Course 2 - Hyperparameters, Regularization and Optimization

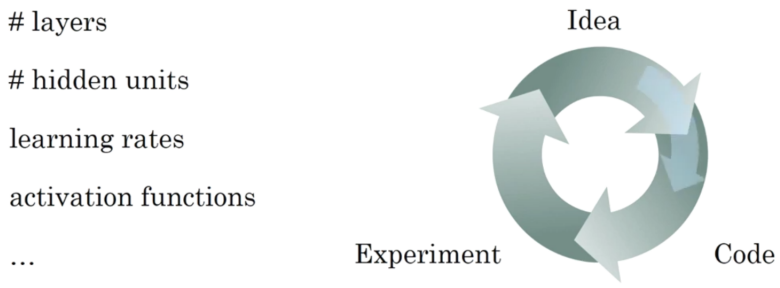
Mohammad Khalaji

August 16, 2020

1 Week 1

1.1 Train/Dev/Test Sets

We should always consider the fact that applied ML is an extremely iterative process with many parameters that affect our final outcome. It is virtually impossible to guess those parameters correctly in our first attempt.

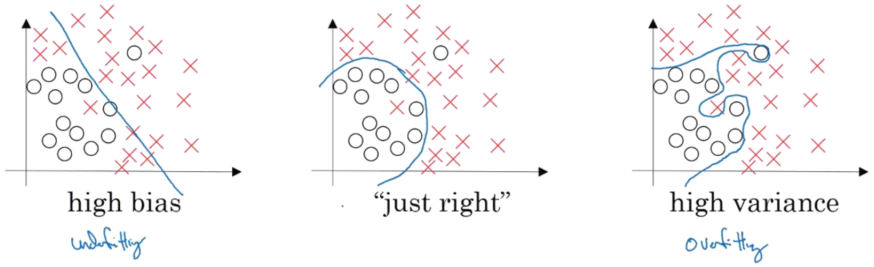


In the previous era of machine learning, it was common to divide all the data and split it according to a 70%/30% proportion, and assign them as your train and test sets respectively. Nowadays, however, with the emergence of big data, dev and test sets occcupy a smaller percentage of our data. For example, when we have a dataset with 100000 examples, an 20% dev set is too big! As a result, the proportions nowadays look like 99.5%/0.25%/0.25%.

Not having a test set might be okay (only train/dev sets).

1.2 Bias/Variance

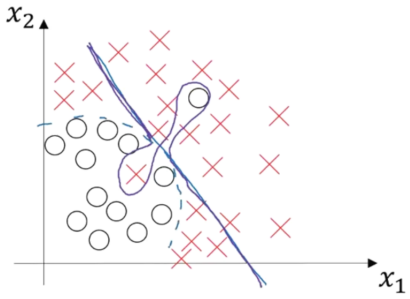
The simplest explanation:



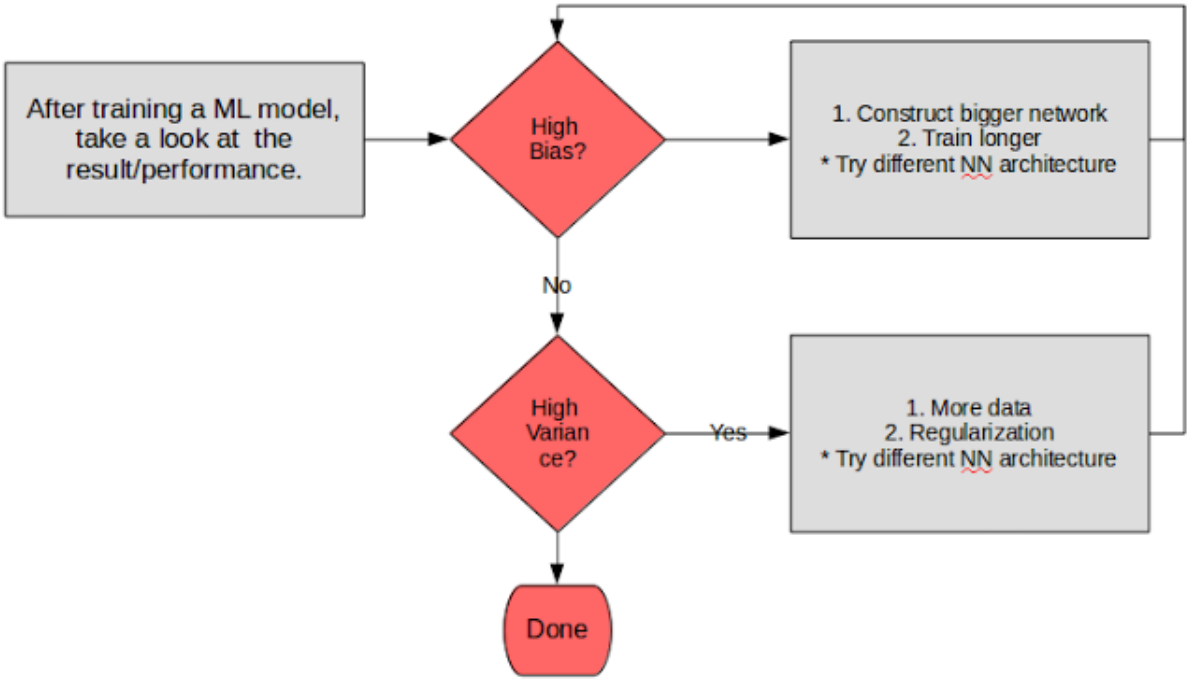
Consider another example for a cat classifer.

	High Variance	High Bias	High Bias High Variance	Low Bias Low Variance
Train Error	1%	15%	15%	0.5%
Test Error	11%	16%	30%	1.5%

The following classifier has both high bias and high variance. High bias because it is a mostly linear classifier, and it mostly underfits; high variance because in some occasions it shows overfitting behavior.



1.3 Basic Recipe for Machine Learning



1.4 Regularization

If you suspect that your neural network has a high variance problem, meaning that it overfits your data, regularization is one of the first things you should try. The other option is to get more training data, but it is not always viable.

Remember that in logistic regression, we were trying to minimize $J(w, b)$. Now we add the regularization term to the previous J.

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \|w\|_2^2$$

It's called L_2 regularization because it uses L_2 norm, which is:

$$\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$$

It is also possible that we use L_1 regularization with the regularization term:

$$\frac{\lambda}{2m} \sum_{j=1}^{n_x} |w| = \frac{\lambda}{2m} \|w\|_1$$

Note that also a regularization term for bias is possible, but since b is not as high-dimentional as w, its regularization term, which is equal to $\frac{\lambda}{m} b^2$, is usually omitted.

In a more complex neural network, though, a more generalized equation will be like this:

$$J(W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^{L} \left\| W^{[l]} \right\|_F^2$$

$\left\| W^{[l]} \right\|_F^2$ is called the *Frobenius Norm* of the matrix $W^{[l]}$, and knowing that the shape of $W^{[l]}$ is $(n^{[l-1]}, n^{[l]})$ it's computed like this:

$$\left\| W^{[l]} \right\|_F^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} \left(w_{ij}^{[l]} \right)^2$$

Regularization is also applied to the gradients (it is called *weight decay*):

$$dw^{[l]} = (from\ backprop) + \frac{\lambda}{m} w^{[l]}$$

1.5 Why Regularization Reduces Overfitting

Consider a situation where you have a neural network that overfits your data. Remember the original non-regularized cost function:

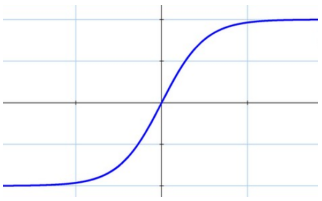
$$J = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

In order to reduce overfitting, we penalized J with a regularization term:

$$J = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \left\| W^{[l]} \right\|_F^2$$

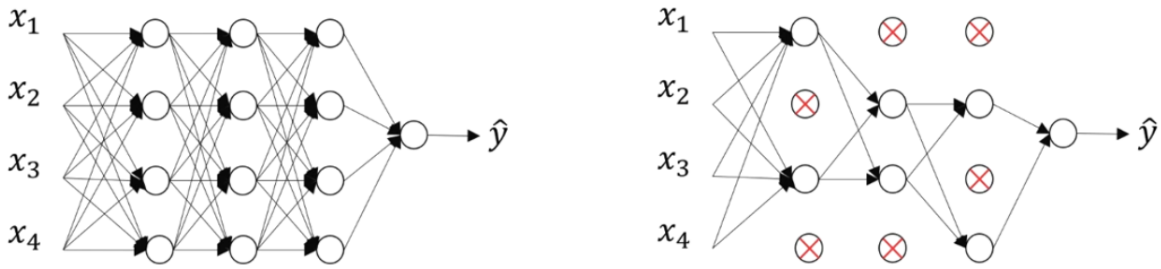
Intuition: If we set λ to be a really big number, our minimization algorithm will seek to set $W^{[l]} = 0$, which essentially means that a lot of hidden units will be zeroed out, which means that their impact will be reduced, and we will end up with a simpler neural network that is closer to the "High Bias" realm than it is to the "High Variance" realm.

Another Intuition: Consider using hyperbolic tangent as the activation function. If we regularize our weights, they will become smaller, and as a result, $z = wx + b$ will be smaller. With smaller zs, we will be wandering in the "linear section" of *tanh*, which means that each layer will be *roughly* linear, which leaves our network having higher bias.



1.6 Dropout Regularization

For each node in the network, set some probability for eliminating it. After selecting the to-be-eliminated nodes, remove all the outgoing and incoming links from and to that node, and then do backpropagation on the diminished network.



Since you are training a much smaller network *on each example*, you will end up with a regularized network.

Dropout Implementation (Inverted Dropout): Keep in mind that this works for both the vectorized version and the non-vectorized version. Given $l = 3$, $keepProb = 0.8$:

$$d^{[3]} = np.random.rand(a^{[3]}.shape)$$

$$a^{[3]} = d^{[3]} * a^{[3]}$$

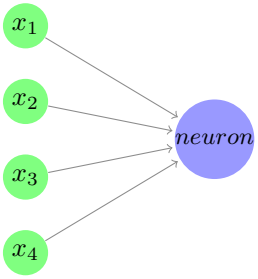
$$a^{[3]} /= keepProb$$

What does the last line do? It keps the expected value of $a^{[3]}$ stable so as to not reduce the value of $z^{[4]} = W^{[4]}a^{[3]} + b^{[4]}$.

Obviously, there will be no dropout in the test phase, because we do not want to use a random element in our final computation.

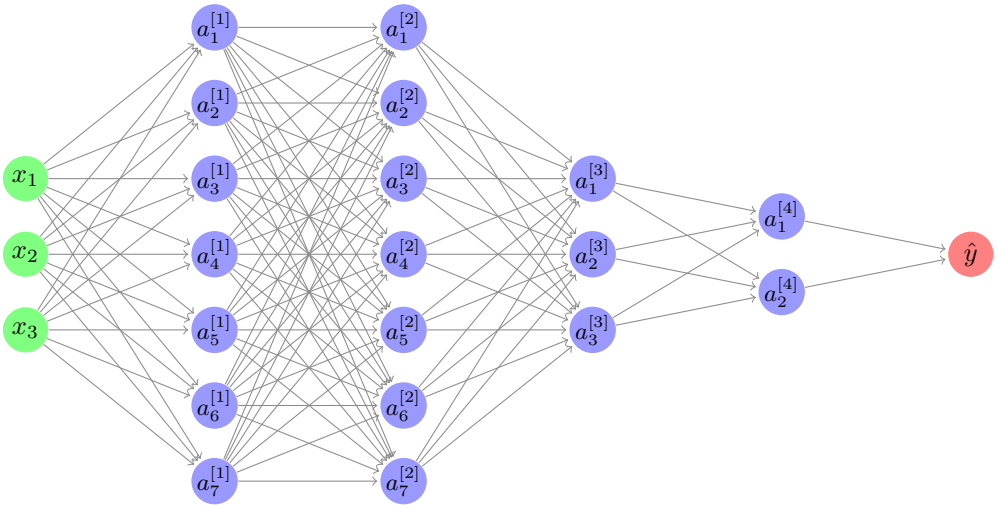
1.7 Understanding Dropout

Consider this one particular purple neuron:



As you can see, it relies on 4 features for its value, as a result, using droupout makes the network reluctant to rely heavily on *one* particular feature, employing a more *egalitarian* approach. Droupout has a similar effect to L_2 regularization's.

Another worthy note: It is good practice to keep *keepProp* variable according to the layer:

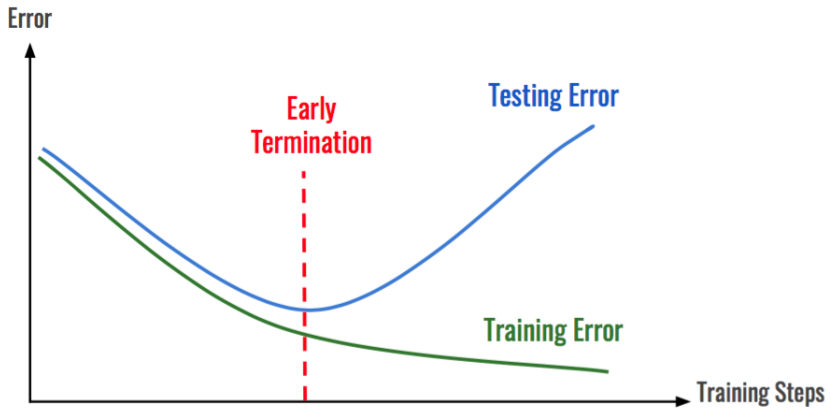


You might choose to use a lower *keepProb* for layers that have more parameters, because you are more worried about their overfitting ($l = 2$ for example).

1.8 Other Regularization Methods

Data Augmentation: Rotating, flipping, distorting and scaling train data images, for example.

Early Stopping:



It does have a downside though, according to the *orthogonalization* principle, we want to worry about one thing at a time. So worrying about the minimization of J, and worrying about not overfitting should be handled separately. Early stopping prevents us from doing that.

1.9 Normalizing Inputs

1. **Subtract Mean:**

$$\mu = \frac{1}{m}\sum_{i=1}^{m} X^{(i)}$$

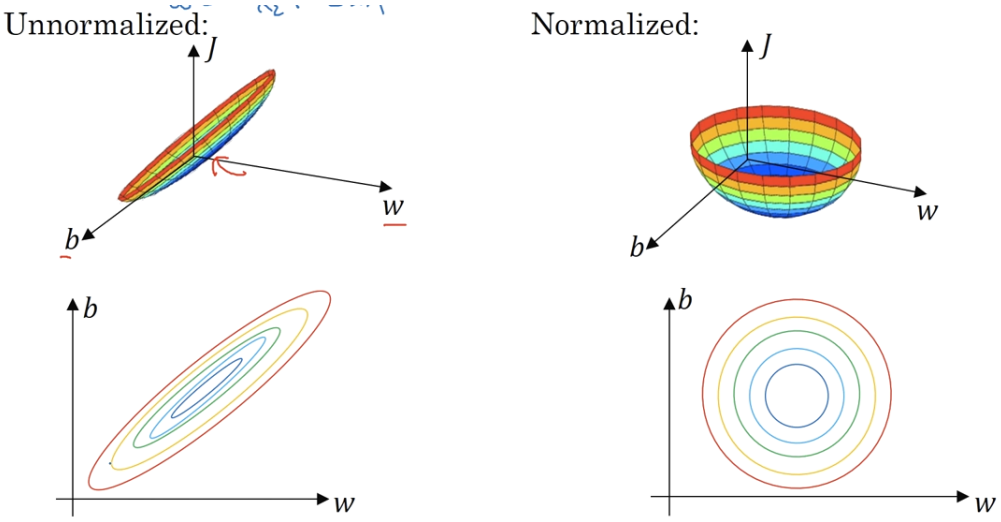
$$X^{(i)} = X^{(i)} - \mu \qquad \forall\ 1 \leq i \leq m$$

2. **Normalize Variance:**

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} X^{(i)} * * 2$$

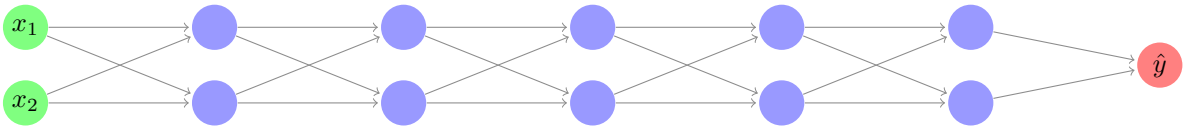
$$X^{(i)} = X^{(i)}/\sigma^2 \qquad \forall\ 1 \leq i \leq m$$

Important: Use the same μ, σ^2 to normalize your test set. DO NOT normalize train/dev/test sets with different values of μ, σ^2.



1.10 Exploding/Vanishing Gradients

Consider the following deep network, where $g^{[l]}(z) = z, b^{[l]} = 0$ in all layers:



$$\hat{y} = W^{[L]}W^{[L-1]}\dots W^{[2]}W^{[1]}X$$

If we have either of the scenarios below, the final $\hat{y}$ will be either exponentially big or small, making it very difficult to train the network.

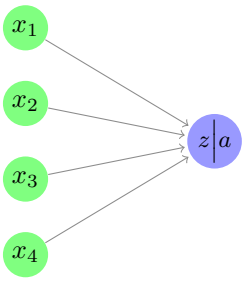
$$W^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix} \Rightarrow \hat{y} = W^{[L]}(1.5)^{L-1}X$$

$$W^{[l]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \Rightarrow \hat{y} = W^{[L]}(0.5)^{L-1}X$$

There is a *partial* solution to this problem, which is careful initialization of the weights.

1.11 Weight Initialization in a Deep Network

Consider this single neuron example, $b = 0$:



$$z = w_1 x_1 + \cdots + w_n x_n$$

Now we don't want z to become too big or too small, so with larger n, we want smaller w_is. One solution is to set $Variance(w_i) = \frac{1}{n}$.

$$w^{[l]} = np.random.randn(shape) * np.sqrt(\frac{1}{n^{[l-1]}})$$

(For ReLU activation functions, $Variance(w_i) = \frac{2}{n^{[l-1]}}$ works better)

(For *tanh* activation function, $Variance(w_i) = \sqrt{\frac{1}{n^{[l-1]}}}$ and $Variance(w_i) = \sqrt{\frac{2}{n^{[l-1]}+n^{[n]}}}$ work better)

1.12 Numerical Approximation of Gradients

$$g(\theta) \approx \frac{f(\theta + \epsilon) - f(\theta + \epsilon)}{2\epsilon}$$

1.13 Gradient Checking

To verify your backprop implementation is correct.

- Take all your parameters, $W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}$, and reshape them into a big vector θ.
- Take all your gradients, $dW^{[1]}, db^{[1]}, \ldots, dW^{[L]}, db^{[L]}$, and reshape them into a big vector $d\theta$.
- Now the question is: Is $d\theta$ the gradient of $J(\theta)$?

Here's how it's done:

- For each i:

$$d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2, \ldots, \theta_i + \epsilon, \ldots) - J(\theta_1, \theta_2, \ldots, \theta_i - \epsilon, \ldots)}{2\epsilon}$$

$$\approx d\theta[i] = \frac{\partial J}{\partial \theta_i}$$

- Now that you have the two vectors $d\theta$ and $d\theta_{approx}$, compute this (the denominator is just a normalizer):

$$\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}, \epsilon = 10^{-7}$$

- If the result is 10^{-7} or smaller, your gradient is very likely to be working properly. However, you need to double check if the value is too big.

2 Week 2

2.1 Mini-Batch Gradient Descent

We saw how vectorization allows us to perform forward and back propagation easily on m training examples.

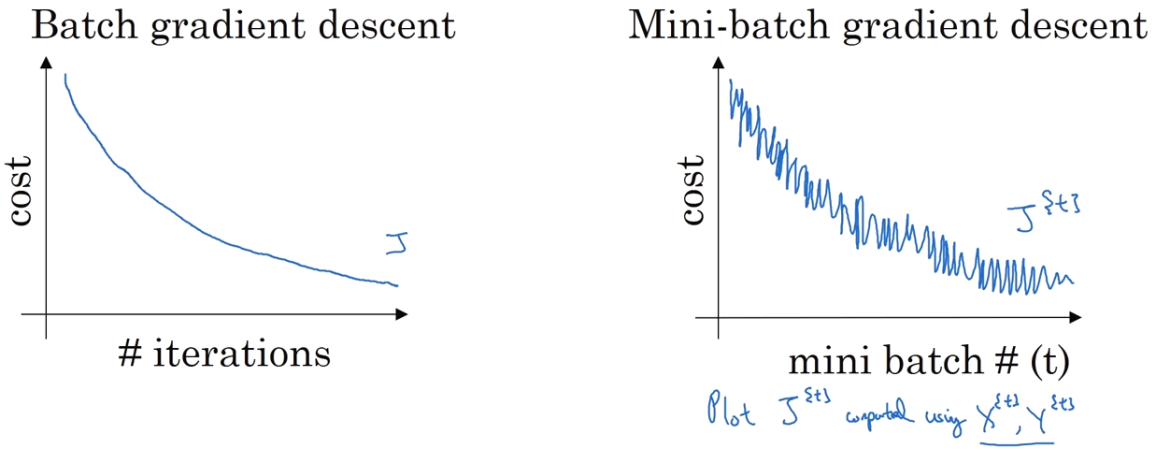
$$X_{(n_x,m)} = \begin{bmatrix} | & & | \\ x^{(1)} & \dots & x^{(m)} \\ | & & | \end{bmatrix}$$
$$Y_{(1,m)} = \begin{bmatrix} y^{(1)} & \dots & y^{(m)} \end{bmatrix}$$

What if $m = 5000000$? We split our training set into mini batches of, say 1000 training samples.

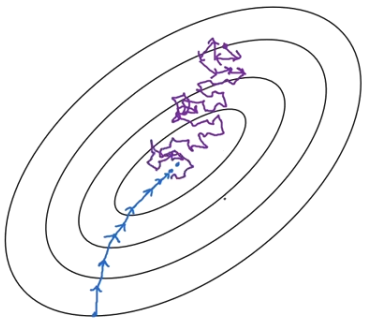
$$X\ Batches : X^{\{1\}}, \dots, X^{\{5000\}}$$
$$Y\ Batches : Y^{\{1\}}, \dots, Y^{\{5000\}}$$

- for $t = 1 \dots 5000$ do:
  - Forward Prop on $X^{\{t\}}$
    $Z^{[1]} = W^{[1]}X^{\{t\}} + b^{[1]}$
    $A^{[1]} = g^{[1]}(Z^{[1]})$
    $\dots$
    $Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$
    $A^{[L]} = g^{[L]}(Z^{[L]})$
  - Compute Cost: $J^{\{t\}} = \frac{1}{1000}\sum \dots + \dots$
  - Back Prop to compute gradients with respect to $J^{\{t\}}$.
  - Update parameters.

2.2 Understanding Mini-Batch Gradient Descent



- If mini-batch size equals m: Batch Gradient Descent (Blue)
- If mini-batch size equals 1: Stochastic Gradient Descent (Purple)



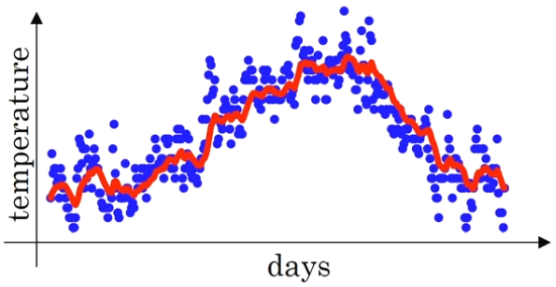
Batch Gradient Descent: Too long per epoch.

Stochastic Gradient Descent: Losing the speed-up of vectorization.

Somewhere in Between: Take advantage of the speed-up of vectorization + Make progress without needing to wait. Typical mini-batch sizes are $64, 128, 256, 512$. Make sure your mini-batches fit in CPU/GPU memory.

2.3 Exponentially Weighted Averages

Consider the following example in which we have a sequence of temperatures $\theta_1, \theta_2, \ldots$, which is shown by blue dots, and a special average sequence V shown in red:



Where:

$$v_t = \beta v_{t-1} + (1-\beta)\theta_1, \beta = 0.9$$

It can be shown that the equation above is the approximate average over the last $\approx \frac{1}{1-\beta}$ instances.

For example, if $\beta = 0.98$, we are averaging over 50 instances. If $\beta = 0.5$, we are averaging over 2 instances.

Understanding Weighted Averages:

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

$$= 0.1\theta_{100} + 0.9\Big(0.1\theta_{99} + 0.9v_{98}\Big)$$

$$= \ldots$$

$$= 0.1\theta_{100} + (0.9)(0.1)\theta_{99} + (0.1)(0.9)^2\theta_{98} + (0.1)(0.9)^3\theta_{97} + \ldots$$

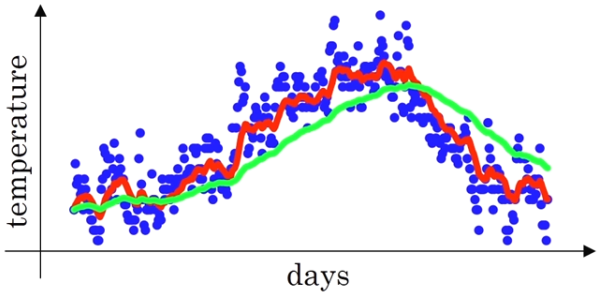
It turns out that all of the coefficients add up to a number very close to 1.

If we compute 0.9^10, we will reach $\approx 0.35 \approx \frac{1}{e}$, which means that after 10 days, the coefficient will be almost a third of the coefficient of the current thay.

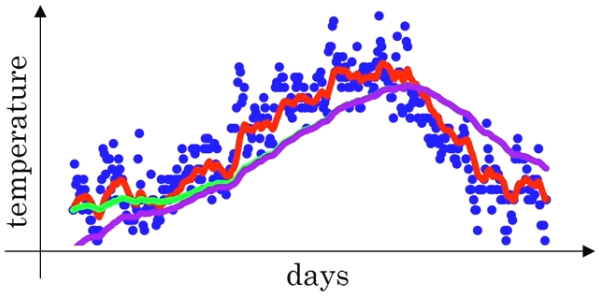
$$(1-\epsilon)^{\frac{1}{\epsilon}} \approx \frac{1}{e}$$

2.4 Bias Correction of Exponentially Weighted Averages

For $\beta = 0.98$, we would normally expect the green curve.



However, if we implement the formula $v_t = \beta v_{t-1} + (1-\beta)\theta_t$, we end up with the purple curve.



This phenomenon is called bias, and it happens because of our initialization of v_0, in which we set $v_0 = 0$.

$$v_0 = 0$$

$$v_1 = 0.98v_0 + 0.02\theta_1 = 0.02\theta_1$$

$$v_2 = 0.98v_1 + 0.02\theta_2 = (0.98)(0.02)\theta_1 + 0.02\theta_2$$

Obviously, our initial vs are smaller than expected. What is the remedy to this situation? Instead of calculating v_t, we calculate $\frac{v_t}{1-\beta^t}$

$$v_2 = \frac{(0.98)(0.02)\theta_1 + 0.02\theta_2}{1-(0.98)^2}$$

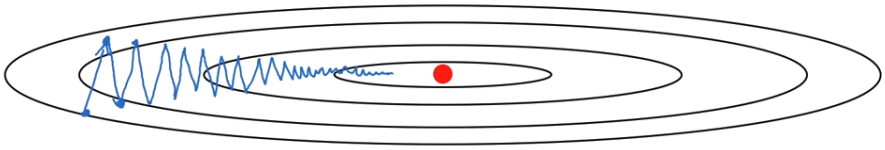
We can see that $(0.98)(0.02) = 0.0196$ and 0.02 in the nominator, add up to $1-(0.98)^2 = 0.0396$ in the denominator, making v_2 a weighted average of θ_1 and θ_2, thus removing the bias.

This bias correction method helps the purple curve get closer to the green curve.

2.5 Gradient Descent with Momentum

Compute an exponentially weighted average of the gradients, and use that to update your parameters.

The up and down oscillations in gradient descent algorithm prevents us from using a large learning rate, whcih results in the slow convergence of the algorithm.



- On iteration t:

  Compute dW, db on current mini-batch.
  $v_{dW} = \beta v_{dW} + (1-\beta)dW$
  $v_{db} = \beta v_{db} + (1-\beta)db$
  $W = W - \alpha v_{dW}$, $b = b - \alpha v_{db}$

We often see this algorithm with the $(1-\beta)$ omitted.

2.6 RMSProp

- On iteration t:

  Compute dW, db on current mini-batch.
  $s_{dW} = \beta s_{dW} + (1-\beta)dW^2$, (element-wise)
  $s_{db} = \beta s_{db} + (1-\beta)db^2$, (element-wise)
  $W = W - \alpha \frac{dW}{\sqrt{s_{dW}+\epsilon}}$, $b = b - \alpha \frac{db}{\sqrt{s_{db}+\epsilon}}$

Since we want to reduce the oscillations in vertical (b) direction, we are changing our update in a way that horizontal update has greater weight. s_{dW} is relatively small, whereas s_{db} is relatively large. As a result, the updates in the vertical direction are slowed down, whereas updates in the horizontal direction are speeded up.

More generally, in dimensions where vertical oscillations (oscillations that are not toward the minima) are existent, s will be relatively large, allowing us to use a larger learning rate.

2.7 Adam Optimization Algorithm

It basically takes momentum and RMSProp and puts them together as a new algorithm.

- On iteration t:

  Compute dW, db on current mini-batch.

  Momentum with β_1:

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

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

  RMSProp with β_2:

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

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

  Perform bias correction:

  $$v_{dW}^{corrected} = \frac{v_{dW}}{1 - \beta_1^t}$$

  $$v_{db}^{corrected} = \frac{v_{db}}{1 - \beta_1^t}$$

  $$s_{dW}^{corrected} = \frac{s_{dW}}{1 - \beta_2^t}$$

  $$s_{db}^{corrected} = \frac{s_{db}}{1 - \beta_2^t}$$

  Update parameters:

  $$W = W - \alpha \frac{v_{dW}^{corrected}}{\sqrt{s_{dW}^{corrected}} + \epsilon}$$

  $$b = b - \alpha \frac{v_{db}^{corrected}}{\sqrt{s_{db}^{corrected}} + \epsilon}$$

- α: Needs to be tuned
- $\beta_1 = 0.9$
- $\beta_2 = 0.999$
- $\epsilon = 10^{-8}$

2.8 Learning Rate Decay

Smaller learning rates tend to wander around tighter regions around the minimum. As a result, it is intuitive that we slow down our learning with the passage of time.

We know that 1 epoch means 1 pass through the data.

$$\alpha = \frac{1}{1 + decayRate * epoch}\alpha_0$$

There are other decay methods too:

$$\alpha = (0.95)^{epoch}\alpha_0$$

$$\alpha = \frac{k}{\sqrt{epoch}}\alpha_0$$