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.

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)	- 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, vectoriztion 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 mey 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:

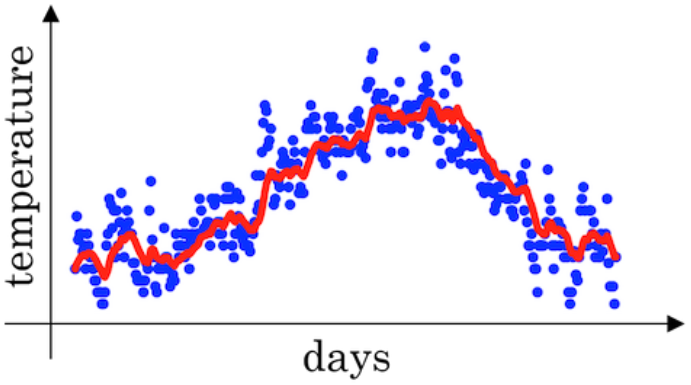
MA methods	calculations
Simple Moving Average (SMA)	calculated from the average closing prices for a specified period
Weighted Moving Average (WMA)	calculated by multiplying the given price by its associated weighting (assign a heavier weighting to more current data points) and totaling the values
Exponential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

For a list of daily temparatures:

$$
\begin{aligned}
&\theta_1 = 40^o F \\
&\theta_2 = 49^o F \\
&\theta_3 = 45^o F \\
&\dots \\
&\theta_{180} = 60^o F
\end{aligned}
$$

$\theta_{181} = 56^o F$

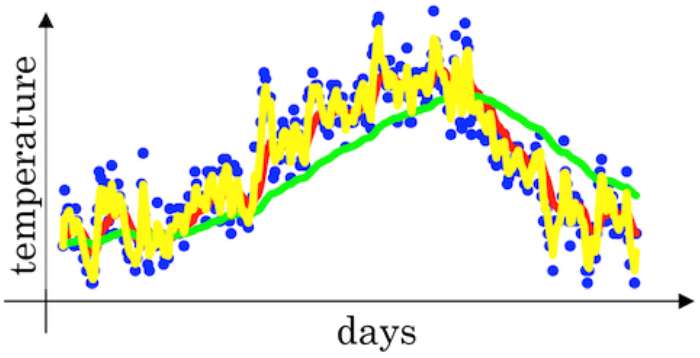
This data looks a little bit noisy (blue dots):



$V_0 = 0$
$V_1 = 0.9 * V_0 + 0.1 * \theta_1$
$V2 = 0.9 * V_1 + 0.1 * \theta_2$
...
$V_t = 0.9 * V_{t-1} + 0.1 * \theta_t$

If we want to compute the trends, by averaging over a larger window, 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 (see the green curve below).
- When $\beta = 0.5$, which something like averaging over just two days temperature. 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 waht the temparature changes (see the yellow curve below).



Understanding exponentially weighted averages

This topic is basically related to gradient descent optimizations.

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

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:

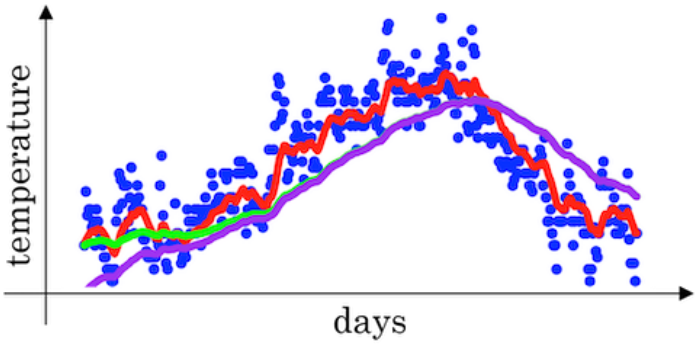
- If $\beta = 0.9$ it would take about 10 day for `v` to decay about `1/3` ($\frac{1}{e} \approx \frac{1}{3}$) of the peak;
- If $\beta = 0.98$ it would be 50 days.

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.

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 temparature example aboves, when we set $\beta = 0.98$, we won't actually get the green curve; Insead, 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 inital bias. This can be corrected by scaling with `1/(1-β^t)` where `t` is the iteration number.

Original	Correction
$V_1 = 0.02\theta_1$	$\frac{0.02\theta_1}{1-0.98} = \theta_1$
$V_2 = 0.0196\theta_1 + 0.02\theta_2$	$\frac{0.0196\theta_1}{1-0.98^2} = \frac{0.0196\theta_1+0.02\theta_2}{0.0396}$

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 can reduce these 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 hyperparameters of `alpha`, the learning rate, and `beta` which controls your EWA. Common value for `beta` is `0.9`.
- Don't bother with bias correction

On iteration t:

Compute dW, db on the current minibatch

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

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

$$W = W - \alpha v_{dw},$$
$$b = b - \alpha v_{db}$$

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.

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

```
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.

```
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, 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 exponentially weighted average of the squares of the past gradients, 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`	tune
`beta1` (parameter of the momentum, for `dW`)	`0.9`
`beta2` (parameter of the RMSprop, for `dW^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 alpha.

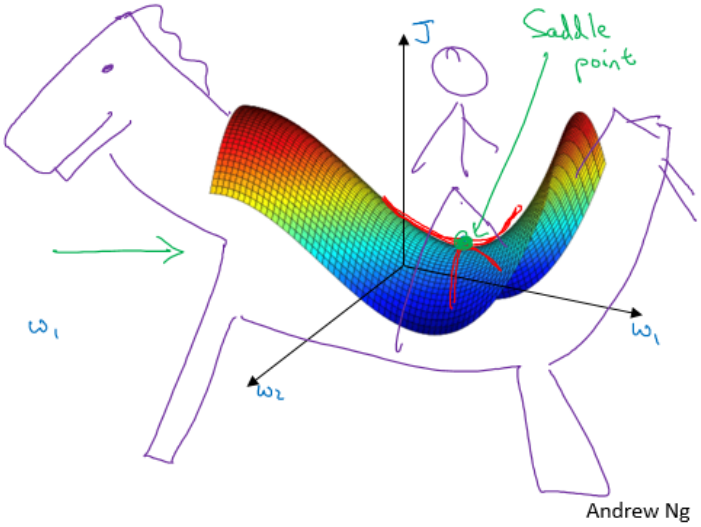
Learning rate decay methods can help by making learning rate smaller when optimum is near. There are several decay methods:

Decay factor	description
0.95^{epoch_num}	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