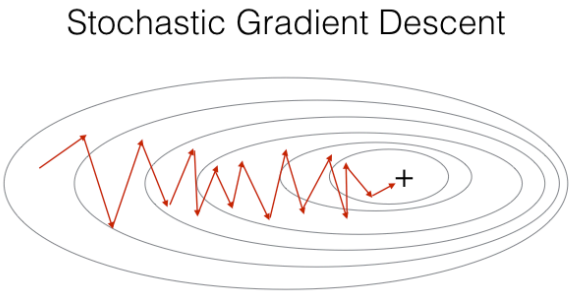
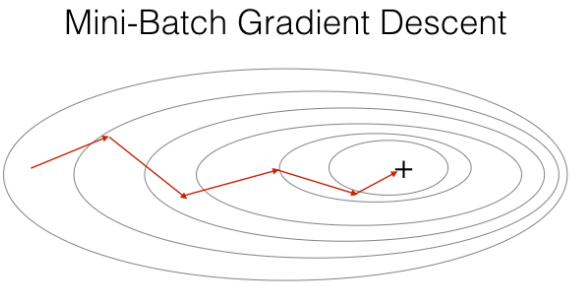
2025-08-08T20-12-05
Tags: DLS

DLS C2 week 2

Mini-batch gradient descent





Batch gradient descent	• Processes the entire train set before taking one gradient step.
Mini-batch gradient descent	• Splits the dataset into mini-batches and takes a gradient step after each mini-batch.

Notation	Description	Shape
$x^{(i)}$	i-th train example	$(n_x, 1)$
$z^{[l]}$	linear activation at layer l	varies by layer
$X^{\{t\}}$	input mini-batch no. t	(n_x, m_t)
$Y^{\{t\}}$	output mini-batch no. t	$(1, m_t)$
m	total num of train examples	scalar
m_t	num of examples in mini-batch t	scalar

What is an Epoch?

- It is a one pull pass over the entire train set.

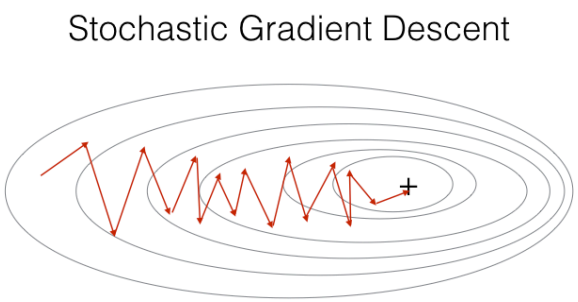
In mini-batch gradient descent there are multiple gradient steps—once per mini-batch.
If there's 5000 mini-batches, there's 5000 updates per epoch.

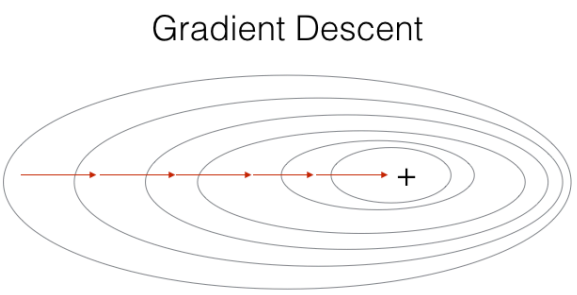
You train for multiple epochs until convergence.

Mini-batch gradient descent procedure:

1. Forward propagation on X^t
    1. $Z^{[1]} = W^{[1]}X^{\{t\}} + b^{[1]}$
    2. $A^{[1]} = g^{[1]}(Z^{[1]})$
    3. $\dots$
    4. $Z^{[1]} = W^{[1]}A^{[L-1]} + b^{[L]}$
    5. $A^{[L]} = g^{[L]}(Z^{[L]})$
2. Compute cost for mini-batch $J^{\{t\}} = \frac{1}{m_t}\sum_{i=1}^{1000}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} + \sum_l ||W^{[l]}||_F^2$
3. Backward propagation by computing gradients w.r.t. $J^{\{t\}}$ using $(X^{\{t\}}, Y^{\{t\}})$
    1. $dZ^{[L]} = A^{[L]} - Y^{\{t\}}$
    2. $dW^{[L]} = \frac{1}{m_t}dZ^{[L]}(A^{[L-1]})^\intercal + \frac{\lambda}{m}W^{[L]}$
    3. $db^{[L]} = \frac{1}{m_t}\sum dZ^{[L]}$
    4. $\dots$
    5. $dA^{[1]} = (W^{[2]})^\intercal \cdot dZ^{[2]}$
    6. $dZ^{[1]} = dA^{[1]} * g^{[1]'}(Z^{[1]})$
    7. $dW^{[1]} = \frac{1}{m_t}dZ^{[1]} \cdot X^\intercal + \frac{\lambda}{m}W^{[1]}$
    8. $db^{[1]} = \frac{1}{m_t}\sum dZ^{[1]}$
4. Update the parameters
    1. $W^{[l]} := W^{[l]} - \alpha \cdot dW^{[l]}$
    2. $b^{[l]} := b^{[l]} - \alpha \cdot db^{[l]}$

Types of gradient descent





Batch, $m_t = m$

- Stable and smooth steps.
- Slow.

Stochastic, $m_t = 1$

- Noisy and zigzag steps.
- May head in a wrong direction.
- Loses speedup from vectorization.

Mini-batch, $m_t \in (1, m)$

- Moderately noisy but stable steps.
- More consistent than SGD.
- Faster than batch GD.
- Fastest learning.

Type	Mini-batch size
Batch gradient descent	$m_t = m$

Type	Mini-batch size
Stochastic gradient descent	$m_t = 1$
Mini-batch gradient descent	$m_t \in (1, m)$

Choosing mini-batch size

If train set is small $(m \leq 2000)$: use batch GD. Otherwise choose a mini-batch size.

Use powers of 2 for typical mini-batch sizes: $64, 128, 256, 512, 1024, \ldots$ due to hardware memory alignment.

Ensure mini-batch fit in CPU/GPU memory.

Implementation

There are two steps in building mini-batches: shuffling and partioning.

1. Shuffling
   - Create a shuffled version of the train set (X, Y).
   - Each column of X an Y corresponds to a single train example.
   - Shuffle synchronously—the i^{th} column X stays paired with the i^{th} column of Y after shuffling.
   - Shuffling ensures that examples are split randomly into different mini-batches.
2. Partitioning
   - Divide the shuffled train set into mini-batches of size m_t.
   - If the num of train examples is not divisible by m_t, the final mini-batch will have fewer examples. This is normal.

```
def random_mini_batches(X, Y, mini_batch_size=64):

    m = X.shape[1] # num of train examples
    mini_batches = []

    # shuffling
    permutation = list(np.random.permutation(m))
    shuffled_X = X[:, permutation]
    shuffled_Y = Y[:, permutation].reshape((1, m))

    # partition
    num_complete_minibatches = math.floor(m / mini_batch_size) # num of mini batches
    for k in range(0, num_complete_minibatches):
        start = k * mini_batch_size
        end = (k+1) * mini_batch_size
        mini_batch_X = shuffled_X[:, start:end]
        mini_batch_Y = shuffled_Y[:, start:end]

        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    # end case when m is not divisisible by m_t
    if m % mini_batch_size != 0:
        rem_start = num_complete_minibatches * mini_batch_size
        rem_end = m
        mini_batch_X = shuffled_X[:, rem_start:rem_end]
        mini_batch_Y = shuffled_Y[:, rem_start:rem_end]
```

```
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    return mini_batches
```

Exponentially weighted averages

Exponentially weighted averages (EWAs) smooths out data by giving more weight to recent observations.

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

- v_t: the exponentially weighted average at t
- β: decay factor, a hyperparameter (usually close to 1)
- θ_t: the current observation at t

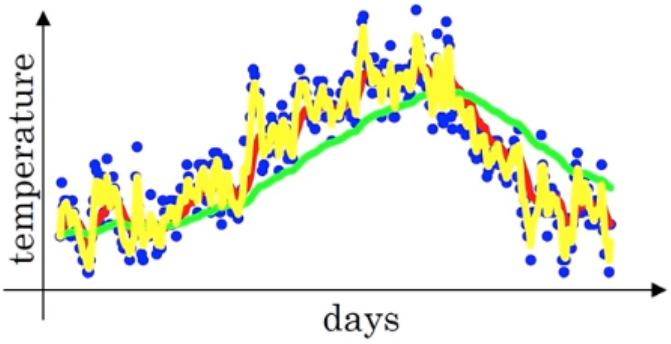
EWAs are used in optimization algorithms like Momentum, Adam, and RMSProp.

Python implemenation:

```
V = 0  # initialization
for t in range(1, T+1):
    V = beta * V + (1 - beta) * theta_t
```

Effective number of days

The effective number of days which the average is computed is $\approx \frac{1}{1-\beta}$



Indicator	β	Effective num. of days	Smoothness	Adaptability
🔴	0.90	~10 days	Smooth average	Moderately fast to adapt
🟢	0.98	~50 days	Very smooth	Slow to adapt
🟡	0.50	~2 days	Noisy and sensitive to outliers	Very responsive

Higher β is smoother but slower to adapt.
Lower β is noisier but faster to adapt.

If $\beta = 1 - \epsilon$, then

$$(1-\epsilon)^{\frac{1}{\epsilon}} \approx \frac{1}{e} \approx 0.368$$

Meaning, it takes about $\frac{1}{\epsilon}$ steps for the weight to decay to around $\frac{1}{e}$ of its original value.

Method	Accuracy	Memory use	Complexity
Moving window average	More accurate	High	High
Exponentially weighted average	Slightly less	Very Low	Very Low

For context, here's the formula of the Moving Window Average:

$$M_t = \frac{1}{k}\sum_{i=0}^{k-1} x_{t-i}$$

- M_t: moving average at t
- k: window size
- x_{t-i} the value at $t-i$

EWA is ideal for DL for its low memory use and complexity where is low.

EWA is biased toward zero when t is small. This is solved using *bias correction*.

Bias correction

Use bias correction to remove the bias:

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

Use case: only when early estimates matter.

Rationale: When initializing using $v_0 = 0$, the early values of v_t underestimates. This is significant when β is close to 1 as it takes longer for the average to catch up to the real values. The denominator $1-\beta^t$ corrects for the missing weight from early time steps. As $t \to \infty$, $B^t \to 0$. So, $\hat{v}_t \approx v_t$.

Remarks:

- As time progresses, the impact of bias correction diminishes. This makes the estimates converge to similar values.

Gradient descent with momentum

Problem with standard gradient descent:

1. When optimizing J with elongated contours, the algorithm may oscillate back and forth across the narrow dimension (e.g., vertical axis).
2. This forces the use of a small learning rate α, since a large α might cause overshooting and end up diverging.

3. Convergence slows due to small α.

What do we want?

- Slow learning in the vertical direction to prevent oscillations.
- Fast learning in the horizontal direction to quickly reach the minimum.

The solution? Use momentum.

- Momentum smooths the optimization path by averaging gradients over time.
- Averaging causes vertical oscillations to cancel out.
- Averaging causes horizontal direction to accumulate.
- It can be applied with batch gradient descent, mini-batch gradient descent or stochastic
- You have to tune a momentum hyperparameter β and a learning rate α.

Initialize $v_{dW} = 0$ and $v_{db} = 0$.

Momentum procedure during iteration t:

1. Compute the gradients dW, db on the current mini-batch.
2. Update velocity
   - $v_{dW} = \beta \cdot v_{dW} + (1 - \beta) \cdot dW$
   - $v_{db} = \beta \cdot v_{db} + (1 - \beta) \cdot db$
3. Update parameters
   - $W := W - \alpha \cdot v_{dW}$
   - $b := b - \alpha \cdot v_{db}$

β is the momentum hyperparameter.
The common value for β is 0.9 which is equivalent to averaging over ~10 iterations.

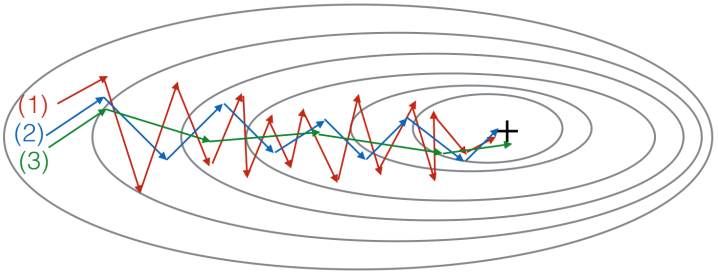
Physics analogy:

- β acts as friction, preventing infinite acceleration
- v_{dW}, v_{db} acts as velocities
- dW, db acts as accelerators

Alternate form:

- $v_{dW} = \beta \cdot v_{dW} + dW$
- $v_{db} = \beta \cdot v_{db} + db$

This requires tuning for α. Not intuitive.



- (1) is gradient descent.
- (2) is gradient descent with momentum (small β).
- (3) is gradient descent with momentum (large β).

RMSprop

RMSprop (*Root Mean Square Propagation*) adapts the learning rate α per parameter based on EWAs of the squared gradients.

RMSprop allows faster convergence by:

- dampening oscillations in steep directions.
- enabling quicker updates in flatter directions.

RMSprop procedure during iteration t:

1. Compute the gradients dW, db on the current mini-batch.
2. Maintain EWAs of squared gradients
   - $S_{dW} = \beta \cdot S_{dW} + (1 - \beta) \cdot dW^2$
   - $S_{db} = \beta \cdot S_{db} + (1 - \beta) \cdot db^2$
   - Squaring of dW^2 and db^2 is element-wise.
3. Parameter update
   - $W := W - \alpha \cdot \frac{dW}{\sqrt{S_{dw}} + \varepsilon}$
   - $b := b - \alpha \cdot \frac{db}{\sqrt{S_{db}} + \varepsilon}$
   - $\epsilon = 10^{-8}$ is a small constant added for numerical stability.

During implementation, use:

- β_2 for RMSprop.
- β for momentum.

Downsides of RMSprop:

- RMSprop is sensitive to the choice of α.
  - If α is too high: updates become excessively large—leading to divergence.
  - If α is too low: the updates become too small—leading to slow convergence. At worst case, getting stuck in local minima.
- RMSprop does not incorporate momentum, leading to suboptimal convergence paths.

Adam

Adam (*Adaptive Moment Estimation*) merges momentum and RMSprop.

Background:

- Momentum uses EWAs of gradients to smooth updates.
- RMSprop uses EWAs of squared gradients to scale updates adaptively.

V_{dW} is the first moment (the mean) which guides the direction of the updates.
S_{dW} is the second moment (the uncentered variance) which adjusts the step size for each parameter update.

Implementation

Initialize all moment vectors at $t = 0$:

- $v_{dW} = 0$
- $v_{db} = 0$
- $S_{dW} = 0$
- $S_{db} = 0$

During iteration t:

1. Compute the gradients dW, db on the current mini-batch.
2. Momentum update (first moment)
   - $v_{dW} = \beta_1 \cdot v_{dW} + (1 - \beta_1) \cdot dW$
   - $v_{db} = \beta_1 \cdot v_{db} + (1 - \beta_1) \cdot db$
3. RMSprop update (second moment)
   - $S_{dW} = \beta_2 \cdot S_{dW} + (1 - \beta_2) \cdot dW^2$
   - $S_{db} = \beta_2 \cdot S_{db} + (1 - \beta_2) \cdot db^2$
4. Bias correction due to the bias to 0 caused by initialization
   - $\hat{v}_{dW} = \dfrac{v_{dW}}{1 - \beta_1^t}$
   - $\hat{v}_{db} = \dfrac{v_{db}}{1 - \beta_1^t}$
   - $\hat{S}_{dW} = \dfrac{S_{dW}}{1 - \beta_2^t}$
   - $\hat{S}_{db} = \dfrac{S_{db}}{1 - \beta_2^t}$
5. Parameter update
   - $W := W - \alpha \dfrac{\hat{v}_{dW}}{\sqrt{\hat{S}_{dW}} + \varepsilon}$
   - $b := b - \alpha \dfrac{\hat{v}_{db}}{\sqrt{\hat{S}_{db}} + \varepsilon}$

Hyperparameter	Symbol	Typical value
Learning rate	α	needs to be tuned
Momentum decay rate (first moment)	β_1	0.9

Hyperparameter	Symbol	Typical value
RMSprop decay rate (second moment)	β_2	0.999
Numerical stability constant	ε	10^{-8}

Both 0.9 and 0.999 are values proposed by the authors (Kingma et. al.).

Learning rate decay

Learning rate decay can speed up a learning algorithm by slowly reducing learning rate over time.

Epoch-based learning rate decay formula:

$$\alpha = \frac{1}{1 + \text{decay rate} \cdot \text{epoch num}} \alpha_0$$

- a_0 and decay rate must be tuned.

Learning rate decay is lower in priority than finding a good fixed α.

Exponential decay

$$\alpha = \alpha_0 \times (\text{decay factor})^{\text{epoch num}}$$

- $\text{decay factor} < 1$
- α decreases exponentially fast.

Inverse square root decay

$$\alpha = \frac{k \cdot a_0}{\sqrt{\text{epoch num}}}$$

- k is a constant.

Mini-batch step decay

$$\alpha = \frac{k \cdot a_0}{\sqrt{t}}$$

- t is the mini-batch number.

Discrete staircase decay

1. Maintain α for several steps.
2. Reduce by a factor (e.g., halve every x number of steps)

Manual decay

When training one or few models at a time:

- Watch the model during its long training.
- Manually reduce α based on model performance
- Adjust hour-by-hour or day-by-day.

Problem of local optima

Early practitioners worry about optimization algorithms getting stuck in a bad local optima.

A critical point is a point where the gradient is zero, $\nabla f\left(a,b\right)=\vec{0}$.

A saddle point is a critical point where it is neither a local maximum nor a local minimum.

In high-dimensional spaces most critical points are saddle points. Therefore, getting a saddle point is more likely than getting a bad local minima.

A plateau are large flat regions where the gradient is zero or near-zero. Plateaus slows down learning.

Optimization algorithms like Momentum, RMSProp, and Adam help escape plateaus faster.