Improving Deep Neural Networks:
Hyperparameter tuning, Regularization and Optimization

Week-2

This week we learn about optimization algorithms that will enable us to train Neural Networks much faster. We already knew that Applied Machine Learning is a highly iterative process, so if the data size is huge, it would be difficult to iterate the process multiple times which may hurt our efficiency.

$$X_{(n_x,m)} = [X^{(1)}, X^{(2)}, X^{(3)} \dots\dots\dots X^{(1000)}, X^{(1001)} \dots\dots\dots X^{(2000)} \dots\dots\dots\dots\dots\dots\dots\dots X^{(m)}]$$

$$Y_{(1,m)} = [Y^{(1)}, Y^{(2)}, Y^{(3)} \dots\dots\dots Y^{(1000)}, Y^{(1001)} \dots\dots\dots Y^{(2000)} \dots\dots\dots\dots\dots\dots\dots\dots Y^{(m)}]$$

If the dataset is huge we make them into mini-batches.

$$X = [\underbrace{X^{(1)}, X^{(2)}, X^{(3)} \dots\dots\dots X^{(1000)}}_{X^{\{1\}}}, \underbrace{X^{(1001)} \dots\dots\dots X^{(2000)}}_{X^{\{2\}}} \dots\dots\dots\dots\dots \underbrace{\dots\dots\dots\dots\dots\dots\dots\dots X^{(m)}}_{X^{\{t\}}}]$$

$$Y = [\underbrace{Y^{(1)}, Y^{(2)}, Y^{(3)} \dots\dots\dots Y^{(1000)}}_{Y^{\{1\}}}, \underbrace{Y^{(1001)} \dots\dots\dots Y^{(2000)}}_{Y^{\{2\}}} \dots\dots\dots\dots\dots \underbrace{\dots\dots\dots\dots\dots\dots\dots\dots Y^{(m)}}_{Y^{\{t\}}}]$$

So, we run forward propagation on each of the mini batches.
Then, we use Back-Prop to update the Parameters

An iteration of this process is called “1 epoch”-> Single Pass through the training set.

If mini-batch size = m -> Batch Gradient Descent.
If mini-batch size = 1 -> Stochastic Gradient Descent.
(Every example is its own mini-batch)

In Batch Gradient descent, there is less noise and we relatively take large steps.
But it takes too long for each iteration.

Stochastic Gradient descent (SGD), can be extremely noisy as one example can change your path. The good thing is we make progress after a single training example but there is too much noise which can be reduced by a smaller learning rate but the huge disadvantage is that, we lose all the speedup from vectorisation. Instead of taking the advantage from vectorisation, we are simply using 1 example which is a huge waste of resources.

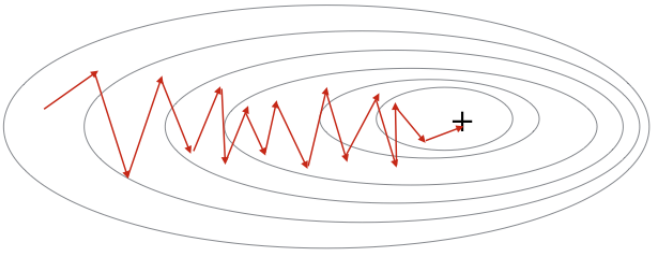
Mini-Batch Gradient Descent is somewhere in between 1 & m.
This gives the fastest learning.

- We get the fastest vectorisation.
- Make progress without processing the entire training set.

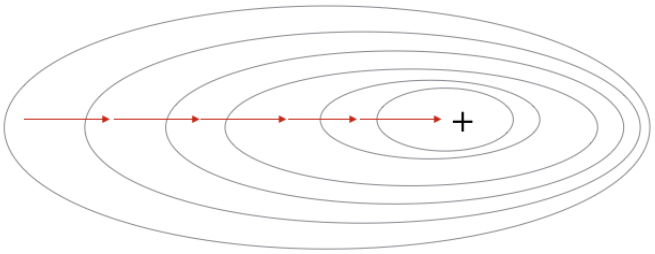
Mini-batch sizes:
Small Training set-> Batch Gradient Descent m≤2000)
Typical Mini Batch Sizes: 64,128,256,512,1024
$2^6,\ 2^7,\ 2^8,\ 2^9,\ 2^{10}$

In practice, the size of a mini-batch is another hyperparameter.

Stochastic Gradient Descent



Gradient Descent



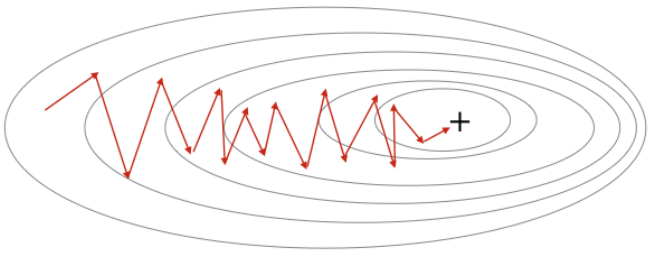
"+" denotes a minimum of the cost. SGD leads to many oscillations to reach convergence. But each step is a lot faster to compute for SGD than for GD, as it uses only one training example (vs. the whole batch for GD).

Note: Implementing SGD requires 3 for-loops in total:

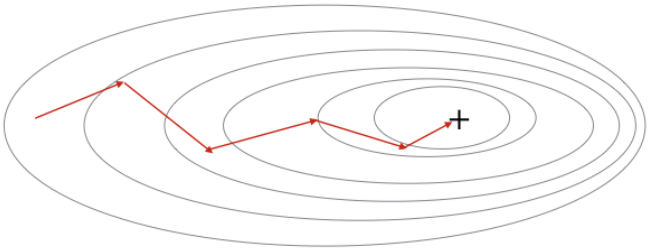
1. Over the number of iterations
2. Over the m training examples
3. Over the layers (to update all parameters, from $(W^{[1]}, b^{[1]})$ to $(W^{[l]}, b^{[l]})$.

In practice, you'll often get faster results if you do not use neither the whole training set, nor only one training example, to perform each update. Mini-batch gradient descent uses an intermediate number of examples for each step. With mini-batch gradient descent, you loop over the mini-batches instead of looping over individual training examples.

Stochastic Gradient Descent



Mini-Batch Gradient Descent



"+" denotes a minimum of the cost. Using mini-batches in your optimization algorithm often leads to faster optimization.

- The difference between gradient descent, mini-batch gradient descent and stochastic gradient descent is the number of examples you use to perform one update step.
- You have to tune a learning rate hyperparameter α.
- With a well-turned mini-batch size, usually it outperforms either gradient descent or stochastic gradient descent (particularly when the training set is large).

Then, we saw how exponentially weighted averages also called as exponentially weighted moving averages in statistics.

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

V_t is approximated as $\frac{1}{1-\beta}$

which means if $\beta = 0.9$, $V_t \approx$ Average of 10 days temperature.
if $\beta = 0.98$, $V_t \approx$ Average of 50 days temperature, which means it adapts slowly to a temperature change, MORE LATENCY.

To get a better understanding,

$V_{100} = 0.9 * V_{99} + 0.1 * \theta_{100}$

$V_{99} = 0.9 * V_{98} + 0.1 * \theta_{99}$

$V_{98} = 0.9 * V_{97} + 0.1 * \theta_{98}$

$V_{100} = 0.1 * \theta_{100} + 0.9 * (0.9 * V_{98} + 0.1 * \theta_{99})$

$V_{100} = (0.1 * \theta_{100}) + (0.1 * 0.9 * \theta_{99}) + (0.1 * (0.9)^2 * \theta_{98}) + (0.1 * (0.9)^3 * \theta_{97}) \dots\dots + (0.1 * (0.9)^{98} * \theta_2) + (0.1 * (0.9)^{99} * \theta_1))$

$$(0.9)^{10} \approx 0.35 \approx \frac{1}{e} \qquad (1-\varepsilon)^{\frac{1}{\varepsilon}} = \frac{1}{e}$$

Why not simple averages and moving averages?

- More memory
- More computations

After this, we learnt about Bias correction in exponentially weighted average

$V_0 = 0$

$V_1 = 0.98 * V_0 + 0.1 * \theta_1 = 0.1 * \theta_1$ -> This is very less, similarly the initial few terms will very less.
So, we will add another term called Bias to nullify this problem.

Bias Correction, $V_t := \frac{V_t}{1-\beta^t}$

We can use this concept to calculate dW and db and update the parameters accordingly, what this does is it reduces the vertical oscillations and move horizontally towards the minimum.
As, the vertical average is '0', and horizontal average is towards right, we will reach the minimum quickly i.e. in fewer iterations.

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

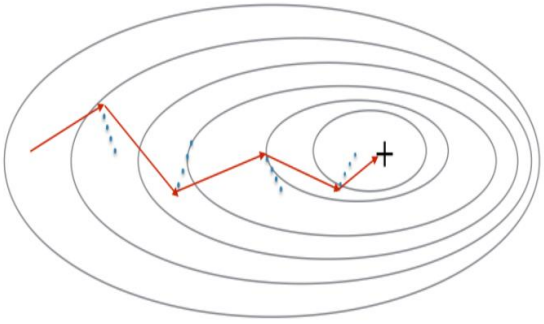
$V_{db} = \beta * V_{db} + (1-\beta) * db$

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

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

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.

Momentum takes into account the past gradients to smooth out the update. We will store the 'direction' of the previous gradients in the variable V. Formally, this will be the exponentially weighted average of the gradient on previous steps. You can also think of V as the "velocity" of a ball rolling downhill, building up speed (and momentum) according to the direction of the gradient/slope of the hill.



The red arrows show the direction taken by one step of mini-batch gradient descent with momentum.
The blue points show the direction of the gradient (with respect to the current mini-batch) on each step.
Rather than just following the gradient, we let the gradient influence V and then take a step in the direction of V.

Next, we learnt about **RMS** (**Root Mean Square**) **prop**, the goal is still the same reduce vertical moment increase horizontal speed.

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

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

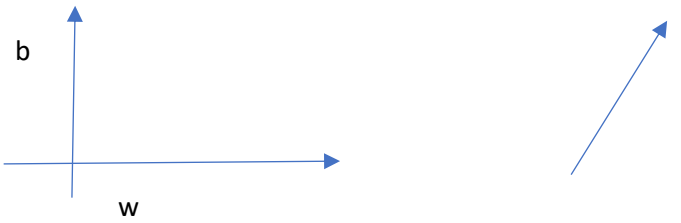
This is an element wise operation

$$W = W - \alpha * \frac{dW}{\sqrt{S_{dW}} + \varepsilon}$$

$$b = b - \alpha * \frac{db}{\sqrt{S_{db}} + \varepsilon}$$

Numerical Stability, so you don't end up dividing with '0'

This derivatives are much larger in vertical direction than in horizontal direction, so slope is very large in b direction.
So, it has a very large db and a very small dW.
So, db^2 is relatively large to dW^2
So, S_{db} is relatively larger than S_{db}.
So, b (vertical direction) is divided by larger number, which will damper the oscillations in vertical direction and updates in horizontal direction are divided by a smaller number.
The net effect with RMS prop would be in such a way that your updates would end up looking in such a way that vertical oscillations are damped up and you can go in horizontal direction.
So, we can also use a faster learning rate α and increase the speed of the learning.



Note that:

- If β=0, then this just becomes standard gradient descent without momentum.
- Momentum takes past gradients into account to smooth out the steps of gradient descent. It can be applied with batch gradient descent, mini-batch gradient descent or stochastic gradient descent.
- You have to tune a momentum hyperparameter β and a learning rate α.

Next, we learn about **Adam**.
The **name** Adam comes from **Adaptive moment estimation**.

Adam is one of the most effective optimization algorithms for training neural networks. It combines ideas from RMSProp (described in lecture) and Momentum.

How does Adam work?

1. It calculates an exponentially weighted average of past gradients, and stores it in variables V (before bias correction) and $V^{Corrected}$ (with bias correction).
2. It calculates an exponentially weighted average of the squares of the past gradients, and stores it in variables S (before bias correction) and $S^{Corrected}$ (with bias correction).
3. It updates parameters in a direction based on combining information from "1" and "2".

$V_{dW} = 0, S_{dW} = 0$
$V_{db} = 0, S_{db} = 0$
On iteration t:

Compute dW,db using current mini-batch.

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

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

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

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

Adding Bias Correction:

$$V_{dW}^{Corrected} = \frac{V_{dW}}{\sqrt{1-\beta_1^t}}, V_{db}^{Corrected} = \frac{V_{db}}{\sqrt{1-\beta_1^t}}$$

"MOMENNTUM"

$$S_{dW}^{Corrected} = \frac{S_{dW}}{\sqrt{1-\beta_2^t}}, S_{db}^{Corrected} = \frac{S_{db}}{\sqrt{1-\beta_2^t}}$$

"RMS Prop" β_2

$$W = W - \alpha * \frac{V_{dW}^{Corrected}}{\sqrt{S_{dW}^{Corrected}} + \varepsilon}$$

$$b = b - \alpha * \frac{V_{db}^{Corrected}}{\sqrt{S_{db}^{Corrected}} + \varepsilon}$$

Hyperparameters Choice:
α = needs to be tuned
β_1= 0.9 (dW, db)
β_2= 0.99 (dW^2, db^2)
$\varepsilon = 10^{-8}$

Next, we have seen about learning rate decay.
We understood the reason why we have to change our learning rate α.
Initially, we can have a large running rate but later we want a slower learning rate, so we will modify our learning rate accordingly:

1 epoch is 1 pass through the data

$$\alpha = \frac{\alpha_0}{1+(decay-rate*epoch-number)}$$

Other Methods:

1. $\alpha = 0.95^{epoch-number}$ -> exponentially decay
2. $\alpha = \frac{K}{\sqrt{epoch-number}}\alpha_0$ (or) $\frac{K}{\sqrt{t}}\alpha_0$
3. We also use discreet stair case, so that the learning-rate decreases as we go along.
4. Manual decay: works only if you are having smaller number of models

We then saw the problem of local optima and plateaus.

Problem of local optima:

- Most points of '0' are not local optima, most of them are "Saddle point".

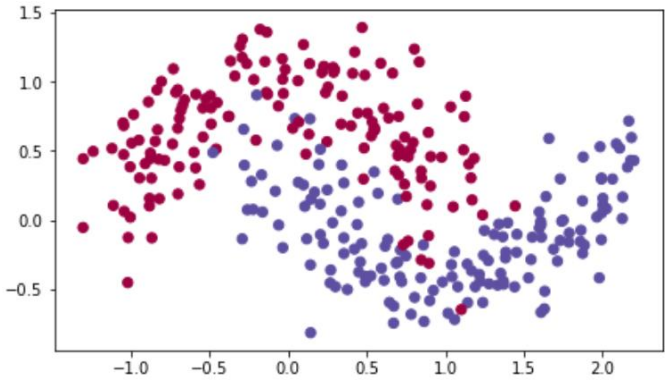
Problem of Plateaus:
Plateau is a region where derivative is closer to '0' for a long time.

- Unlikely to get stuck in bad local optima.
- Plateaus can make learning slow.

Programming Assignment:

Problem Statement:

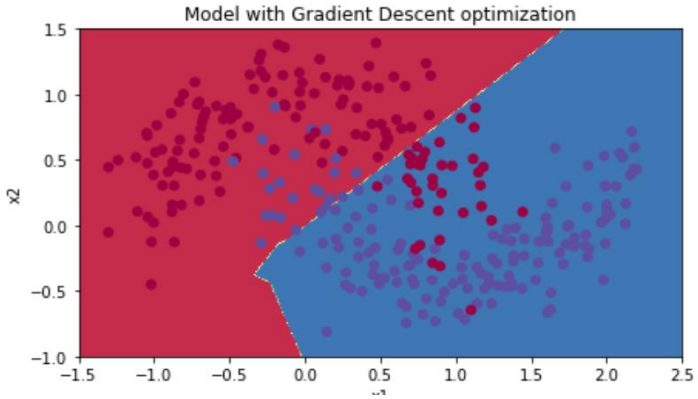
The following "moons" dataset to test the different optimization methods.
(The dataset is named "moons" because the data from each of the two classes looks a bit like a crescent-shaped moon.)

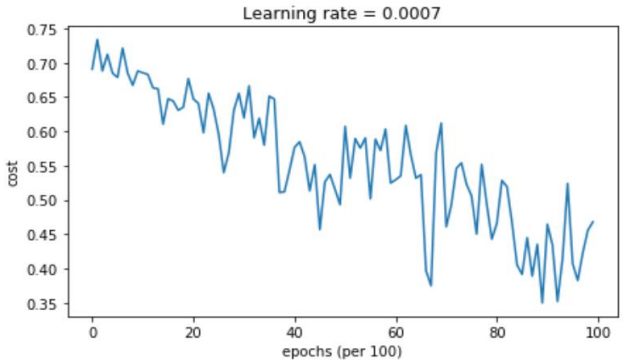


We have already implemented a 3-layer neural network.
We will train it with:

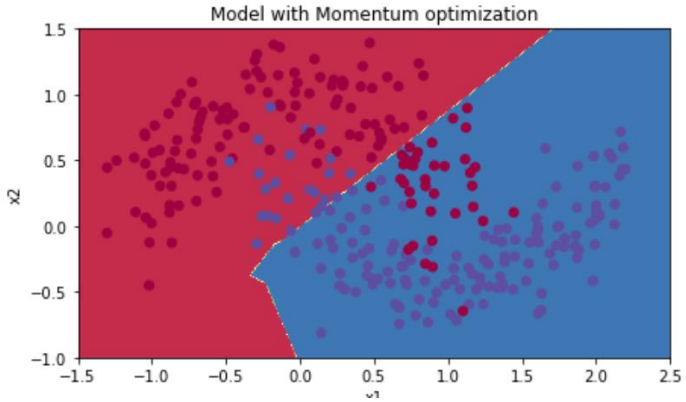
- Mini-batch Gradient Descent
- Mini-batch Momentum
- Mini-batch Adam

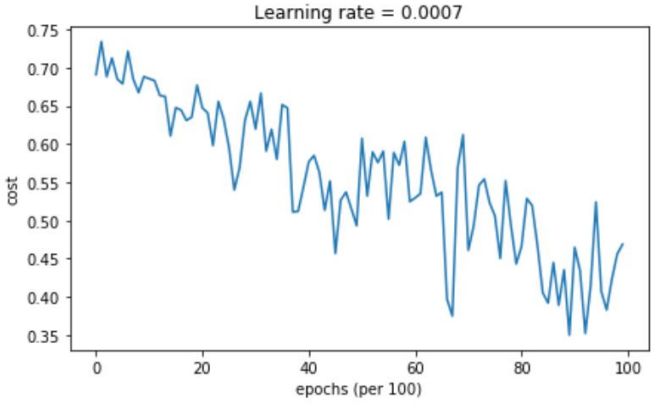
Mini-batch Gradient Descent



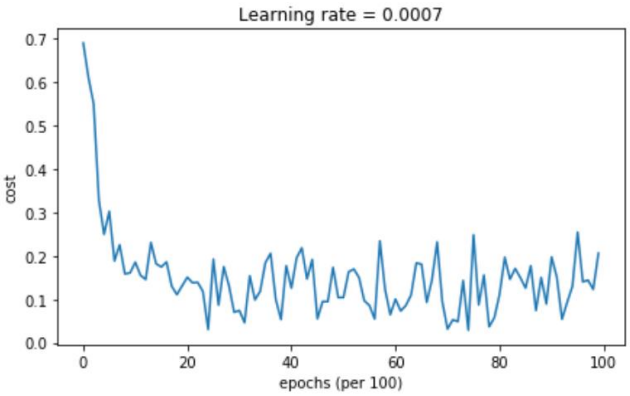


Mini-batch Momentum





Mini-batch Adam



Observations:

Momentum usually helps, but given the small learning rate and the simplistic dataset, its impact is almost negligible.
Also, the huge oscillations we see in the cost come from the fact that some minibatches are more difficult than others for the optimization algorithm.

Adam on the other hand, clearly outperforms mini-batch gradient descent and Momentum.
If you run the model for more epochs on this simple dataset, all three methods will lead to very good results. However, you've seen that Adam converges a lot faster.

Some advantages of Adam include:

- Relatively low memory requirements (though higher than gradient descent and gradient descent with momentum)
- Usually works well even with little tuning of hyperparameters (except α)

How do you choose β_1?

- The larger the momentum β_1 is, the smoother the update because the more we take the past gradients into account. But if β_1 is too big, it could also smooth out the updates too much.
- Common values for β_1 range from 0.8 to 0.999.
  If you don't feel inclined to tune this, β_1=0.9 is often a reasonable default.
- Tuning the optimal β_1 for your model might need trying several values to see what works best in term of reducing the value of the cost function J.

Results:

Gradient Descent:	Momentum:	Adam:
Accuracy – 79.7 %	Accuracy – 79.7 %	Accuracy – 94 %
Comment: Oscillations in the cost function.	Comment: Oscillations in the cost function.	Comment: Smoother cost function