in a window are weighted equally, then suddenly discarded.
  - EWA fixes this by making the weight **continuous and decaying smoothly**.

- **4. Enables Memory-Efficient Computation**
  - Only last state v_{t-1} needs to be stored.
  - Time complexity: $O(1)$per update.

    Perfect for **online learning** & **streaming data**.

- **5. Foundation for Optimization Algorithms**
  - Used in **Momentum**, **RMSProp**, **Adam** optimizers.
  - Example:

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

    where g_t is the gradient at step t.

- **6. Bias Correction**
  - Early values are biased towards zero (since history is short).
  - Correction:

    $\hat{v}_t = \frac{v_t}{1-\beta^t}$
  - This makes estimates unbiased, crucial in optimizers like **Adam**.

✅ **Summary :**

Exponentially Weighted Averages provide a mathematically principled way to **smooth data, prioritize recent history, reduce variance, and build efficient online estimators**. Their recursive structure makes them the backbone of **modern optimization algorithms**.

Exponentially Weighted Averages – Derivation

1. Recursive Definition

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

---

2. Expanding Recursively

Example ($\beta = 0.9$):

$v_{100} = 0.9v_{99} + 0.1\theta_{100}$

$v_{99} = 0.9v_{98} + 0.1\theta_{99}$

$v_{98} = 0.9v_{97} + 0.1\theta_{98}$

Substitute backward:

$v_{100} = 0.1\theta_{100} + 0.1(0.9)\theta_{99} + 0.1(0.9)^2\theta_{98} + \cdots$

---

3. Weighting of Past Observations

- Weight of an observation k steps in the past:
  $(1-\beta)\beta^k$
- Shows **exponential decay** → recent values dominate.

---

4. Effective Window Size

Approximate number of terms contributing significantly:

$$\frac{1}{1-\beta}$$

- If $\beta = 0.9 \Rightarrow$ 10 days average
- If $\beta = 0.98 \Rightarrow 50\ days\ average$
- If $\beta = 0.5 \Rightarrow$ $2\ days\ average$

---

5. Exponential Approximation

Using the limit:

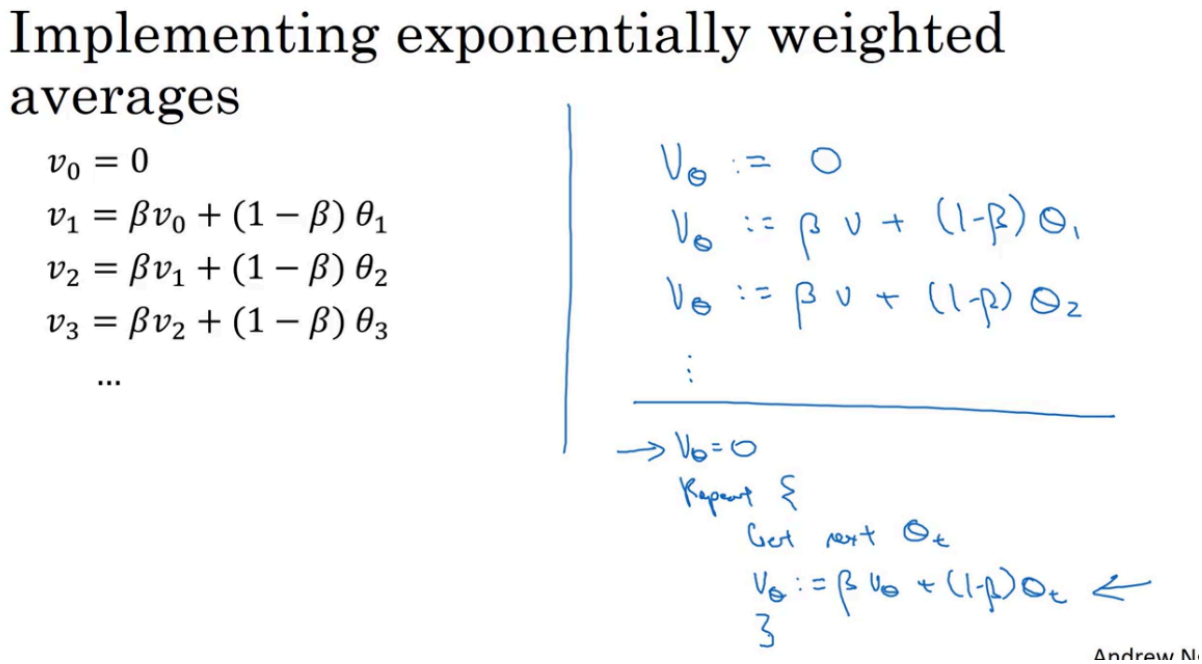
$\beta^k \approx e^{-k(1-\beta)}$

This connects EWA to **continuous-time exponential decay**.

Core takeaway for notes:

EWA is mathematically equivalent to taking a weighted moving average where weights decay *exponentially*. The parameter β controls the "memory length," approximately $\frac{1}{1-\beta}$.

Implementation of Exponentially Weighted Averages (EWA)



Recursive Definition

$v_t = \beta v_{t-1} + (1-\beta)\,\theta_t, \quad v_0 = 0$

- v_t: exponentially weighted average at step t
- θ_t: new observation (e.g., temperature, gradient, etc.)
- $\beta \in [0, 1)$: controls memory length (higher β → longer memory)

Step-by-step Expansion

- For t=1:

$v_1 = \beta v_0 + (1-\beta)\theta_1$

- For t=2:

$v_2 = \beta v_1 + (1-\beta)\theta_2$

- For t=3:

$v_3 = \beta v_2 + (1-\beta)\theta_3$

...and so on.

Algorithm (pseudocode)

Initialize: v = 0
For each new data point ▶:
  v ← β * v + (1 - β) * θ_t

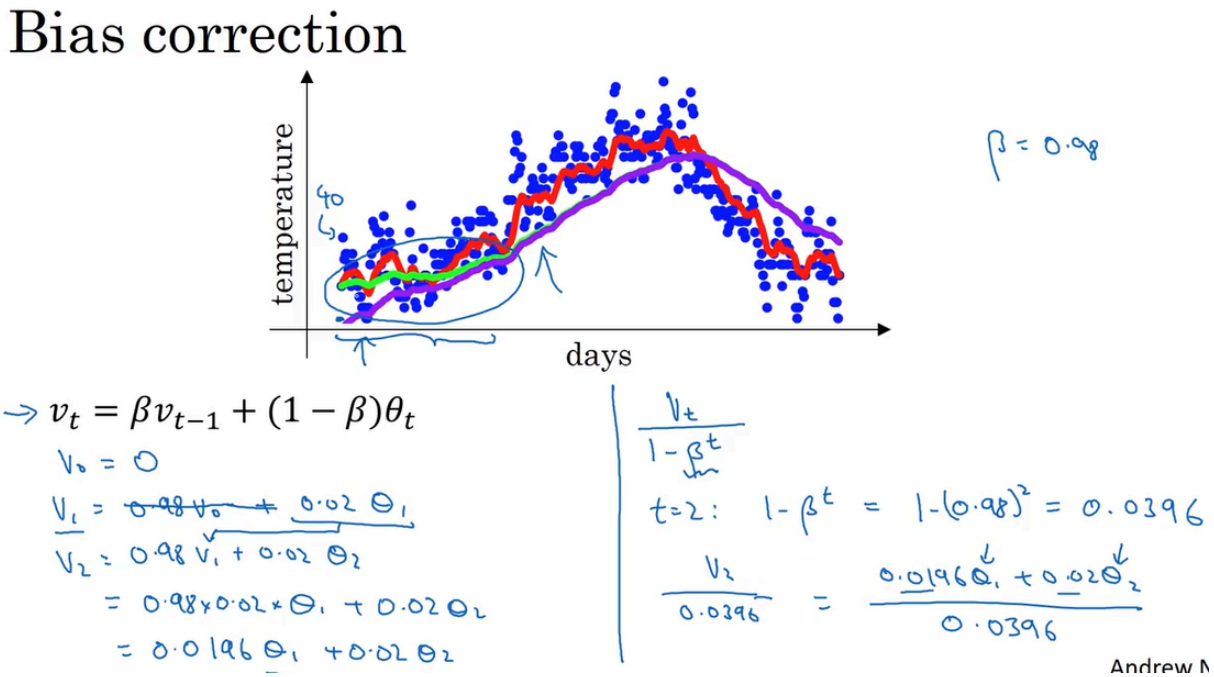
Bias-Corrected Form (important at early t)

$\hat{v}_t = \frac{v_t}{1-\beta^t}$

Intuition

- v_t is a **running average** of the past θ\theta's.
- Effective memory ≈ $\frac{1}{1-\beta}$.
- As $t \to \infty$, bias vanishes automatically, but correction helps at small tt.

Bias Correction in Exponentially Weighted Averages



We start with the standard recursion:

$v_t = \beta v_{t-1} + (1-\beta)\theta_t, \quad v_0 = 0$

Problem: Initialization Bias

- At the beginning, $v_0 = 0$.
- For small t, the moving average is **biased toward 0**, because not enough past terms have accumulated.
- Example ($\beta = 0.98$):

$v_1 = 0.02\theta_1$

$v_2 = 0.98 \cdot 0.02\theta_1 + 0.02\theta_2 = 0.0196\theta_1 + 0.02\theta_2$

These weights are **too small** compared to the true exponential distribution.

Correction Formula

To fix this bias, we normalize:

$\hat{v}_t = \frac{v_t}{1-\beta^t}$

- Here, $1-\beta^t$ accounts for the missing mass at early timesteps.
- As $t \to \infty$, $\beta^t \to 0$, so $1-\beta^t \to 1$, and correction becomes unnecessary.

Example

At t=2, with $\beta = 0.98$:

$1-\beta^2 = 1-(0.98)^2 = 0.0396$

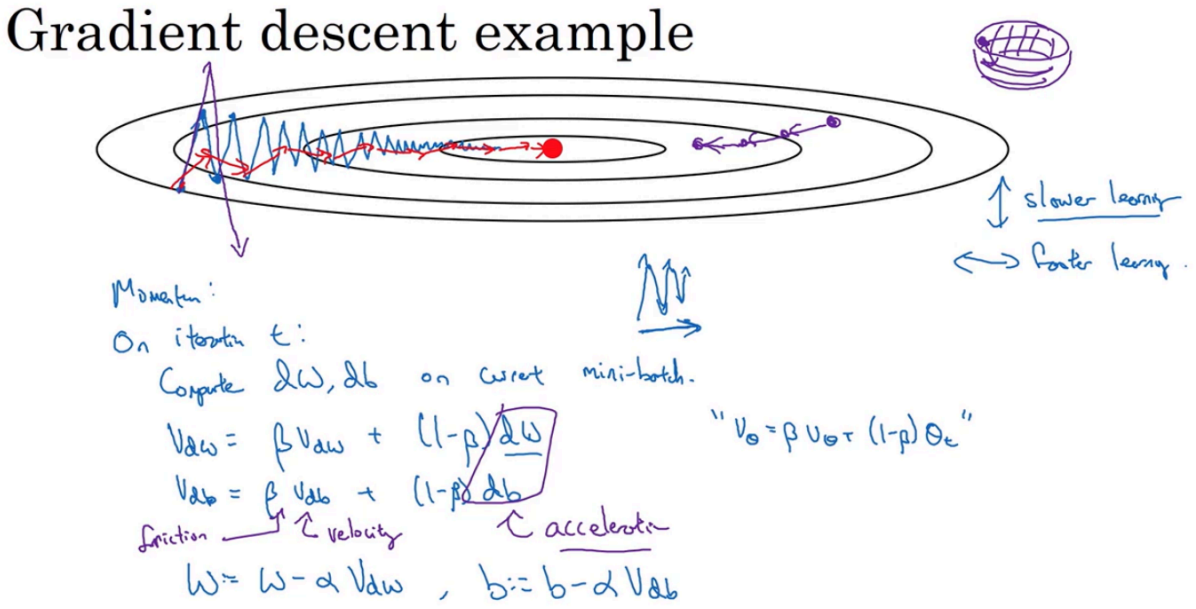
$\hat{v}_2 = \frac{v_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$

This rescales the weights so they sum correctly, removing bias.

Key Idea

- Without correction → underestimation at early timesteps.
- With correction → unbiased exponential moving average.
- This is **exactly what Adam optimizer uses** in deep learning for momentum and RMSprop.

Gradient Descent with Momentum



Intuition

- Standard Gradient Descent may oscillate a lot, especially in narrow valleys of the cost function.
- **Momentum** helps smooth the path and speeds up convergence by adding "velocity" to the updates.
- Think of it as a ball rolling down a hill:
  - Gradient = slope → pushes the ball.
  - Momentum = velocity → keeps it moving in the right direction.
  - Friction (via β) prevents overshooting.

Update Rules

On iteration t:

1. **Compute gradients** on current mini-batch:

   $dW,\ db$

2. **Update velocities:**

$$V_{dW} = \beta V_{dW} + (1 - \beta) dW$$

$$V_{db} = \beta V_{db} + (1 - \beta) db$$

- β → momentum hyperparameter (e.g., 0.9)
- Acts like a smoothing factor (Exponential Weighted Average).
- V accumulates past gradients → provides "direction memory".

3. **Update parameters:**

$$W := W - \alpha V_{dW}$$

$$b := b - \alpha V_{db}$$

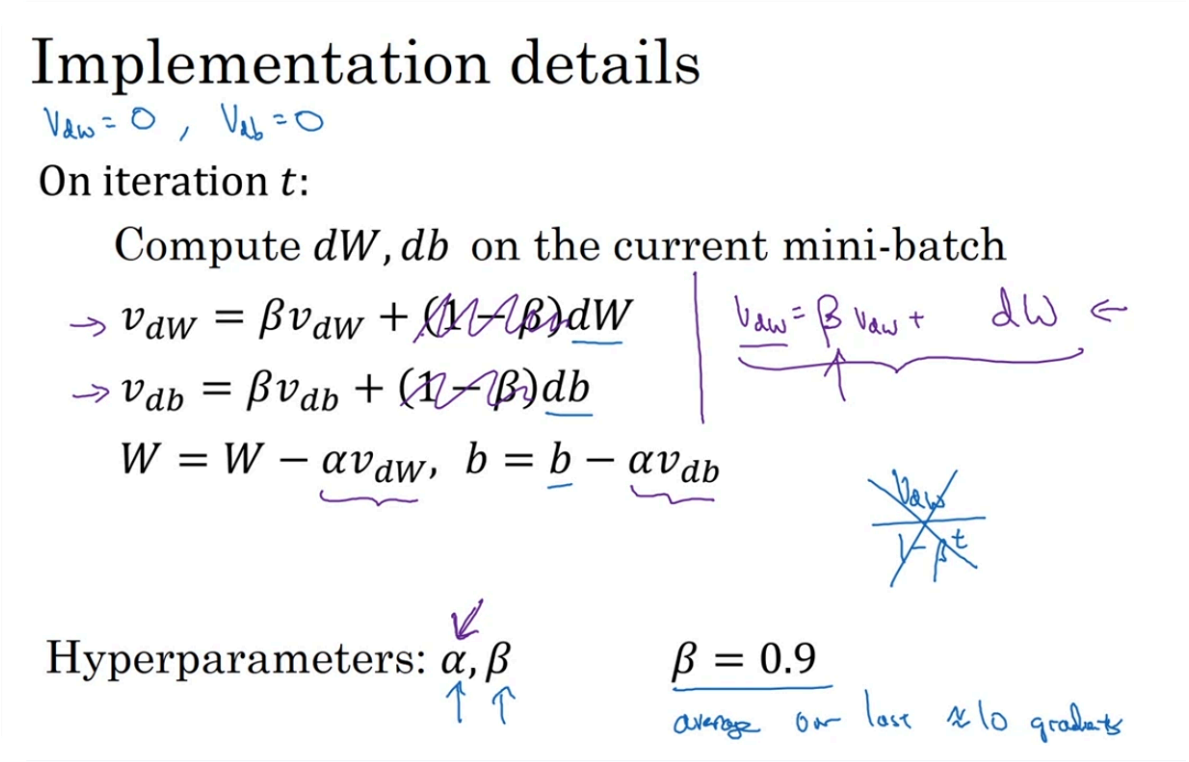
- α → learning rate.
- Updates follow the **smoothed gradient**, reducing oscillation.

Analogy

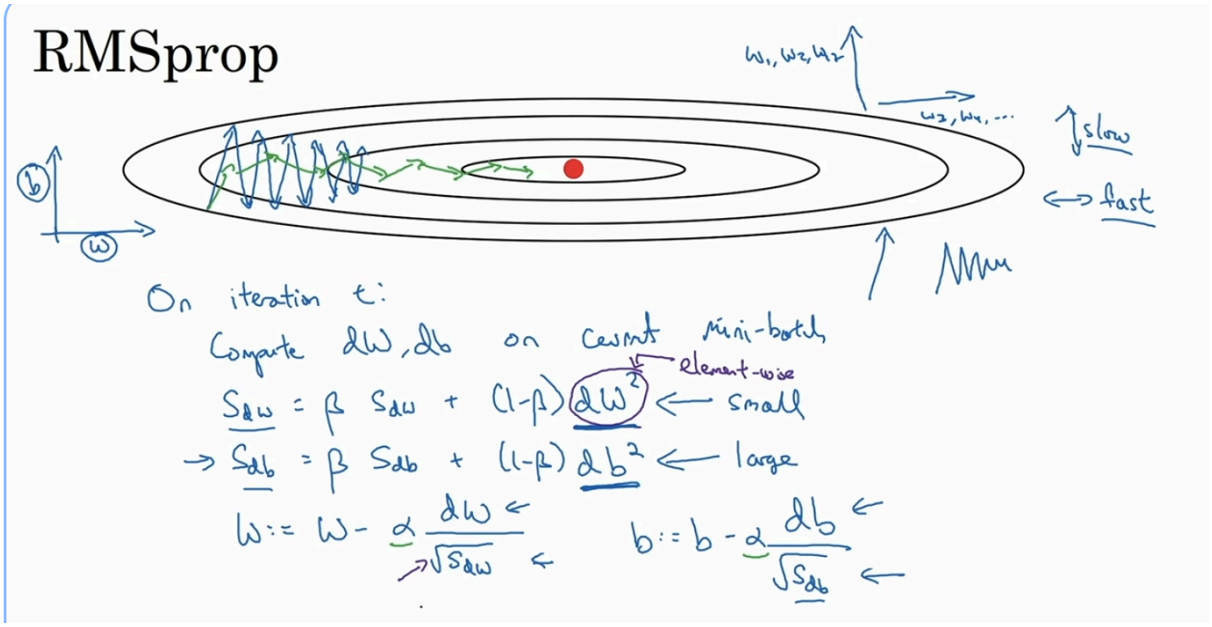
- Imagine rolling down a bowl:
  - Without momentum → small zig-zag steps (slow).
  - With momentum → smoother, faster path to the minimum.

✅ **Key takeaway**:

Momentum accelerates gradient descent, reduces oscillations, and helps escape narrow valleys.



⚡ RMSprop (Root Mean Square Propagation)



🔬 Intuition

- Gradient Descent may **oscillate** when one direction (say w_1) has **steep curvature** (large gradients), and another (w_2) has **flat curvature** (small gradients).
- Standard momentum helps but still struggles when scales differ across dimensions.

👉 RMSprop solves this by:

1. Scaling the learning rate **independently per parameter**.
2. Dividing each update by a **running average of squared gradients**.
3. This prevents very large updates in steep directions, while allowing bigger steps in flat directions.

📐 Mathematical Formulation

On iteration t:

1. **Compute gradients:**

$dW,\ db$

1. **Update running average of squared gradients:**

$S_{dW} = \beta S_{dW} + (1-\beta)(dW)^2$

$S_{db} = \beta S_{db} + (1-\beta)(db)^2$

- Here, squaring is **element-wise**.
- $\beta \approx 0.9$ (decay rate).
- This gives a **smoothed moving average** of squared gradients.

1. **Parameter update (scaled by RMS):**

$W := W - \alpha \frac{dW}{\sqrt{S_{dW}} + \epsilon}$

$b := b - \alpha \frac{db}{\sqrt{S_{db}} + \epsilon}$

- ϵ (like 10^{-8}) avoids division by zero.
- Effective learning rate is **adaptive per dimension**.

⚙️ Implementation Details

- **Initialize:**

$$S_{dW} = 0, \quad S_{db} = 0$$

- **At each step:**
  - Compute gradients.
  - Update squared moving averages.
  - Update parameters using scaled learning rates.

🎯 Hyperparameters

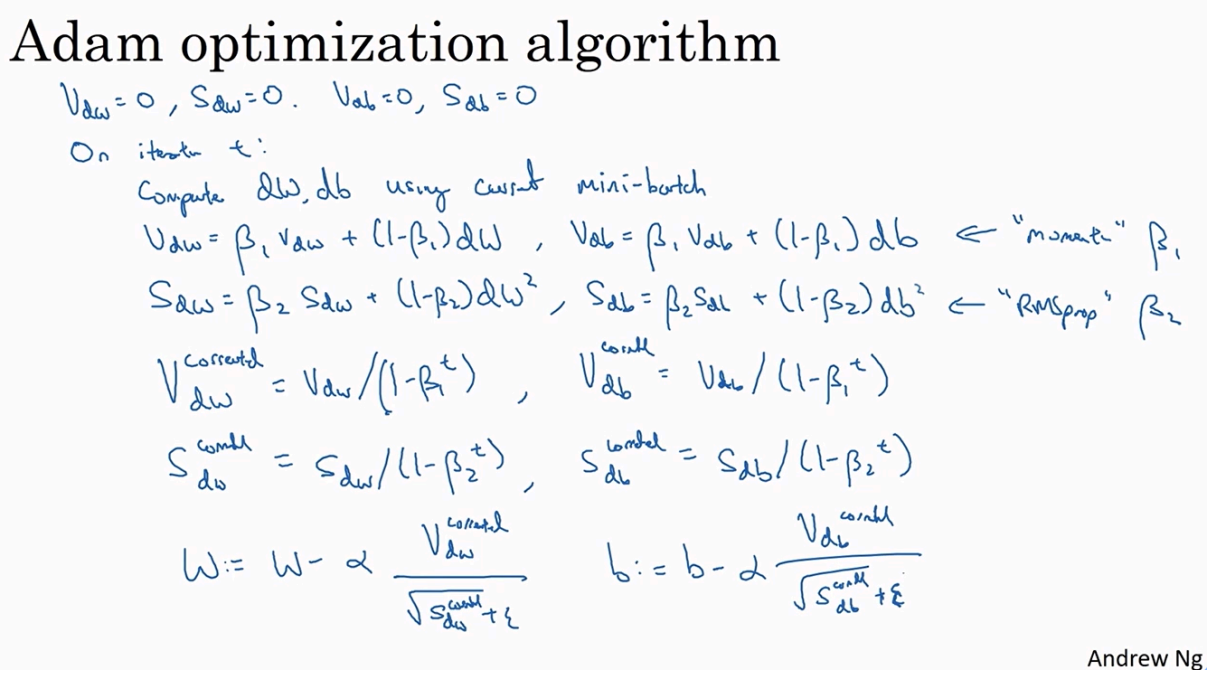
- Learning rate: α (usually smaller, like 0.001).
- Decay rate: β=0.9.
- Numerical stability: $\epsilon = 10^{-8}$.

✅ Key Takeaways

- RMSprop **adapts learning rate per parameter**.
- Prevents oscillations in steep directions.
- Faster convergence compared to plain SGD or momentum.
- Basis of **Adam Optimizer** (which combines RMSprop + Momentum).

Adam Optimization Algorithm

(Adaptive Movement Estimation)



Initialization

$$V_{dW} = 0, \quad S_{dW} = 0, \quad V_{db} = 0, \quad S_{db} = 0$$

Iteration t:

Step 1: Compute gradients

$$dW, \; db$$

Step 2: Momentum (1st moment estimate)

$$V_{dW} = \beta_1 V_{dW} + (1 - \beta_1) dW$$

$$V_{db} = \beta_1 V_{db} + (1 - \beta_1) db$$

Step 3: RMSprop (2nd moment estimate)

$$S_{dW} = \beta_2 S_{dW} + (1 - \beta_2)(dW^2)$$

$$S_{db} = \beta_2 S_{db} + (1 - \beta_2)(db^2)$$

Step 4: Bias correction

$$\hat{V}_{dW} = \frac{V_{dW}}{1-\beta_1^t}, \quad \hat{V}_{db} = \frac{V_{db}}{1-\beta_1^t}$$

$$\hat{S}_{dW} = \frac{S_{dW}}{1-\beta_2^t}, \quad \hat{S}_{db} = \frac{S_{db}}{1-\beta_2^t}$$

Step 5: Parameter update

$$W := W - \alpha \frac{\hat{V}_{dW}}{\sqrt{\hat{S}_{dW}} + \epsilon}$$

$$b := b - \alpha \frac{\hat{V}_{db}}{\sqrt{\hat{S}_{db}} + \epsilon}$$

Hyperparameters

- α=0.001 (default learning rate)
- β1=0.9 (momentum term)
- β2=0.999 (RMSprop term)
- ϵ=10^{-8} (stability)

✅ Adam = Momentum + RMSprop + Bias correction

⚡ Works well in practice, default settings often sufficient.

📌 Most widely used optimizer in Deep Learning.

Learning Rate Decay

🔑 Key Idea:

When we train with **gradient descent**, we usually pick a **fixed learning rate (α)**.

But:

- If α is **too large** → the algorithm may overshoot → fail to converge.
- If α is **too small** → the algorithm converges but **very slowly**.

👉 The trick is: **start with a relatively larger learning rate**, then **gradually reduce it** as training progresses.

This is called **Learning Rate Decay**.

🧠 Intuition (Andrew Ng style):

Imagine you are **rolling a ball into a valley**:

- At the start, the ball is far away → you want **big steps** to reach quickly.
- As you approach the bottom, you want **smaller steps** → so you don't overshoot.

📉 So, we **decay the learning rate over time**.

🧮 Common Decay Formulas

1. **Exponential Decay**

$\alpha = \alpha_0 \cdot e^{-kt}$

- α_0: initial learning rate
- k: decay rate
- t: epoch (iteration number)

1. **Inverse Decay** (most common in practice)

$\alpha = \frac{\alpha_0}{1+kt}$

1. **Step Decay**

   Every few epochs, reduce α by a factor. Example:

- Every 10 epochs → α = α × 0.

⚙️ Coding Intuition (mini example with MNIST)

```
import tensorflow as tf

# Initial learning rate
initial_lr = 0.1

# Exponential decay
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=initial_lr,
    decay_steps=1000,
    decay_rate=0.96,
    staircase=True  # if True → step decay
)
```

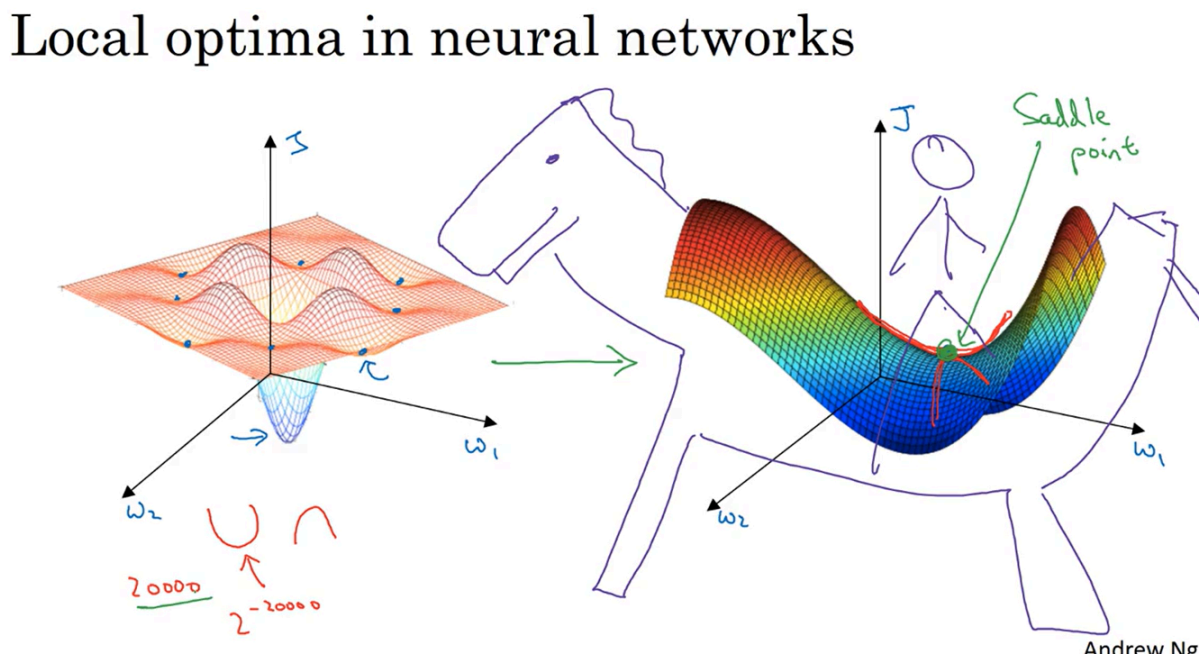
```
optimizer = tf.keras.optimizers.SGD(learning_rate=lr_schedule)
```

Here, TensorFlow automatically updates the learning rate as training goes on.

✅ Takeaways (like Andrew Ng's summary):

- Learning rate is one of the **most important hyperparameters**.
- Fixed α can be inefficient.
- **Decay strategies** help you:
  - Move fast at the beginning,
  - Converge smoothly at the end.

The Problem of Local Optima



🔎 Breaking Down the Slide

Left side (Orange surface)

- The axes are **weights** w_1, w_2.

- The vertical axis is the **cost function** $J(w_1, w_2)$.
- The surface has **many dips and bumps** → shows that the cost function in neural networks is **non-convex**.
- The blue dots represent possible **local minima**.
- But in **high dimensions**, the chance of being stuck in a bad local minimum is very small.

Right side (Horse Saddle analogy 🐎)

- Neural networks often have **saddle points**, not bad local minima.
- A **saddle point** is flat in one direction (like sitting in a horse saddle), but slopes down in another.
- The person (stick figure) at the saddle point → gradient is near zero → gradient descent can **slow down a lot**.
- That's why training sometimes looks like it's "stuck".

🧠 Intuition from Andrew Ng

- In **2D toy examples**, it looks like local minima are everywhere.
- But in **high dimensions (like 20,000+ parameters in neural nets)**, local minima are rare.
- Instead, you mostly see **plateaus and saddle points**.

✅ Takeaway (like Andrew's summary points)

- Don't worry too much about local optima.
- The real difficulty is escaping **saddle points** where learning slows down.
- Optimizers like **Momentum, RMSProp, Adam** help push through these flat regions.

Quiz important notes:

Let's compute the values step by step.

Given:

- $\left(\theta_1 = 30^\circ C\right)$ (March 1st)
- $\left(\theta_2 = 15^\circ C\right)$ (March 2nd)
- $\left(\beta = 0.5\right)$
- $\left(v_0 = 0\right)$
- The formula without bias correction: $(v_t = \beta v_{t-1} + (1-\beta)\theta_t)$
- The bias-corrected value: $(v_t^{\text{corrected}} = \frac{v_t}{1-\beta^t})$

We need to find (v_2) (without bias correction) and $(v_2^{\text{corrected}})$ (with bias correction).

Step 1: Compute (v_1) (without bias correction)
$\left[v_1 = \beta v_0 + (1-\beta)\theta_1 = (0.5 \times 0) + (1-0.5) \times 30 = 0 + 0.5 \times 30 = 15\right]$

Step 2: Compute (v_2) (without bias correction)
$\left[v_2 = \beta v_1 + (1-\beta)\theta_2 = (0.5 \times 15) + (1-0.5) \times 15 = 7.5 + 0.5 \times 15 = 7.5 + 7.5 = 15\right]$
So, $(v_2 = 15)$.

Step 3: Compute $(v_2^{\text{corrected}})$ (with bias correction)
The bias correction formula is $(v_t^{\text{corrected}} = \frac{v_t}{1-\beta^t})$.
For (t=2):
$\left[1 - \beta^2 = 1 - (0.5)^2 = 1 - 0.25 = 0.75\right]$
$\left[v_2^{\text{corrected}} = \frac{v_2}{1-\beta^2} = \frac{15}{0.75} = 20\right]$

Therefore:

- $(v_2 = 15)$
- $(v_2^{\text{corrected}} = 20)$