Week 2: Optimization Algorithms

Learning Objectives

- Apply optimization methods such as (Stochastic) Gradient Descent, Momentum, RMSProp and Adam.
- Use random minibatches to accelerate convergence and improve optimization.
- Describe the benefits of learning rate decay and apply it to your optimization.

Optimization Algorithms

Mini-batch gradient descent

Vectorization allows you to process all M examples relatively quickly if M is very large, but it can still be slow.

For example, `m = 5,000,000` (or `m = 50,000,000` or even bigger), we have to process the entire training sets of five million training samples before we take one little step of gradient descent.

We can use the `mini-batch` method to let gradient descent start to make some progress before we finish processing the entire, giant training set of 5 million examples by splitting up the training set into smaller, little baby training sets called mini-batches. In this case, we have 5000 mini-batches with 1000 examples each.

Notations:

- `(i)`: the i^{th}training sample
- `[l]`: the l^{th} layer of the neural network
- `{t}`: the t^{th} mini batch

In every step of the iteration loop, we need to loop for `num_batches` and do forward and backward computation for each batch.

for t in range (num_batches):

1. **Forward propagation**

   Vectorize this:

   $$Z^{[1]} = W^{[1]}X^{\{t\}} + b^{[1]}$$
   $$A^{[1]} = g^{[1]}(Z^{[1]})$$
   ...
   $$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$
   $$A^{[l]} = g^{[1]}(Z^{[l]})$$

2. **Compute cost function**

   Cost $J^{\{t\}} = \frac{1}{1000}\sum_{i=1}^{l} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2.1000}\sum_{l} \parallel w^{[l]} \parallel_F^2$. (*here l denotes number of samples in one mini-batch*)

3. **Backward propagation**

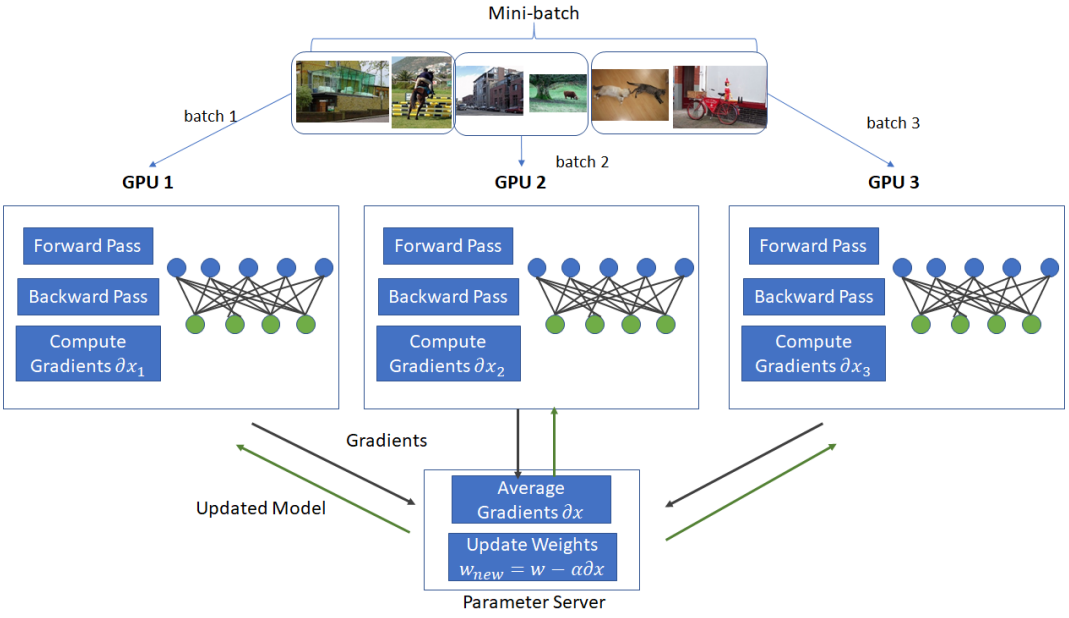
compute the gradient with respect to $J^{\{t\}}$ (using $(x^{\{t\}}, y^{\{t\}})$)

4. **Update parameters** (using parameters, and grads from backprop)
   $w^{[l]} := w^{[l]} - \alpha dw^{[l]}$,
   $b^{[l]} := b^{[l]} - \alpha db^{[l]}$

With mini-batch gradient descent, a single pass through the training set is one epoch, which is the above 5 million examples, means 5000 gradient descent steps.

▼ How Mini-Batch work

1. **Divide the data into mini-batches**: The first step is to divide the entire training dataset into smaller, equally sized batches. The number of mini-batches (n) is determined by the total number of data points and the chosen batch size.



2. **Iterate over mini-batches**: The algorithm then iterates through each mini-batch in sequence. For each mini-batch:
   - The gradient of the loss function is calculated with respect to the current model parameters.
   - The model parameters are updated based on the calculated gradient and the chosen learning rate.
3. **Repeat:** Once the algorithm has processed all mini-batches in the current epoch, it starts over again with the first mini-batch. This process continues for a pre-defined number of epochs.

Benefits of Mini-batch Gradient Descent:

- **Faster training**: The use of mini-batches allows for faster training compared to batch gradient descent, which updates the model parameters based on the entire dataset at once. This is because mini-batch gradient descent can utilize vectorized operations and parallelization, which can significantly improve computational efficiency.

- **Reduced memory requirements**: Mini-batches require less memory to store compared to the entire dataset, making it suitable for larger datasets or deep learning models with a large number of parameters.
- **Improved generalization**: Mini-batch gradient descent can sometimes lead to better generalization performance compared to batch gradient descent. This is because mini-batches introduce a form of stochasticity in the optimization process, which can help the model avoid overfitting to the training data.

Disadvantage of Mini-batch Gradient Descent:

- **Noisier gradients**: Mini-batch gradients are inherently noisier than gradients calculated over the entire dataset. This can lead to more erratic updates and potentially slower convergence compared to batch gradient descent.
- **Hyperparameter tuning**: Choosing an appropriate batch size is a critical hyperparameter that can significantly impact the performance of mini-batch gradient descent. Tuning the batch size requires additional experimentation and validation.

Understanding mini-batch gradient descent

batch size	method	description	guidelines
`m`	batch gradient descent (only 1 minibatch == whole training set)	- cost function decreases on every iteration; - but too long per iteration	use it for a small training set of size <2000
`1`	stochastic gradient descent (every example is it own minibatch → total of `m` minibatches)	- cost function oscillates, can be extremely noisy; - wander around minimum; - lose speedup from vectorization, inefficient.	use a smaller learning rate when it oscillates too much
between `1` and `m`	mini-batch gradient descent	- somewhere in between, vectorization advantage, faster; - not guaranteed to always head toward the minimum but more consistently in that direction than stochastic gradient descent; - not always exactly converge, may oscillates in a very small region, reducing the learning rate slowly may also help.	- mini-batch size is a hyperparameter; - batch size better in [64, 128, 356, 512], a power of 2; - make sure that mini-batch fits in CPU/GPU memory

Exponentially Weighted Averages

Moving averages are favored statistical tools of active traders to measure momentum. There are three MA methods:

MA methods	calculations
Simple Moving Average (SMA)	calculated from the average closing prices for a specified period
Weighted Moving Average (WMA)	calculated by multiplying the given price by its associated weighting (assign a heavier weighting to more current data points) and totaling the values
Exponential Moving Average (EMA)	also weighted toward the most recent prices, but the rate of decrease is exponential

EMA	SMA	WMA
Exponential Moving Average	**Simple Moving Average**	**Weighted Moving Average**
Greater weight to the most recent data points	Gives equal weight to each price point within the period	Weightage decreases linearly for older prices
Best for short-term trend observation	Best for long-term trend observation	Best for short-term trend observation

What is a Moving Average (MA)

A moving average is a statistical tool used to analyze datapoints by creating a series of averages from different subsets of the complete dataset.

SMA, EMA, and WMA are types of MAs.

The MA smooths out price data on a chart to create a single flowing line, making it **easier to identify the direction of the trend.**

SMA vs EMA compared

`SMA` and `EMA` are both types of moving averages but differ in their calculation.

- The `SMA` assigns equal weight to all values.
- The `EMA` gives more weight to recent data, offering a quicker response to price changes.

WMA vs EMA compared

Both `WMA` and `EMA` give more importance to recent data but differ in their approach.

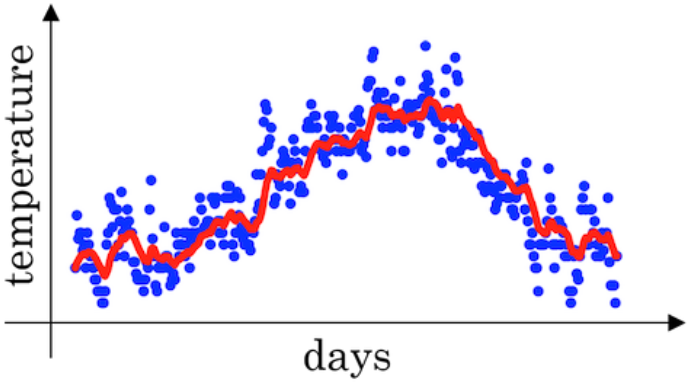
- The `WMA` uses a linear weighting method.
- The `EMA` uses an exponential approach, making the EMA quicker to react to recent price movements.

Example:

For a list of daily temperatures throughout the year:

$$
\begin{aligned}
&day_1 : \theta_1 = 40^oF \\
&day_2 : \theta_2 = 49^oF \\
&day_3 : \theta_3 = 45^oF \\
&... \\
&day_180 : \theta_{180} = 60^oF \\
&day_181 : \theta_{181} = 56^oF \\
&...
\end{aligned}
$$

Visualizing the above data:



Daily temperature in London throughout the year

If you want to compute the trend of the temperature of the year, you can do as follow:

$$
\begin{aligned}
&V_0 = 0 \\
&V_1 = 0.9 * V_0 + 0.1 * \theta_1 \\
&V_2 = 0.9 * V_1 + 0.1 * \theta_2 \\
&... \\
&V_t = 0.9 * V_{t-1} + 0.1 * \theta_t
\end{aligned}
$$

or the above formula is equivalent to:

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

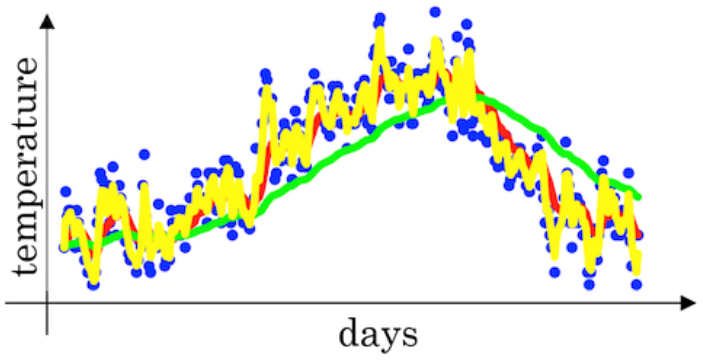
You can think of V_t as the approximately average over $\approx \frac{1}{1-\beta}$ days temperature.

For the above example, with $\beta = 0.9$, you can think about it as average temperature over $\frac{1}{1-\beta} = \frac{1}{1-0.9} \approx 10$ days, and that was the red line, which is called the `Exponentially Weighted Moving Average` line or EMA.

> → The higher value for β, the smoother the trend line is, as it is the average value over larger time period.
>
> → the curve will be shifted further to the right because you're now averaging over a much larger window.

If we want to compute the trends, by averaging over a larger window ($\frac{1}{1-\beta}$), the above exponentially weighted average formula adapts more slowly when the temperature changes. So, there's just a bit more latency. (See the red curve above)

- When $\beta = 0.98$ then it's giving a lot of weight to the previous value and a much smaller weight just 0.02, to whatever you're seeing right now (as in the formula: $V_t = 0.98 * V_{t-1} + 0.02 * \theta_t$) (see the green curve below).
- When $\beta = 0.5$, which something like averaging over just two days temperature (period = $\frac{1}{1-0.5}$ days). And by averaging only over two days temperatures, as if averaging over much shorter window, it's much more noisy, much more susceptible to outliers. But this adapts much more quickly to want the temperature changes (see the yellow curve below).



Understanding exponentially weighted averages

This topic is basically related to gradient descent optimizations.

Let's take a look at the formula for WMA as described earlier:

$$V_t = \beta V_{t-1} + (1 - \beta)\theta_t$$
$$V_{t-1} = \beta V_{t-2} + (1 - \beta)\theta_{t-1}$$
$$V_{t-2} = \beta V_{t-3} + (1 - \beta)\theta_{t-2}$$
$$...$$

From the above sequence, we can observe that:

$$
\begin{aligned}
V_t &= \beta V_{t-1} + (1-\beta)\theta_t \\
&= \beta[\beta V_{t-2} + (1-\beta)\theta_{t-1}] + (1-\beta)\theta_t \\
&= \beta^2 V_{t-2} + \beta(1-\beta)\theta_{t-1} + (1-\beta)\theta_t \\
&= \beta^2 V_{t-2} + (1-\beta)(\beta\theta_{t-1} + \theta_t) \\
&= \beta^2[\beta V_{t-3} + (1-\beta)\theta_{t-2}] + (1-\beta)(\beta\theta_{t-1} + \theta_t) \\
&= \beta^3 V_{t-3} + (1-\beta)\beta^2\theta_{t-2} + (1-\beta)(\beta\theta_{t-1} + \theta_t) \\
&= \beta^3 V_{t-3} + (1-\beta)(\beta^2\theta_{t-2} + \beta\theta_{t-1} + \theta_t) \\
&= \beta^3[\beta V_{t-4} + (1-\beta)\theta_{t-3}] + (1-\beta)(\beta^2\theta_{t-2} + \beta\theta_{t-1} + \theta_t) \\
&= \beta^4 V_{t-4} + (1-\beta)\beta^3\theta_{t-3} + (1-\beta)(\beta^2\theta_{t-2} + \beta\theta_{t-1} + \theta_t) \\
&= \beta^4 V_{t-4} + (1-\beta)(\beta^3\theta_{t-3} + \beta^2\theta_{t-2} + \beta^1\theta_{t-1} + \beta^0\theta_{t-0}) \\
&\dots \\
&= \beta^n V_{t-n} + (1-\beta)\sum_{i=0}^{n-1}\beta^i\theta_{t-i}
\end{aligned}
$$

The exponentially weighted average adds a fraction β of the current value to a leaky running sum of past values. Effectively, the contribution from the $t-n^{th}$ value is scaled by $\beta^n(1-\beta)$.

For example, here are the contributions to the current value after 5 iterations (iteration 5 is the current iteration)

iteration	contribution
1	$\beta^4(1-\beta)$
2	$\beta^3(1-\beta)$
3	$\beta^2(1-\beta)$
4	$\beta^1(1-\beta)$
5	$(1-\beta)$

Since $\beta < 1$, the contribution decreases exponentially with the passage of time. Effectively, this act as a smoother for a function.

e-folding:

Andrew Ng also mentioned an interesting concept related to e-folding. He said:

- If $\beta = 0.9$ it would take about 10 day for `v` to decay about `1/3` ($\frac{1}{e} \approx \frac{1}{3}$) of the peak;
- If $\beta = 0.98$ it would be 50 days.

Here 10 or 50 days is called one lifetime (1 e-folding). Generally, for an exponential decay quantity, after one life time ($\frac{1}{(1-\beta)}$ iterations), $\frac{1}{e} \approx 37$ is remained and after two lifetime, $\frac{1}{e^2} \approx 14$ is left.

For more information, check the e-folding definition.

Implementing EMA

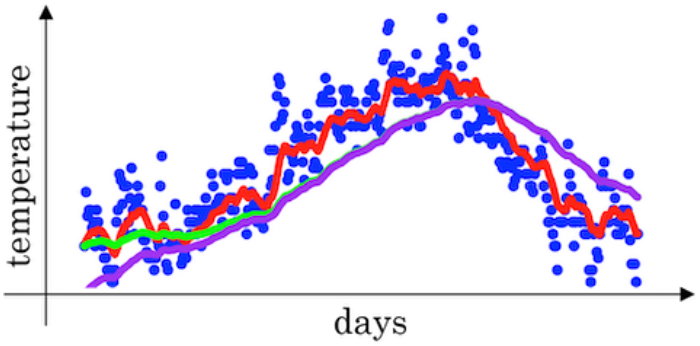
$$
\begin{aligned}
init &: V_\theta \coloneqq 0 \\
day1 &: V_\theta \coloneqq \beta V + (1-\beta)\theta_1 \\
day2 &: V_\theta \coloneqq \beta V + (1-\beta)\theta_2 \\
&\dots
\end{aligned}
$$

or implementation when coding:

$V_\theta = 0$
Repeat for each day {
    Get next θ_t
    $V_\theta \coloneqq \beta V_\theta + (1-\beta)\theta_t$
}

Bias correction in exponentially weighted averages

There's one technical detail called biased correction that can make your computation of these average more accurately. In the temperature example above, when we set $\beta = 0.98$, we won't actually get the green curve; instead, we get the purple curve (see the graph below).



Because when we're implementing the exponentially weighted moving average, we initialize it with `V0 = 0`, subsequently we have the following result in the beginning of the iteration:

- `V1 = 0.98*V0 + 0.02*θ1 = 0.02 * θ1`
- `V2 = 0.98*V1 + 0.02*θ2 = 0.0196 * θ1 + 0.02 * θ2`

As a result, `V1` and `V2` calculated by this are not very good estimates of the first two temperature. So we need some modification to make it more accurate, especially during the initial phase of our estimate to avoid an initial bias.

> → This can be corrected by scaling V_t with `1/(1-β^t)` where `t` is the iteration number.

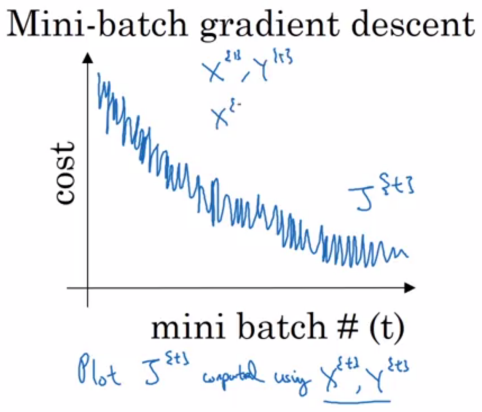
Original	Correction
$V_1 = 0.02 * \theta_1$	$V_1 = (0.02 * \theta_1) * \frac{1}{1-\beta^t} = \frac{0.02*\theta_1}{1-0.98^1} = \theta_1$

Original	Correction
$V_2 = 0.0196 * \theta_1 + 0.02 * \theta_2$	$V_2 = (0.0196 * \theta_1 + 0.02 * \theta_2) * \frac{1}{1-\beta^t} = \frac{0.0196 * \theta_1}{1-0.98^2} = \frac{0.0196 * \theta_1 + 0.02 * \theta_2}{0.0396}$

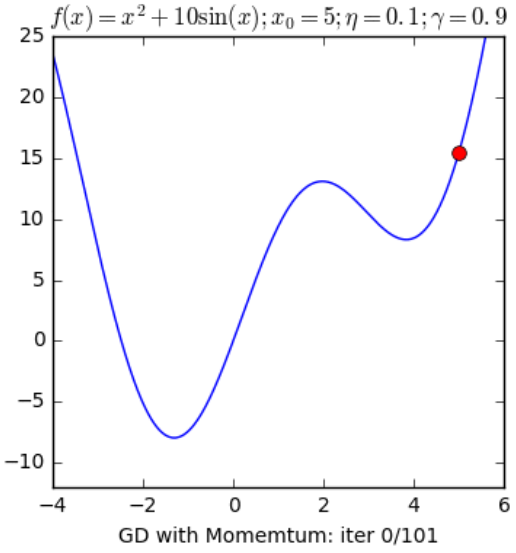
→ When t is large, then β^t will approach 0, this make the bias correction lead to no difference.

→ That's why when t is large, the purple line (EWA with V0 = 0) and the green line (theoretical EWA) above are overlaps. But in the initial phase, when you're still "warming up" your estimates, bias correction can help you obtain a better estimate of the temperature.

Gradient descent with momentum



Because mini-batch gradient descent makes a parameter update after seeing just a subset of examples, the direction of the update has some variance, and so the path taken by mini-batch gradient descent will "oscillate" toward convergence. Using `momentum` helps the GD algorithm converge faster and avoid getting stuck in local minima, therefore reduce these oscillations. We call this algorithm `Gradient Descent with Momentum` or `GDM`.



How GDM works:

(read more here)

In GD, the algorithm updates the parameters (weights and biases) in the direction of the negative gradient (read this). This means that the algorithm moves towards the point where the gradient is zero, which corresponds to a local minimum or maximum of the loss function. However, **GD can get stuck in local minima**, especially in complex optimization problems with multiple potential minima.

`GDM` introduces `momentum` by maintaining a `velocity terms` that **tracks** the **direction** and **magnitude** of recent gradient updates.

This velocity term is updated at each iteration, incorporating a factor of the previous velocity along with the current gradient. The updated velocity is then used to update the parameters.

Benefits of GDM

1. **Faster Convergence**: Momentum helps the algorithm converge faster to the optimal solution by reducing oscillations and guiding it towards more efficient paths in the parameter space.
2. **Improved Stability**: Momentum stabilizes the learning process by preventing the algorithm from getting stuck in local minima or saddle points. This is particularly beneficial for complex optimization problems with multiple potential minima.
3. **Reduced Oscillations**: Momentum reduces the zig-zagging behavior that can occur with GD, making the learning process smoother and more efficient.

EWA (Exponential Weighted Moving Average) in GDM

`EWA` is a technique used to **calculate the momentum term** in `GDM`. It introduces a `decay factor` that exponentially weights the contributions of past gradient.

→ This means that the algorithm places more emphasis on recent gradient directions while gradually reducing the influence of older ones.

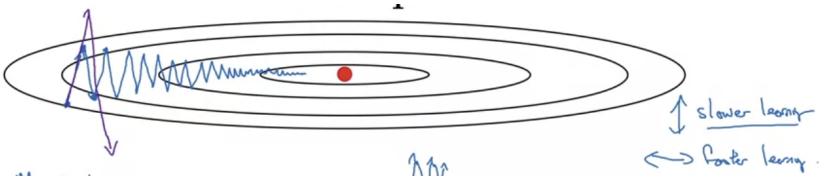
Benefits of EWA:

1. **Adaptive Momentum**: EWA allows the momentum to adapt to the changing landscape of the loss function, adjusting its strength based on the recent gradient directions.
2. **Reduced Sensitivity to Noise**: EWA reduces the algorithm's sensitivity to noise in the gradient, making it more robust and less prone to oscillations.

> Gradient Descent with momentum, which computes an EWA of gradients to update weights almost always works faster than the standard Gradient Descent algorithm.

> Algorithm has two hyper-parameters of `alpha`, the learning rate, and `beta` which controls your EWA. Common value for `beta` is `0.9`.

> Don't bother with bias correction



Implementation details:

💡 On iteration t:

- Compute dW, db on the current mini-batch
- Compute the EWA of db and dW, which helps smooth out the steps of gradient descent.

$$v_{dw} = \beta v_{dw} + (1-\beta)dW$$
$$v_{db} = \beta v_{db} + (1-\beta)db$$
$$W := W - \alpha v_{dw}$$
$$b := b - \alpha v_{db}$$

> *If you think of a bowl, and you take a ball and roll it down the bowl's hill, the derivatives db and dW act as the "velocity" to help the ball roll faster and faster.*

> *β here will take the role of "friction" to prevent the ball from speeding up without limit.*

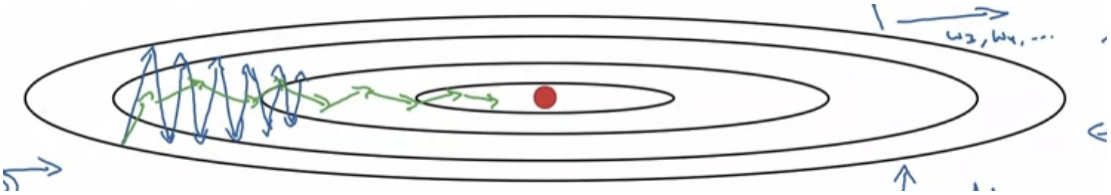
> *→ velocity variables v_{dw} and v_{db} both carry the informations about **slope** (derivative), informations about **momentum**, and also **act as a smoother** for GD.*

Implementation tips:

- If $\beta = 0$ then this just becomes standard gradient descent without momentum.
- The larger the momentum β is, the smoother the update because the more we take the past gradients into account. But if β is too big, it could also smooth out the updates too much.
- Common values for β range from `0.8` to `0.999`. If you don't feel inclined to tune this, $\beta = 0.9$ is often a reasonable default.
- It can be applied with batch gradient descent, mini-batch gradient descent or stochastic gradient descent.

RMSprop

`RMSprop` (Root Mean Square), similar to momentum, has the effects of damping out the oscillations in gradient descent and mini-batch gradient descent and allowing you to maybe use a larger learning rate α.



Your intuition is that in the dimensions where you're getting the oscillations, you end up computing a larger sum. A weighted average for these squares and derivatives, and so you end up dumping out the directions in which there are these oscillations.

The algorithm computes the EWA of the squared gradients and updates weights by the square root of the EWA.

Implementation detail:

💡 On iteration t:

- Compute dW, db on the current mini-batch

- Compute the EWA of db and dW, which helps smooth out the steps of gradient descent.

$$s_{dw} = \beta_2 s_{dw} + (1 - \beta_2) dW^2 \quad \text{\# the square operator is element-wise}$$
$$s_{db} = \beta_2 s_{db} + (1 - \beta_2) db^2 \quad \text{\# the square operator is element-wise}$$
$$W := W - \alpha \frac{s_{dw}}{\sqrt{s_{dw}} + \epsilon}$$
$$b := b - \alpha \frac{s_{db}}{\sqrt{s_{db}} + \epsilon}$$

note that the RMSprop's β_2 is different from the β from GDM.

```
for iteration t:
    # compute dW, db on mini-batch

    s_dW = (beta * s_dW) + (1 - beta) * dW^2
    S_db = (beta * S_db) + (1 - beta) * db^2
    W = W - alpha * dW / sqrt(S_dW + ε)   # ε: small number(10^-8) to avoid dividing by zero
    b = b - alpha * db / sqrt(S_db + ε)
```

Adam optimization algorithm

- `Adam` (Adaptive Moment Estimation) optimization algorithm is basically putting `momentum` and `RMSprop` together and combines the effect of `Gradient Descent with Momentum` together with `Gradient Descent with RMSprop`.
- This is a commonly used learning algorithm that is proven to be very effective for many different neural networks of a very wide variety of architectures.

- In the typical implementation of Adam, `bias correction` is on.

Implementation:

$V_{dw} = 0, S_{dw} = 0, V_{db} = 0, S_{db} = 0$

On iteration t (epoch t):

Compute dW, db using current mini-batch

$$V_{dw} = \beta_1 V_{dw} + (1 - \beta_1) dW, \; V_{db} = \beta_1 V_{db} + (1 - \beta_1) db \quad \leftarrow \text{“momentum” } \beta_1$$
$$S_{dw} = \beta_2 S_{dw} + (1 - \beta_2) dW, \; S_{db} = \beta_2 S_{db} + (1 - \beta_2) db \quad \leftarrow \text{“RMSprop” } \beta_2$$

$$V_{dw}^{corrected} = \frac{V_{dw}}{(1-\beta_1^t)}), \quad V_{db}^{corrected} = \frac{V_{db}}{(1-\beta_1^t)}$$
$$S_{dw}^{corrected} = \frac{S_{dw}}{(1-\beta_1^t)}), \quad S_{db}^{corrected} = \frac{S_{db}}{(1-\beta_1^t)}$$

$$w \coloneqq w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \epsilon}, \qquad b \coloneqq b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \epsilon}$$

```
V_dW = 0
V_db = 0
S_dW = 0
S_db = 0

for iteration t:
  # compute dW, db using mini-batch

  # momentum
  V_dW = (beta1 * V_dW) + (1 - beta1) * dW
  V_db = (beta1 * V_db) + (1 - beta1) * db

  # RMSprop
  S_dW = (beta2 * S_dW) + (1 - beta2) * dW^2
  S_db = (beta2 * S_db) + (1 - beta2) * db^2

  # bias correction
  V_dW_c = V_dW / (1 - beta1^t)
  V_db_c = V_db / (1 - beta1^t)
  S_dW_c = S_dW / (1 - beta2^t)
  S_db_c = S_db / (1 - beta2^t)

  W = W - alpha * V_dW_c / (sqrt(S_dW_c) + ε)
  b = b - alpha * V_db_c / (sqrt(S_db_c) + ε)
```

Implementation tips:

1. It calculate an EWA (Exponentially Weighted Average) of past gradients (momentum), and stores it in variables `V_dW`, `V_db` (before bias correction) and `V_dW_c`, `V_db_c` (with bias correction).
2. It calculates an EWA of the squares of the past gradients (RMSprop), and stores it in variables `S_dW`, `S_db` (before bias correction) and `S_dW_c`, `S_db_c` (with bias correction).
3. It updates parameters in a direction based on combining information from “1” and “2”.

hyperparameter	guideline
`learning rate` α	tune
β_1 (parameter of the momentum, for dW and db)	`0.9`
β_2 (parameter of the RMSprop, for dW^2 as well as db^2)	`0.999`
ϵ (avoid dividing by zero)	`10^-8`

Learning rate decay

The learning algorithm might just end up wandering around, and never really converge, because you're using some fixed value for α.

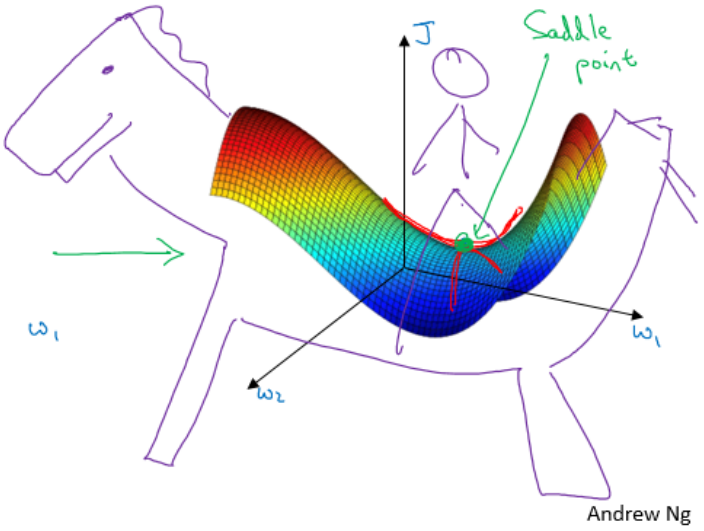
`Learning rate decay` methods can help by making learning rate smaller when optimum is near. There are several decay methods:

Decay factor	description
$0.95^{epoch\_num} * \alpha$	exponential decay
$\frac{k}{\sqrt{epoch\_num}}$ or $\frac{k}{\sqrt{t}}$	polynomial decay
discrete staircase	piecewise constant
manual decay	—

The problem of local optima

- First, you're actually pretty unlikely to get stuck in bad local optima, but much more likely to run into a **saddle point**, so long as you're training a reasonably large NN, save a lot of parameters, and the cost function J is defined over a **relatively high dimensional space**.
- Second, that **plateaus** are a problem and you can actually make learning pretty slow. And this is where algorithms like **momentum** or **RMSprop** or **Adam** an really help your learning algorithm.

This is what a saddle point look like:



Quick notes for optimization algorithms

Recall that in Course 1 we have already known that there are several steps in the Neural Network implementation:

1. Initialize parameters / Define Hyperparameters
2. Loop for num_iterations:
    a. Forward Propagation
    b. Compute Cost Function
    c. Backward Propagation
    d. Update parameters (using parameters and grads from backprop)
3. Use trained parameters to predict labels.

When we create `momentum`, `RMSprop` or `Adam` optimization methods, what we do is to implement algorithms in the **update parameters** step. A good practice is to wrap them up as options so we can compare them during our alchemy training:

```
if optimizer == "gd":
  parameters = "update_parameters_with_gd(parameters, grads, learning_rate)
elif optimizer == "momentum":
    parameters, v = update_parameters_with_momentum(parameters, grads, v, beta, learning_rate)
elif optimizer == "adam":
    t = t + 1 # Adam counter
    parameters, v, s = update_parameters_with_adam(parameters, grads, v, s, t, learning_rate, beta1, beta2, epsilon)
```

Subpages

e-folding

Gradient Descent with Momentum