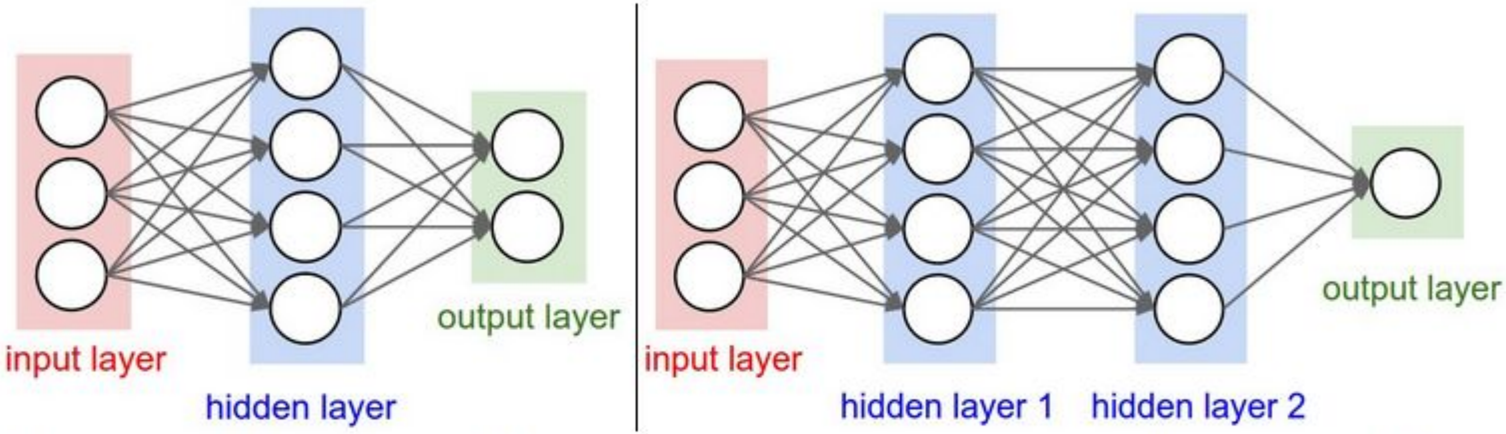
DNN_02

1. Neural Network Architecture
2. Forward Pass
3. Loss Function
4. Optimization
5. Gradient Descent
6. Learning Rate
7. Tips of Gradient Descent
8. Batch Gradient Descent
9. Stochastic Gradient Descent
10. Mini Batch Gradient Descent
11. Exponential Weighted Average
12. Gradient Descent with momentum
13. RMSprop
14. Adam
15. Learning Rate Decay
16. Local Optima

Neural Network Architectures



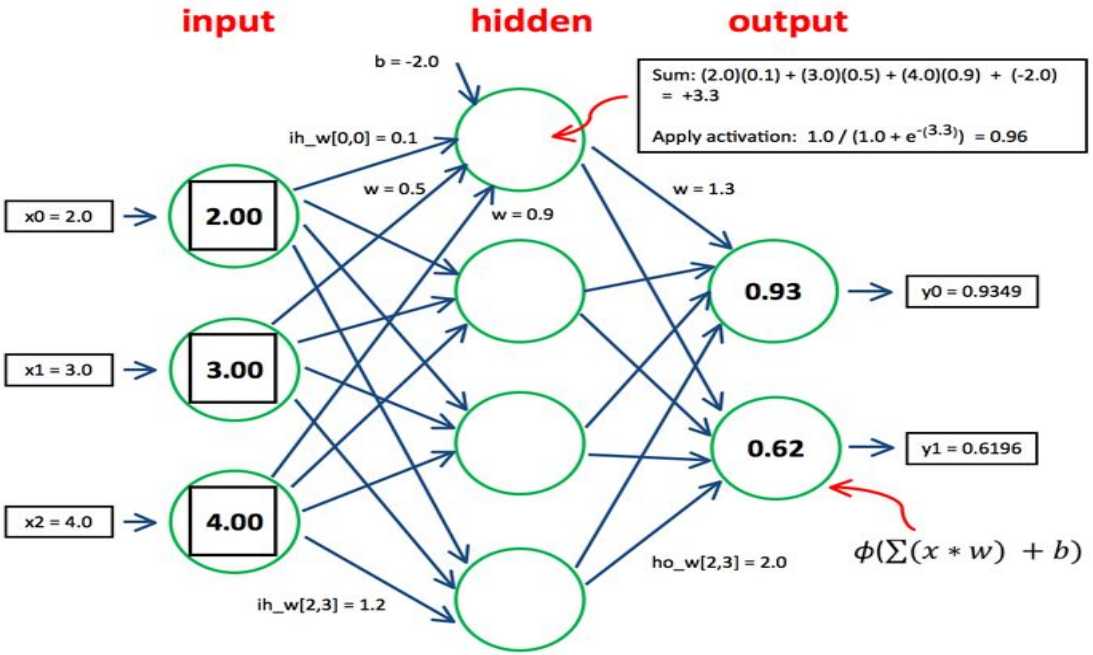
Left: A 2-layer Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs. **Right**: A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Notice that in both cases there are connections (synapses) between neurons across layers, but not within a layer.

Neural Network Architectures ..cont

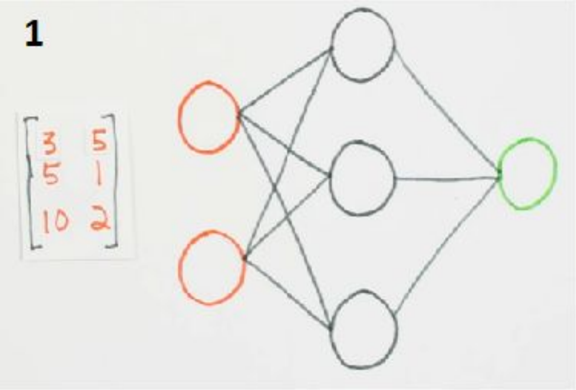
- The first network (left) has 4 + 2 = 6 neurons (not counting the inputs), [3 x 4] + [4 x 2] = 20 weights and 4 + 2 = 6 biases, for a total of 26 learnable parameters.
- The second network (right) has 4 + 4 + 1 = 9 neurons, [3 x 4] + [4 x 4] + [4 x 1] = 12 + 16 + 4 = 32 weights and 4 + 4 + 1 = 9 biases, for a total of 41 learnable parameters.

The forward pass of a fully-connected layer corresponds to one matrix multiplication followed by a bias offset and an activation function.

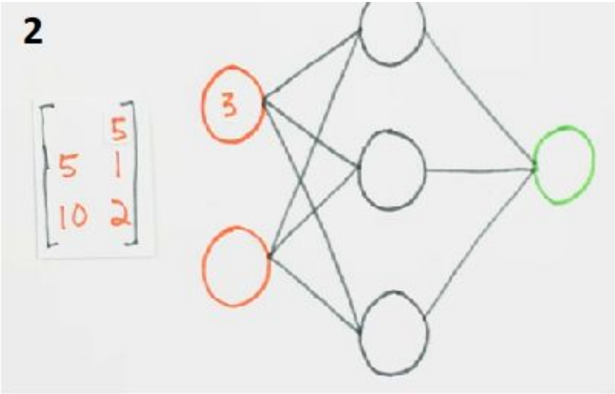
Forward Pass



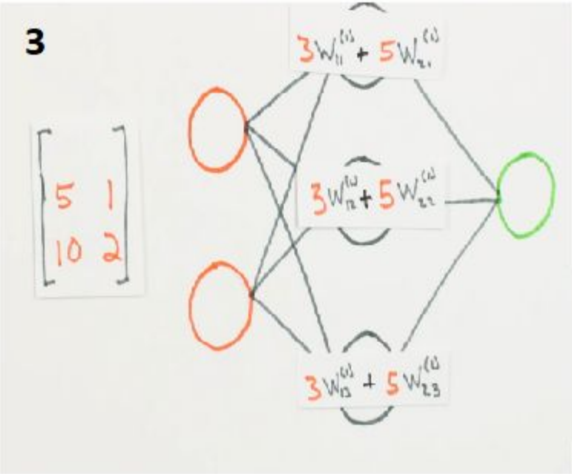
1



2



3

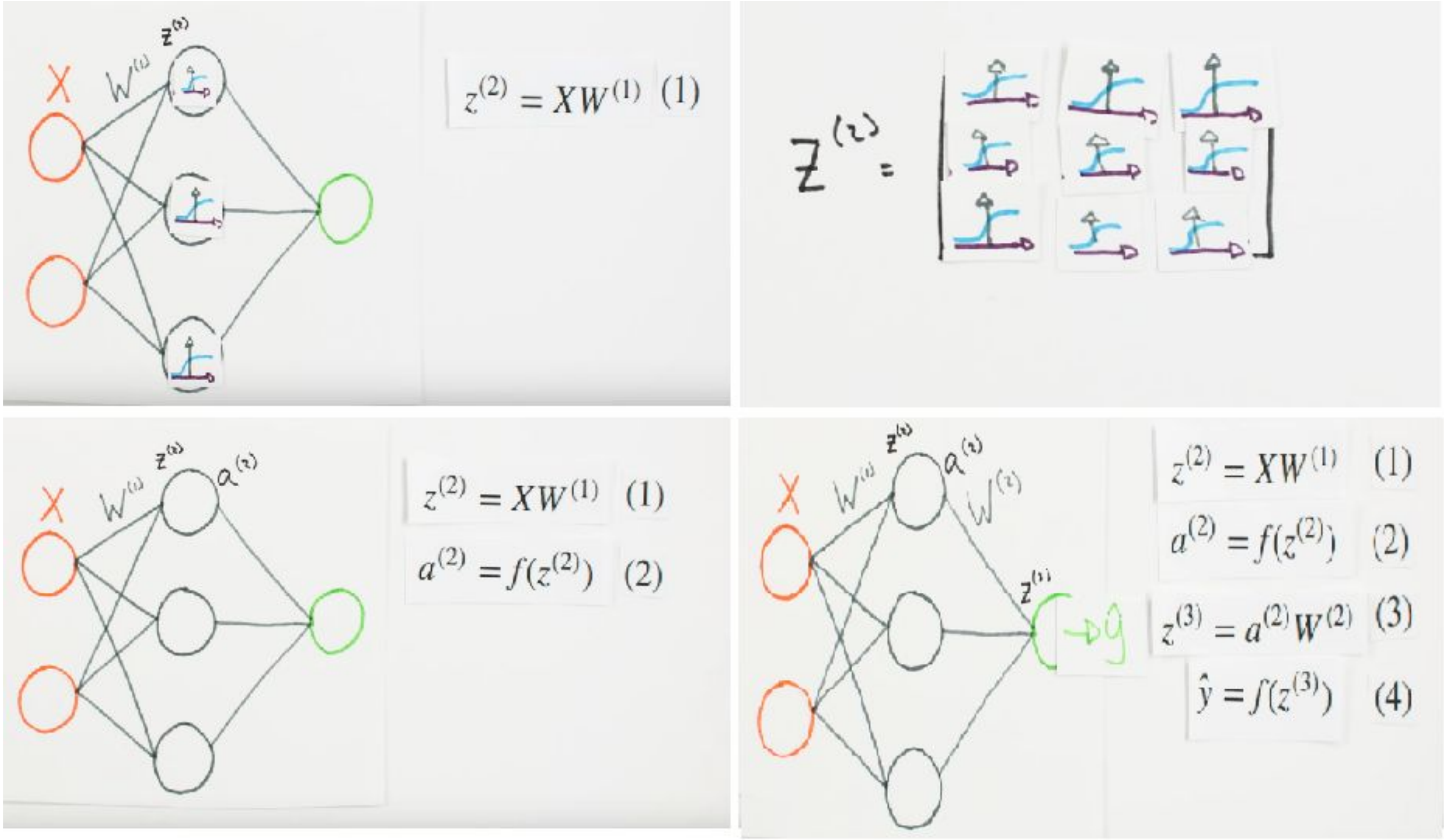


4

$$\begin{bmatrix} 3 & 5 \\ 5 & 1 \\ 10 & 2 \end{bmatrix} \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} & W_{13}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} & W_{23}^{(1)} \end{bmatrix} = \begin{bmatrix} 3W_{11}^{(1)} + 5W_{21}^{(1)} & 3W_{12}^{(1)} + 5W_{22}^{(1)} & 3W_{13}^{(1)} + 5W_{23}^{(1)} \\ 5W_{11}^{(1)} + 1W_{21}^{(1)} & 5W_{12}^{(1)} + 1W_{22}^{(1)} & 5W_{13}^{(1)} + 1W_{23}^{(1)} \\ 10W_{11}^{(1)} + 2W_{21}^{(1)} & 10W_{12}^{(1)} + 2W_{22}^{(1)} & 10W_{13}^{(1)} + 2W_{23}^{(1)} \end{bmatrix}$$

5

$$X \quad W^{(1)} = Z^{(2)}$$



X
$W^{(1)}$
$z^{(2)}$
$z^{(2)} = XW^{(1)}$ (1)
$Z^{(2)} =$
$a^{(2)}$
$a^{(2)} = f(z^{(2)})$ (2)
$W^{(2)}$
$z^{(3)}$
$\hat{y}$
$z^{(3)} = a^{(2)}W^{(2)}$ (3)
$\hat{y} = f(z^{(3)})$ (4)

```python
In [1]: class Neural_Network(object):
            def __init__(self):
                #Define Hyperparameters
                self.inputLayerSize = 2
                self.outputLayerSize = 1
                self.hiddenLayerSize = 3

                #Weights (Parameters)
                self.W1 = np.random.randn(self.inputLayerSize, \
                                          self.hiddenLayerSize)
                self.W2 = np.random.randn(self.hiddenLayerSize, \
                                          self.outputLayerSize)

            def forward(self, X):
                #Propagate inputs though network
                self.z2 = np.dot(X, self.W1)
                self.a2 = self.sigmoid(self.z2)
                self.z3 = np.dot(self.a2, self.W2)
                yHat = self.sigmoid(self.z3)
                return yHat

            def sigmoid(self, z):
                #Apply sigmoid activation function to scalar, vector, or
                return 1/(1+np.exp(-z))
```

$$z^{(2)} = XW^{(1)} \quad (1)$$

$$a^{(2)} = f(z^{(2)}) \quad (2)$$

$$z^{(3)} = a^{(2)}W^{(2)} \quad (3)$$

$$\hat{y} = f(z^{(3)}) \quad (4)$$

Loss/Cost/Objective Function

https://stats.stackexchange.com/questions/154879/a-list-of-cost-functions-used-in-neural-networks-alongside-applications

$$\mathcal{J}(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

Mean Square Error

$$MSE = \underset{\text{average over all results}}{\frac{1}{N}} \sum_{i=1}^{N} \left(\underset{\text{true } y}{y_i} - \underset{\text{estimate of } y}{\hat{y}_i}\right)^{\overset{\text{makes result quadratic}}{2}}$$

Mostly used in linear regression.

Cross Entropy

$$H(x) = \sum_{i=1}^{N} \overset{\text{true label}}{p}(x)\log \underset{\text{estimate}}{q}(x)$$

Mostly used in classification problem / logistic regression.

Methods of Optimization

1. Random Search
2. Random Local Search
3. Following the gradient

Gradient Descent

Gradient descent is an optimization algorithm used to find the values of parameters of a function (f) that minimizes a cost function (cost).

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

Hypothesis:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

Parameters:

$$\theta_0, \theta_1$$

Cost Function:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

Goal: $\underset{\theta_0, \theta_1}{\text{minimize}}\ J(\theta_0, \theta_1)$

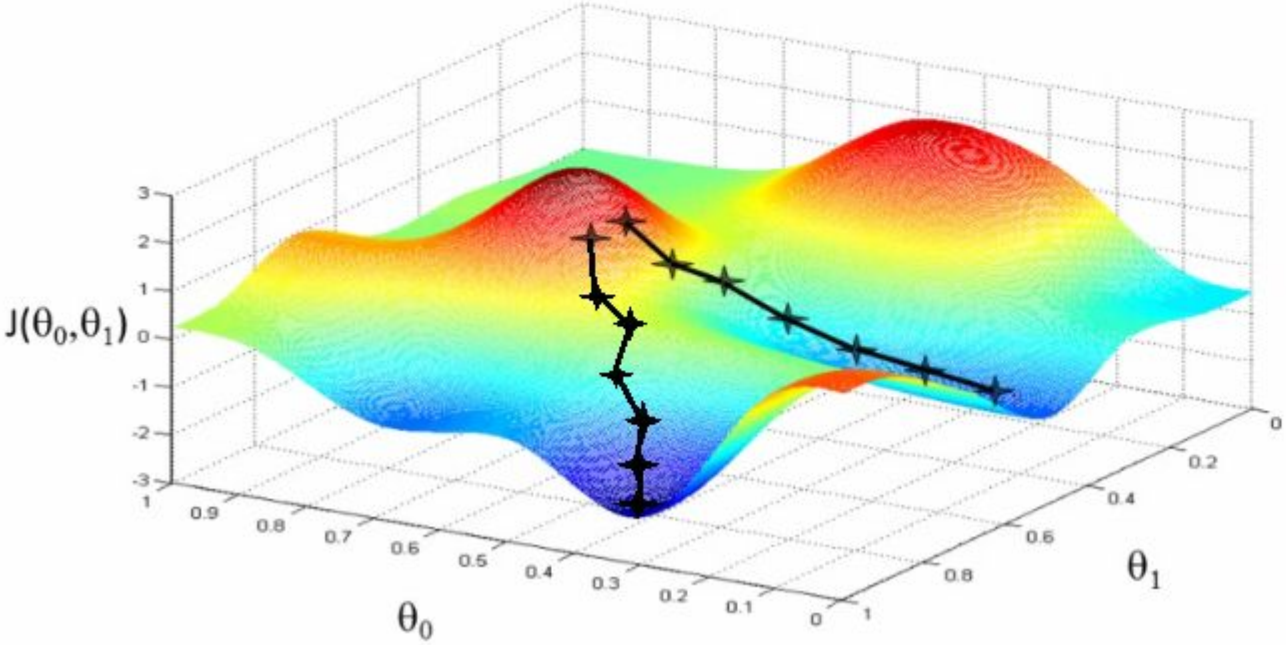
<u>Simplified</u>

$$h_\theta(x) = \theta_1 x$$

$$\theta_1$$

$$J(\theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

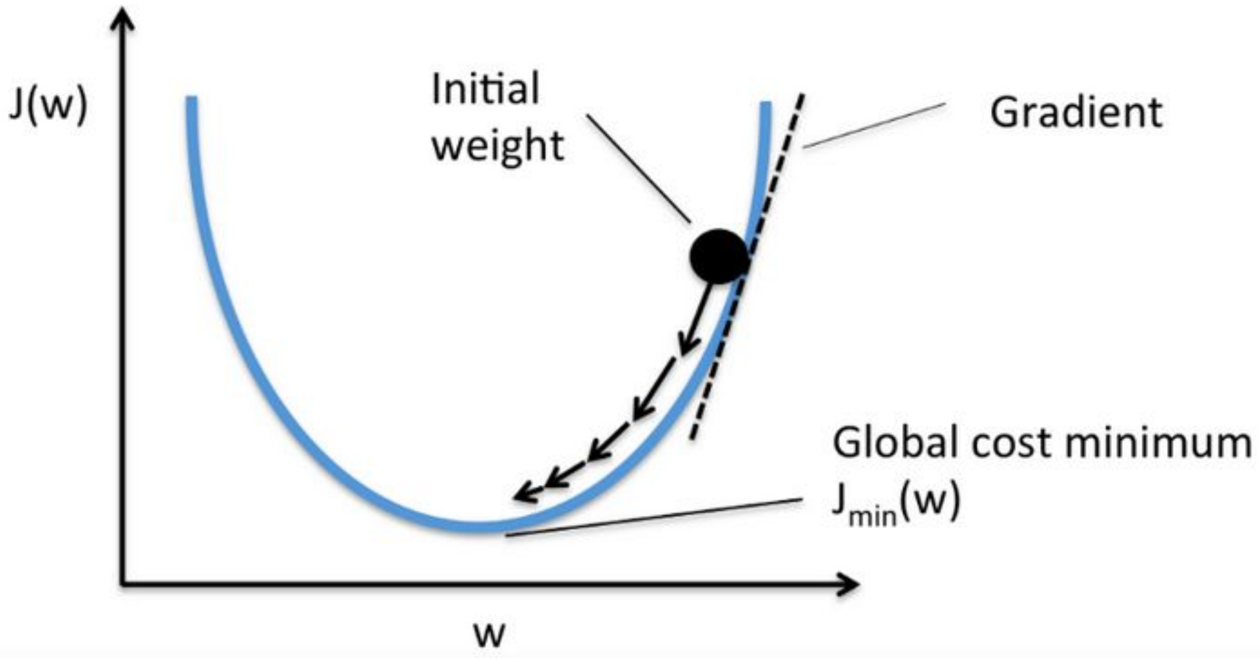
$$\underset{\theta_1}{\text{minimize}}\ J(\theta_1)$$



$J(\theta_0,\theta_1)$
3
2
1
0
-1
-2
-3
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
θ_0
0
0.2
0.4
0.6
0.8
1
θ_1

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad \text{(for } j = 0 \text{ and } j = 1\text{)}$$

Gradient Descent



Gradient Descent Algorithm

repeat until convergence {

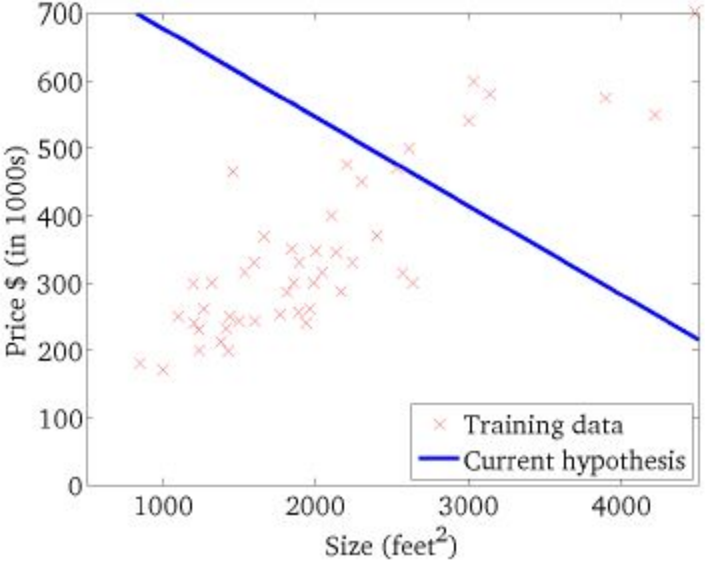
$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x^{(i)}$$

}

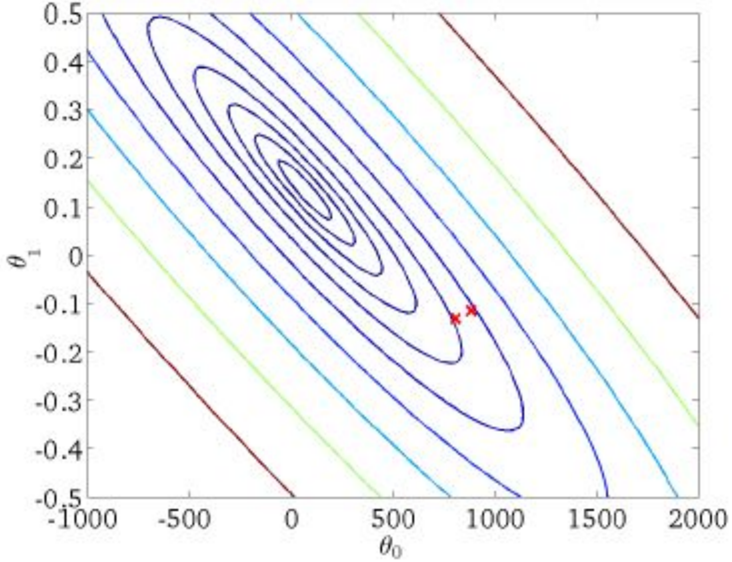
$$h_\theta(x)$$

(for fixed θ_0, θ_1, this is a function of x)



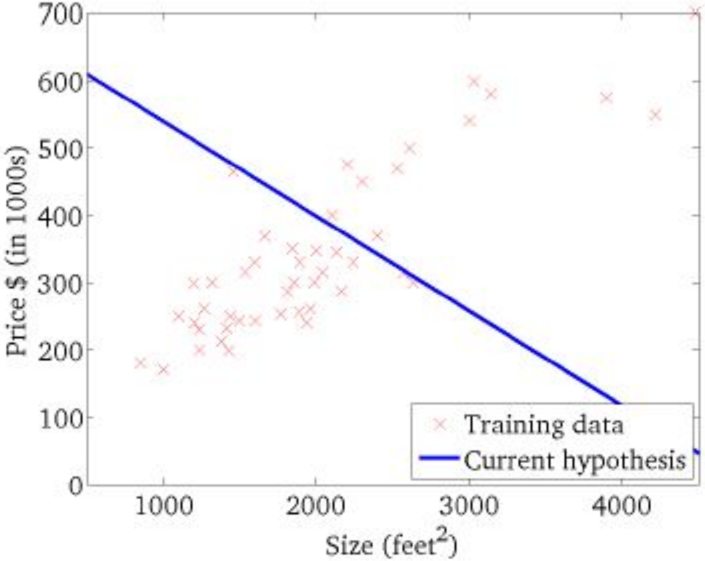
$$J(\theta_0, \theta_1)$$

(function of the parameters θ_0, θ_1)



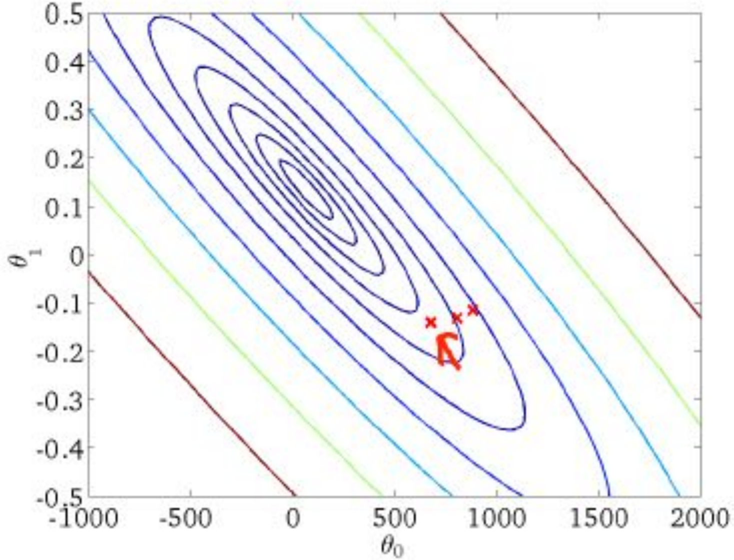
$h_\theta(x)$

(for fixed θ_0, θ_1, this is a function of x)



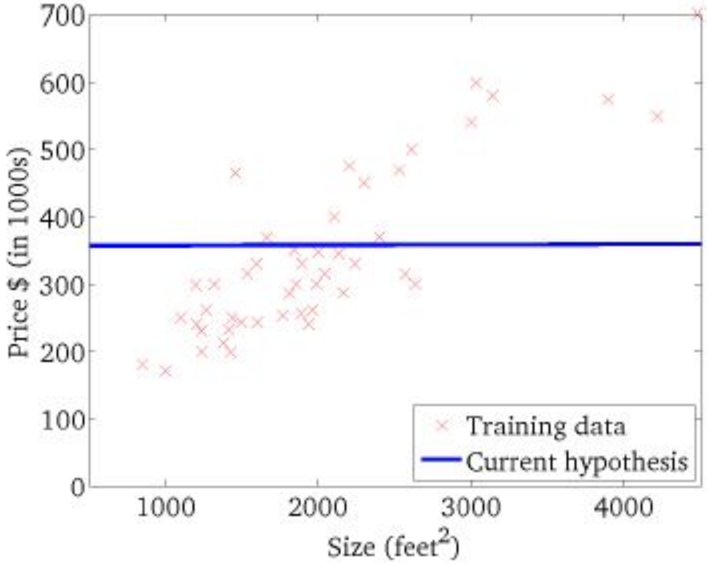
$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



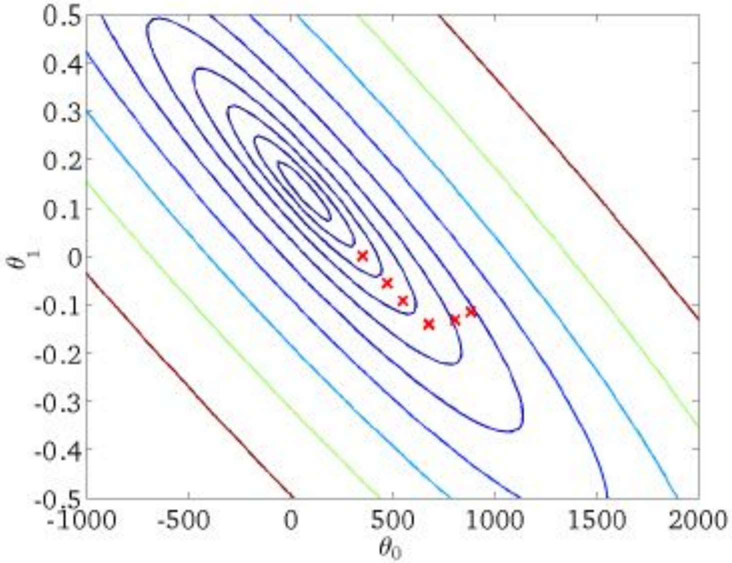
$$h_\theta(x)$$

(for fixed θ_0, θ_1, this is a function of x)



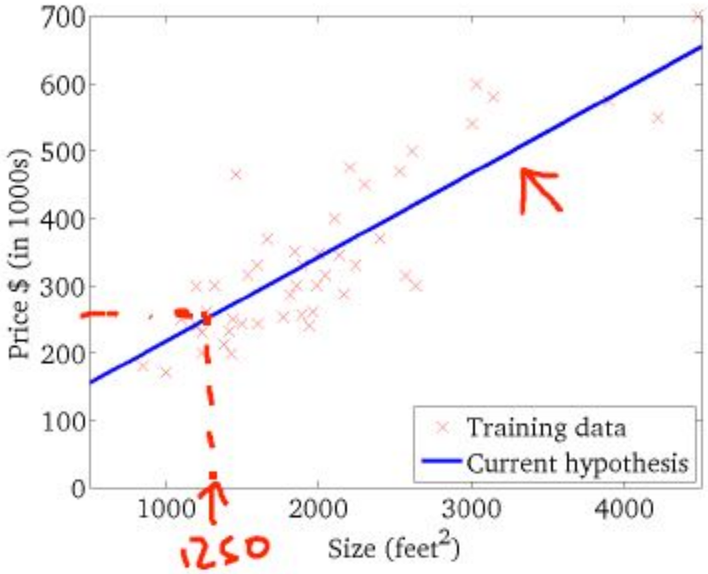
$$J(\theta_0, \theta_1)$$

(function of the parameters θ_0, θ_1)



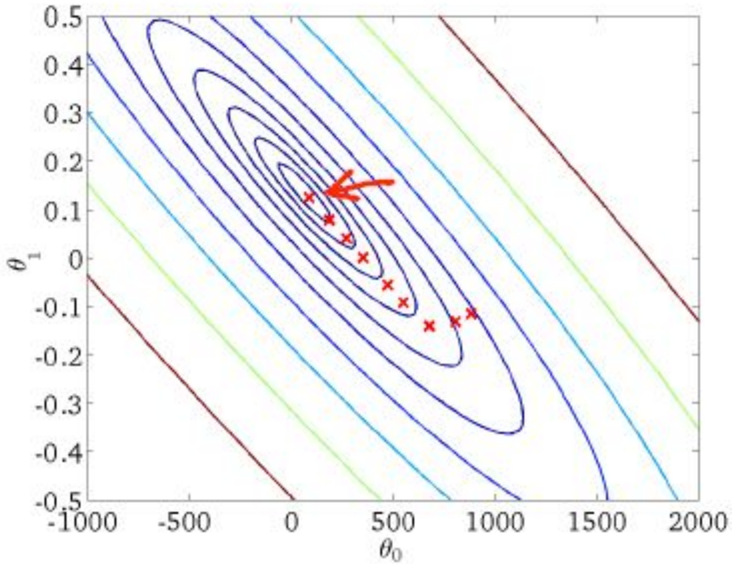
$$h_\theta(x)$$

(for fixed θ_0, θ_1, this is a function of x)



$$J(\theta_0, \theta_1)$$

(function of the parameters θ_0, θ_1)



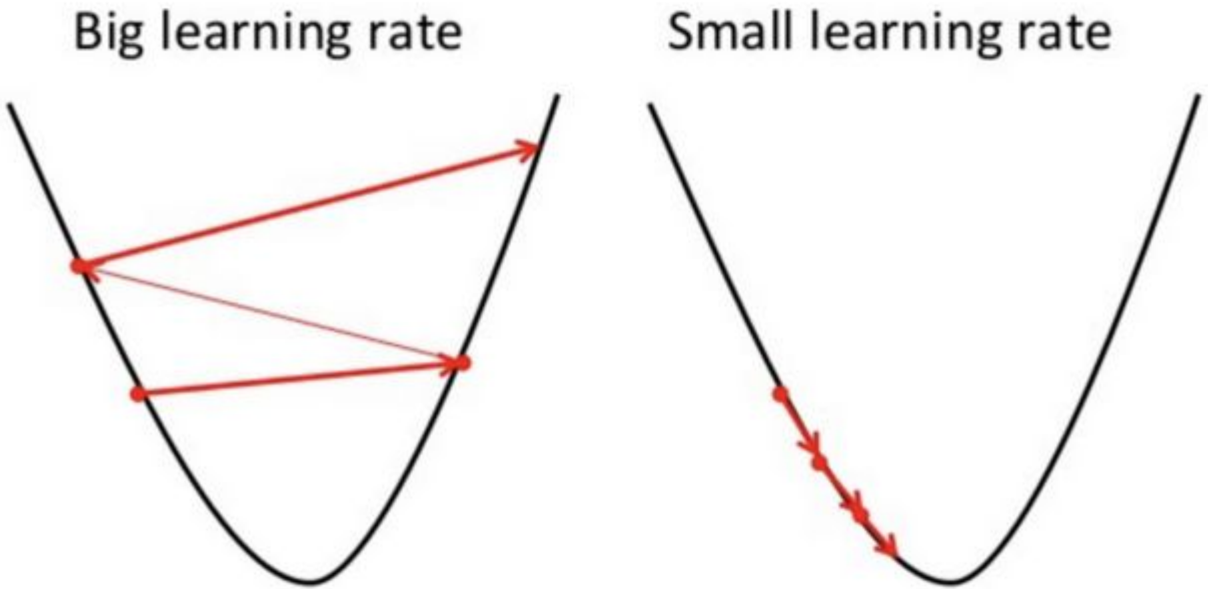
Implementation

Summary of gradient descent

$dz^{[2]} = a^{[2]} - y$	$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = dz^{[2]} a^{[1]^T}$	$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]^T}$
$db^{[2]} = dz^{[2]}$	$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$
$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$	$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$
$dW^{[1]} = dz^{[1]} x^T$	$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$
$db^{[1]} = dz^{[1]}$	$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

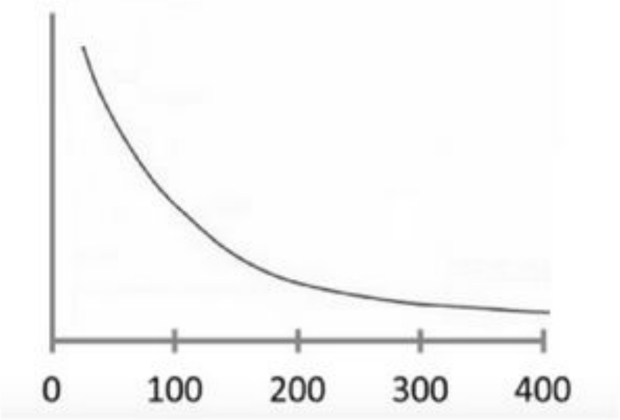
Importance of Learning Rate

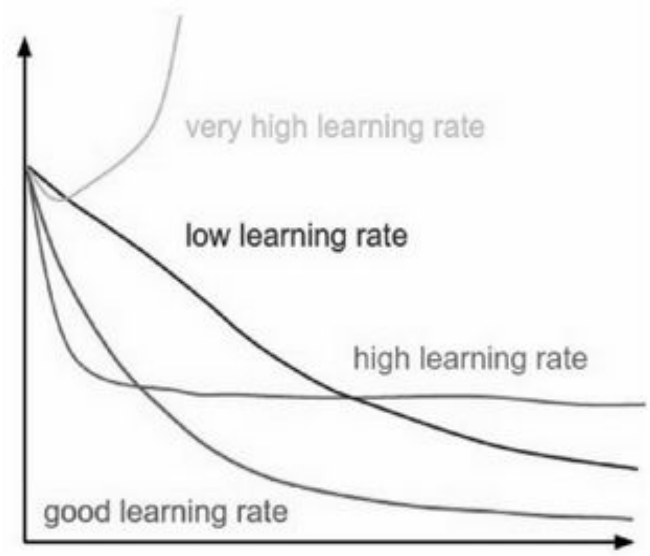
How big the steps are that Gradient Descent takes into the direction of the local minimum are determined by the so-called learning rate. It determines how fast or slow we will move towards the optimal weights.


Big learning rate
Small learning rate

Tips for Gradient Descent

- **Plot Cost versus Time**: Collect and plot the cost values calculated by the algorithm each iteration. The expectation for a well performing gradient descent run is a decrease in cost each iteration. If it does not decrease, try reducing your learning rate.
- **Learning Rate**: The learning rate value is a small real value such as 0.1, 0.001 or 0.0001. Try different values for your problem and see which works best.
- **Rescale Inputs**: The algorithm will reach the minimum cost faster if the shape of the cost function is not skewed and distorted. You can achieved this by rescaling all of the input variables (X) to the same range, such as [0, 1] or [-1, 1].





Number of Iteration Vs Cost

Batch Gradient Descent

- Also known as vanilla gradient descent
- Calculates the error for each example within the training dataset, but only after all training examples have been evaluated, the model gets updated
- computational efficient
- it produces a stable error gradient and a stable convergence
- Stable error gradient may lead to local minima
- It also requires that the entire training dataset is in memory and available to the algorithm
- Use this is training set have approx 2000-3000 input data

Stochastic Gradient Descent

- Calculates the error for each example within the training dataset, and update the model for each example. This means that it updates the parameters for each training example, one by one.
- This can make SGD faster than Batch Gradient Descent, depending on the problem.
- Frequent updates are more computationally expensive.
- The frequency of those updates can also result in noisy gradients, which may cause the error rate to jump around, instead of slowly decreasing.This help in solving the problem of convergence at local minima.

Mini Batch Gradient Descent

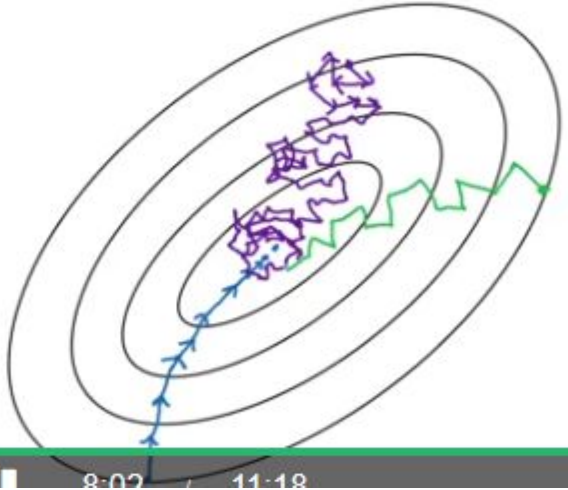
- Mini-batch Gradient Descent is the go-to method since it’s a combination of the concepts of SGD and Batch Gradient Descent.
- It simply splits the training dataset into small batches and performs an update for each of these batches.
- Common mini-batch sizes range between 64, 256, 512

→ If mini-batch size = m : Batch gradient descent. $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$.

→ If mini-batch size = 1 : Stochastic gradient descent. Every example is it own mini-batch.

$(X^{\{1\}}, Y^{\{1\}}) = (x^{(1)}, y^{(1)}) \dots (x^{(2)}, y^{(2)})$ mini-batch.

In practice: Somewhere in-between $\underline{1}$ and $\underline{m}$



Stochastic gradient descent

↓

Lose speedup from vectorization

In-between (mini-batch size not too big/small)

↓

Fastest learning.

- Vectorization. (~1000)
- Make progress without processing entire training set.

Batch gradient descent (mini-batch size = m)

↓

Too long per iteration

Exponential Weighted Average

Temperature in London

$\theta_1 = 40°F$ 4°C ←

$\theta_2 = 49°F$ 9°C

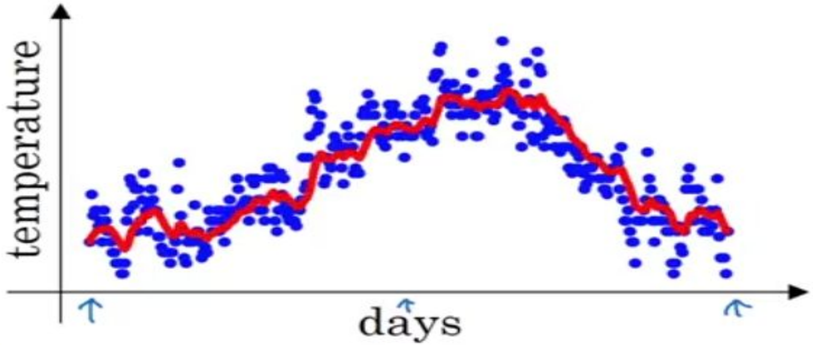
$\theta_3 = 45°F$ ⋮

⋮

$\theta_{180} = 60°F$ 15°C

$\theta_{181} = 56°F$ ⋮

⋮



$V_0 = 0$

$V_1 = 0.9\,V_0 + 0.1\,\theta_1$

$V_2 = 0.9\,V_1 + 0.1\,\theta_2$

$V_3 = 0.9\,V_2 + 0.1\,\theta_3$

⋮

$V_t = 0.9\,V_{t-1} + 0.1\,\theta_t$

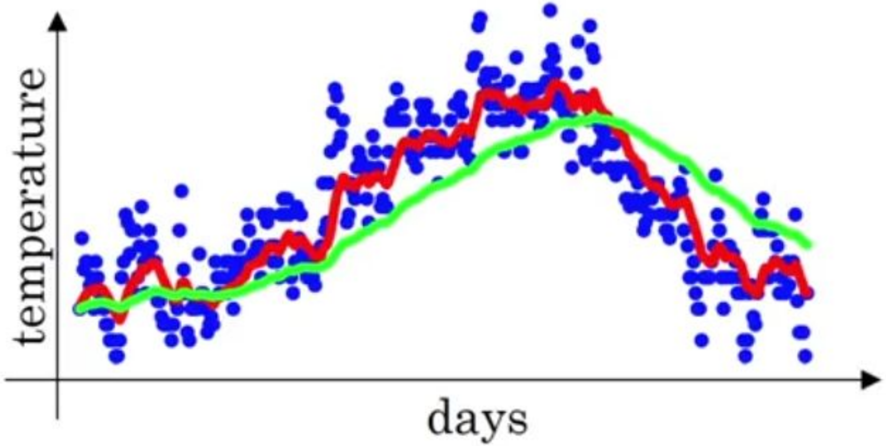
$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

$\beta = 0.9$: $\approx$ 10 days' temperature

$\beta = 0.98$: $\approx$ 50 days

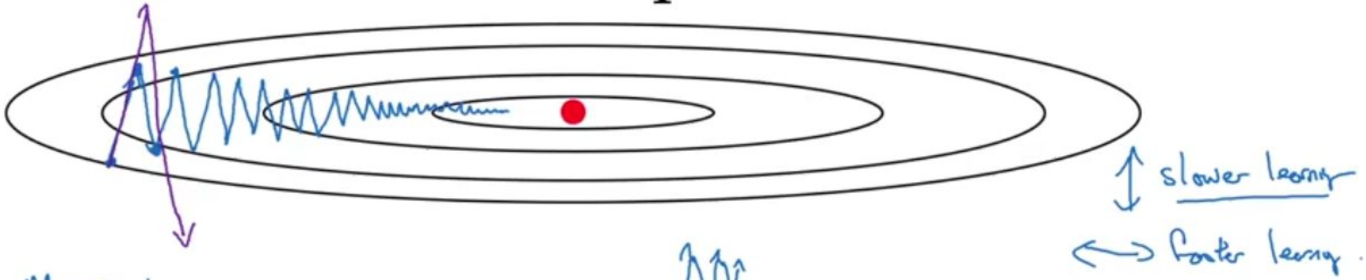
V_t as approximately average over $\approx \frac{1}{1-\beta}$ days' temperature.

$$\frac{1}{1-0.98} = 50$$



Gradient Descent with Momentum

Gradient descent example



Momentum:

On iteration t:

Compute dW, db on current mini-batch.

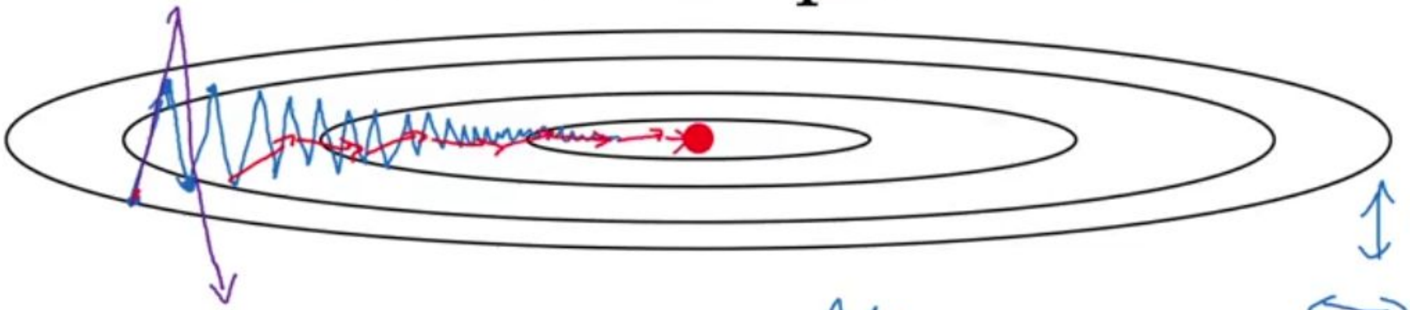
$$V_{dW} = \beta V_{dW} + (1-\beta)\, dW$$

"$V_\theta = \beta V_\theta + (1-\beta)\,\theta_t$"

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

$$W := W - \alpha V_{dW}, \quad b := b - \alpha V_{db}$$

Gradient descent example



$\updownarrow$ slower learning

$\leftrightarrow$ faster learning.

Momentum:

On iteration t:

Compute dW, db on current mini-batch.

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

"$V_\theta = \beta V_\theta + (1-\beta)\,\theta_t$"

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

$$W := W - \alpha V_{dW}, \quad b := b - \alpha V_{db}$$

Implementation details

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

On iteration t:

Compute dW, db on the current mini-batch

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

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

$$W = W - \alpha v_{dW}, \;\; b = \underline{b} - \alpha v_{db}$$

$$\cancel{\frac{V_{dw}}{1-\beta^t}}$$

Hyperparameters: α, β $\qquad \underline{\beta = 0.9}$

Implementation details

$V_{dW} = 0$, $V_{db} = 0$

On iteration t:

Compute dW, db on the current mini-batch

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

$V_{dW} = \beta V_{dW} + \quad dW \leftarrow$

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

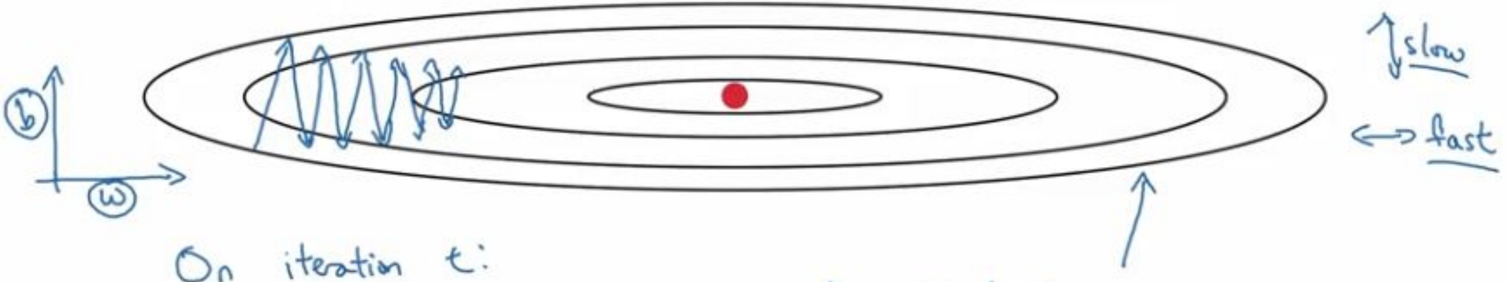
$$W = W - \alpha v_{dW}, \;\; b = b - \alpha v_{db}$$

$\cancel{\frac{V_{dW}}{1-\beta^t}}$

Hyperparameters: α, β $\qquad \beta = 0.9$

RMSprop

RMSprop



On iteration t:

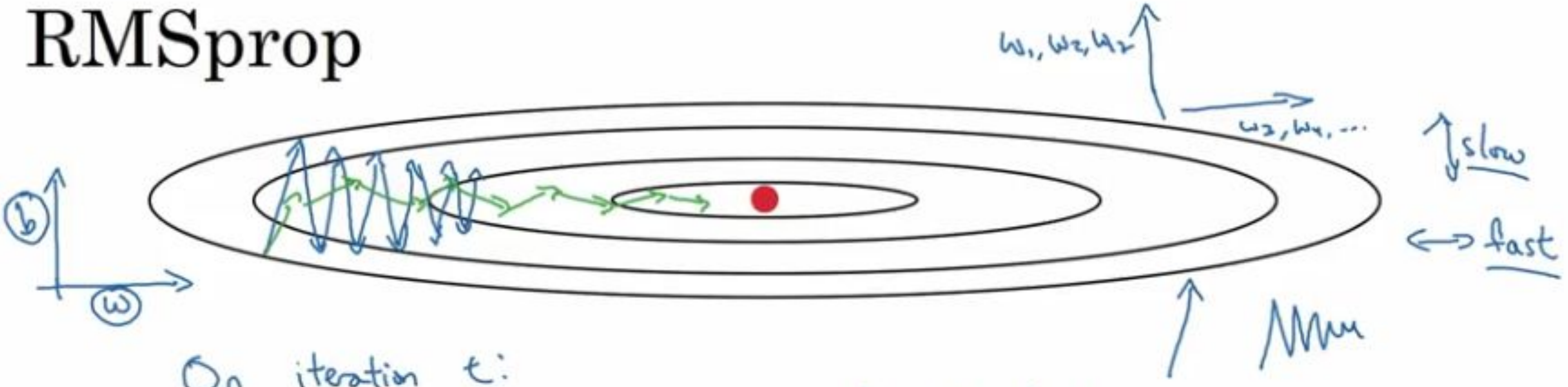
Compute dW, db on current mini-batch

$S_{dW} = \beta S_{dW} + (1-\beta) dW^2$ ← small   element-wise

$S_{db} = \beta S_{db} + (1-\beta) db^2$ ← large

$W := W - \alpha \frac{dW}{\sqrt{S_{dW}}}$ ← $b := b - \alpha \frac{db}{\sqrt{S_{db}}}$ ←

RMSprop



On iteration t:

Compute dW, db on current mini-batch

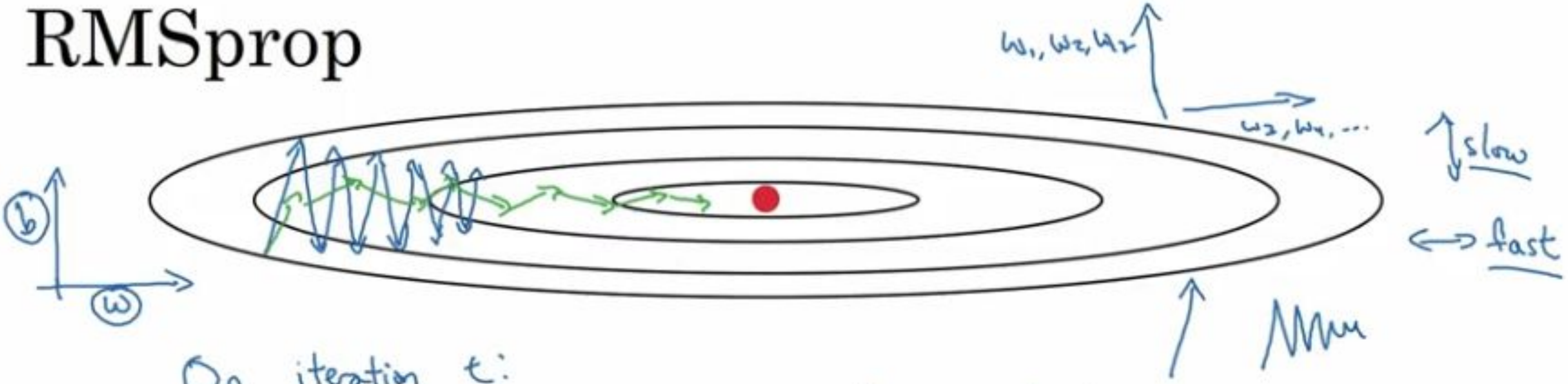
element-wise

$$S_{dW} = \beta S_{dW} + (1-\beta)\, dW^2 \leftarrow \text{small}$$

$$\rightarrow S_{db} = \beta S_{db} + (1-\beta)\, db^2 \leftarrow \text{large}$$

$$W := W - \alpha \frac{dW}{\sqrt{S_{dW}}} \qquad b := b - \alpha \frac{db}{\sqrt{S_{db}}}$$

RMSprop



On iteration t:

Compute dW, db on current mini-batch

element-wise

$$S_{dW} = \beta_2 S_{dW} + (1-\beta_2)\, dW^2 \leftarrow \text{small}$$

$$\rightarrow S_{db} = \beta_2 S_{db} + (1-\beta_2)\, db^2 \leftarrow \text{large}$$

$$W := W - \alpha \frac{dW}{\sqrt{S_{dW}+\varepsilon}} \qquad b := b - \alpha \frac{db}{\sqrt{S_{db}+\varepsilon}}$$

$\varepsilon = 10^{-8}$

Adam optimization algorithm

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

On iteration t:

Compute dW, db using current mini-batch

$V_{dw}=\beta_1 V_{dw}+(1-\beta_1)dW$, $V_{db}=\beta_1 V_{db}+(1-\beta_1)db$ $\leftarrow$ "momentum" β_1

$S_{dw}=\beta_2 S_{dw}+(1-\beta_2)dW^2$, $S_{db}=\beta_2 S_{db}+(1-\beta_2)db$ $\leftarrow$ "RMSprop" β_2

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

$S_{dw}^{corrected}=S_{dw}/(1-\beta_2^t)$, $S_{db}^{corrected}=S_{db}/(1-\beta_2^t)$

$$W:=W-\alpha\frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}}+\varepsilon} \qquad b:=b-\alpha\frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}}+\varepsilon}$$

Hyperparameters choice:

→ α : needs to be tune

→ β_1 : 0.9 → $(\underline{dw})$

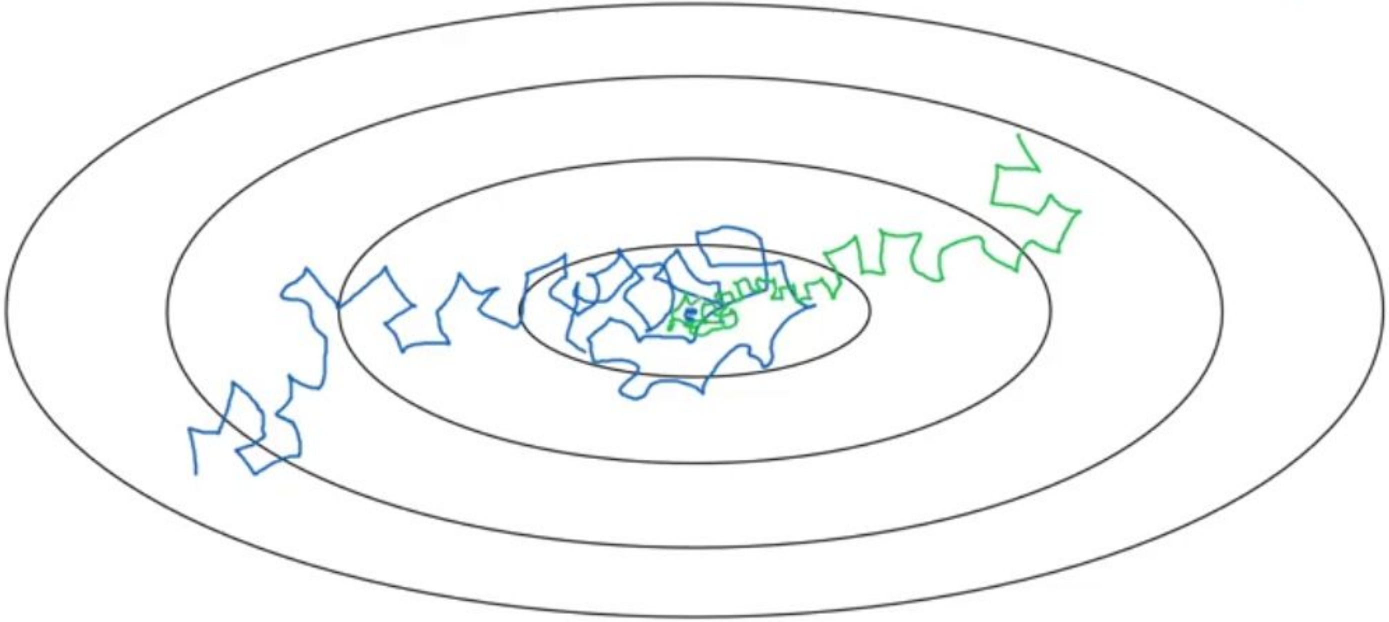
→ β_2 : 0.999 → $(\underline{dw^2})$

→ ε : 10^{-8}

Adam: Adaptive moment estimation

Learning Rate Decay

Slowly reduce α



Learning rate decay

1 epoch = 1 pass through data.

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

$\alpha_0 = 0.2$

decay-rate = 1

Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
⋮	⋮

Other learning rate decay methods

formula

$$\alpha = 0.95^{\text{epoch-num}} \cdot \alpha_0$$

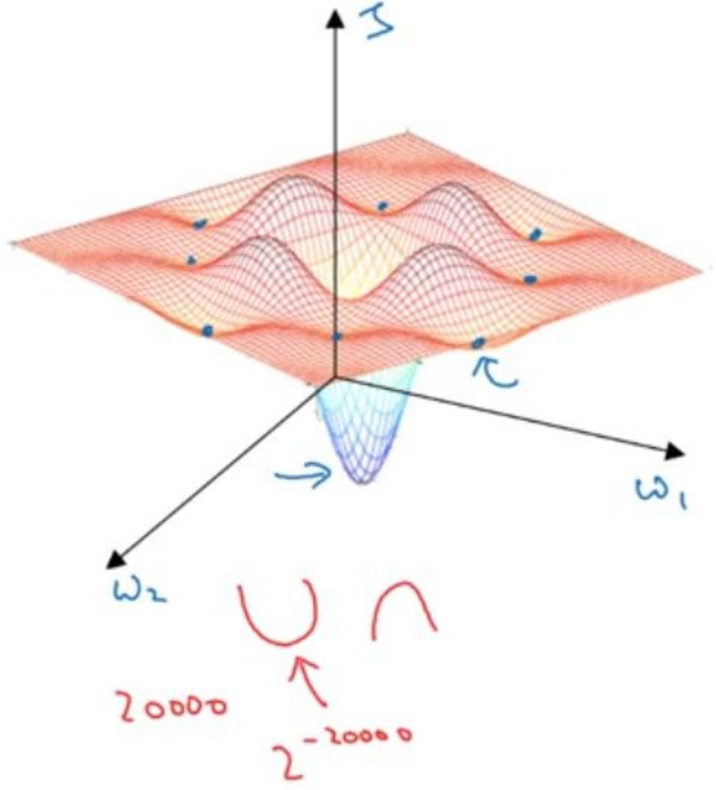
– exponentially decay.

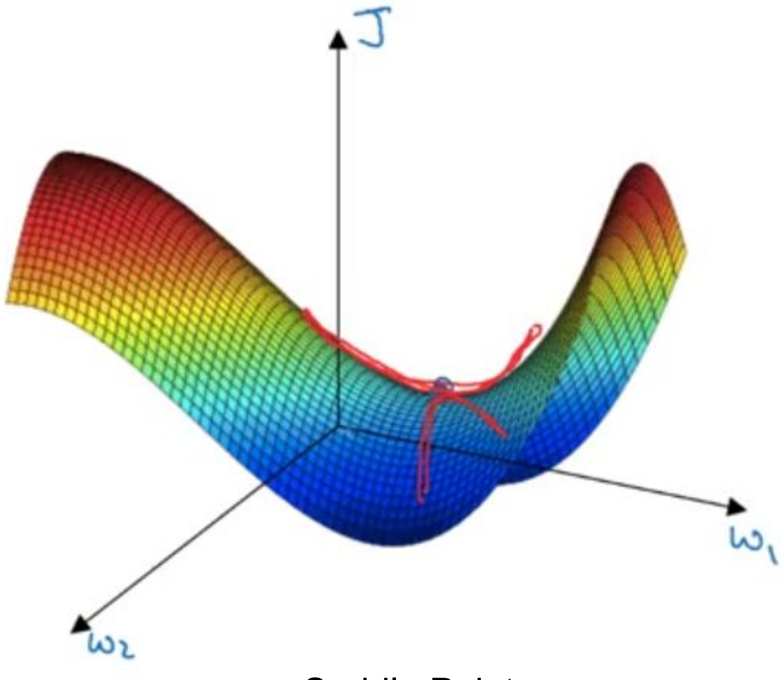
$$\alpha = \frac{k}{\sqrt{\text{epoch-num}}} \cdot \alpha_0 \quad \text{or} \quad \frac{k}{\sqrt{t}} \cdot \alpha_0$$

discrete staircase

Manual decay.

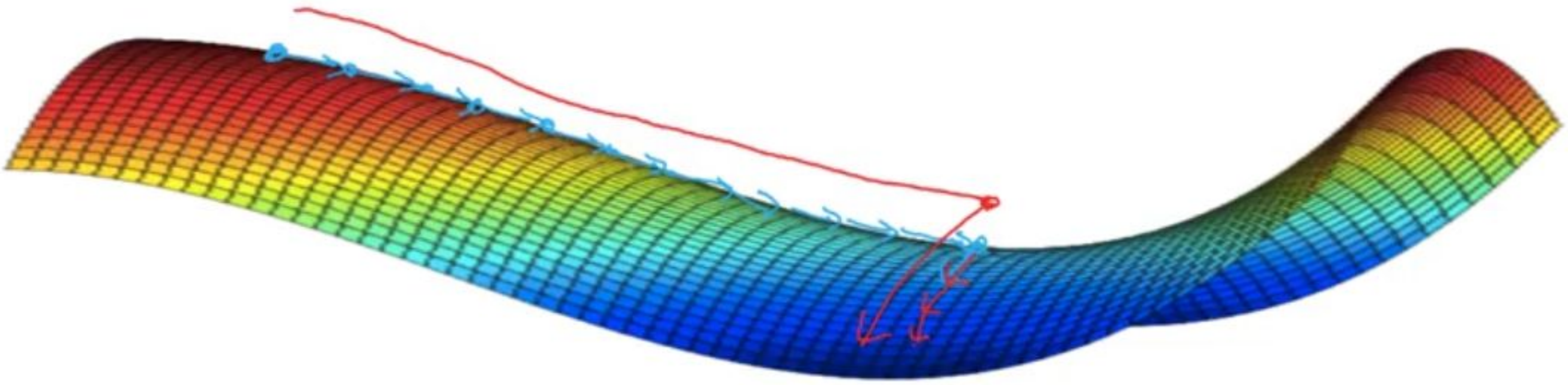
Local optima in neural networks





Saddle Point

Problem of plateaus



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

Reference

https://machinelearningmastery.com/gradient-descent-for-machine-learning/

https://www.analyticsvidhya.com/blog/2017/03/introduction-to-gradient-descent-algorithm-along-its-variants/

http://neuralnetworksanddeeplearning.com/chap1.html

https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0

https://www.coursera.org/learn/deep-neural-network/home/week/2