Optimization algorithms in Deep Learning

Robert Maria

21.11.2019

Contents

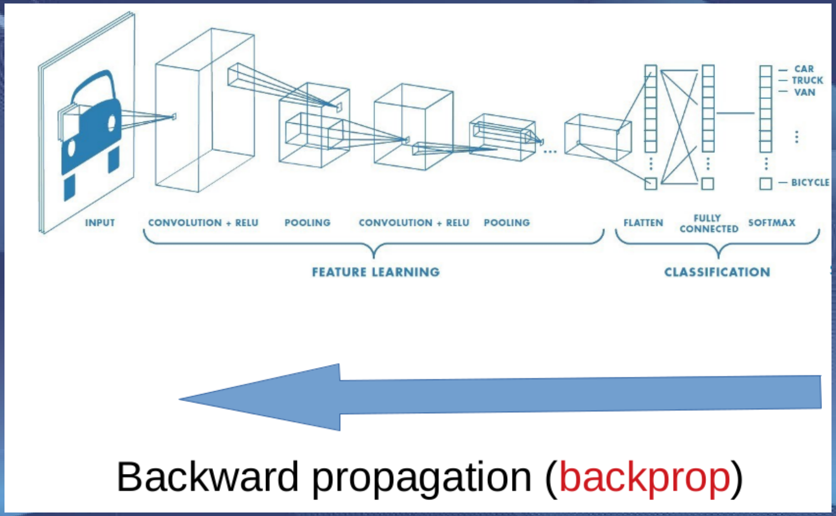
- Optimization algorithms vs backpropagation
- First order methods
- Second order methods

Optimization algorithms vs backpropagation

Backpropagation is an efficient method of computing gradients of the loss function wrt parameters in directed graphs of computations, such as neural networks.

Optimization algorithms vs backpropagation

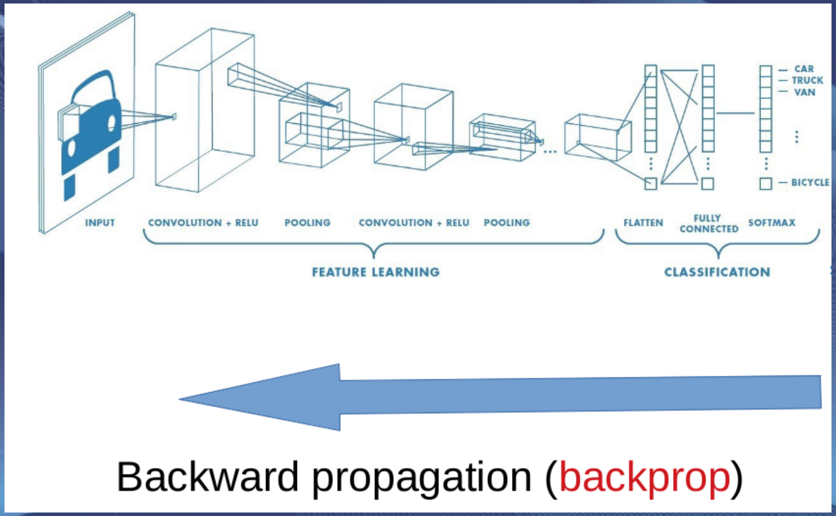
Backpropagation is an efficient method of computing gradients of the loss function wrt parameters in directed graphs of computations, such as neural networks.



Optimization algorithms vs backpropagation

Backpropagation is an efficient method of computing gradients of the loss function wrt parameters in directed graphs of computations, such as neural networks.

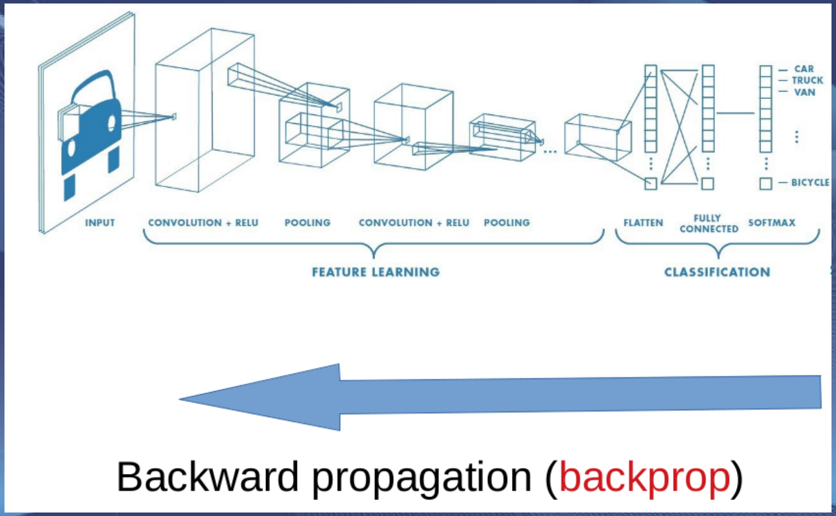
An **optimization algorithm** is used to find parameters that minimize the loss function.

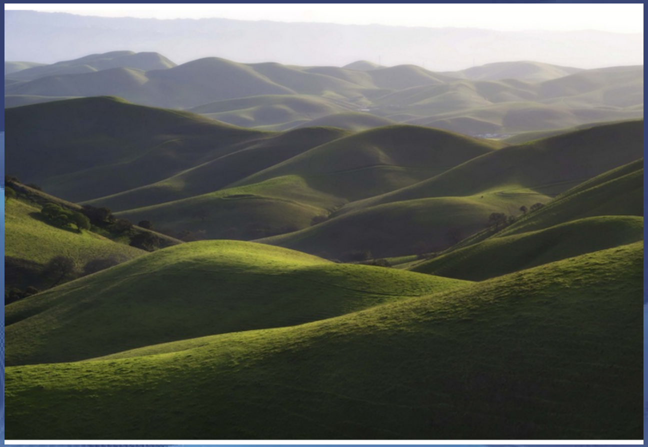


Optimization algorithms vs backpropagation

Backpropagation is an efficient method of computing gradients of the loss function w.r.t. parameters in directed graphs of computations, such as neural networks.

An **optimization algorithm** is used to find parameters that minimize the loss function.





A grassy continuous non-convex function.
source

Types of optimization algorithms

1^{st} order methods:	2^{nd} order methods:
Uses the first derivative information (compute the Jacobian)	Uses the second derivative information (compute the Hessian) Infeasible to compute in practice for high-dimensional datasets

Gradient Descent

The most popular method to optimize neural networks.

Gradient Descent

The most popular method to optimize neural networks.

A way to minimize an objective function $J(\theta)$ parametrized by the model's parameters $\theta \in R^d$ by updating the parameters in the opposite direction of the gradient of the objective function w.r.t. the parameters.

Gradient Descent

The most popular method to optimize neural networks.

A way to minimize an objective function $J(\theta)$ parametrized by the model's parameters $\theta \in R^d$ by updating the parameters in the opposite direction of the gradient of the objective function w.r.t. the parameters.

The learning rate η determines the size of the steps we take to reach a (local) minimum.

Batch (vanilla) Gradient Descent

Computes the gradients of the objective function J(θ) w.r.t. the parameters θ of the entire training set:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

code:

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

source

Drawbacks:

- uses the entire training set
- can perform redundant computations

Stochastic Gradient Descent

Computes the gradients of the objective function J(θ) w.r.t. the parameters θ **of 1 training sample at a time**:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

$x^{(i)}$ training sample

$y^{(i)}$ label

η learning rate

code:

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

Stochastic Gradient Descent

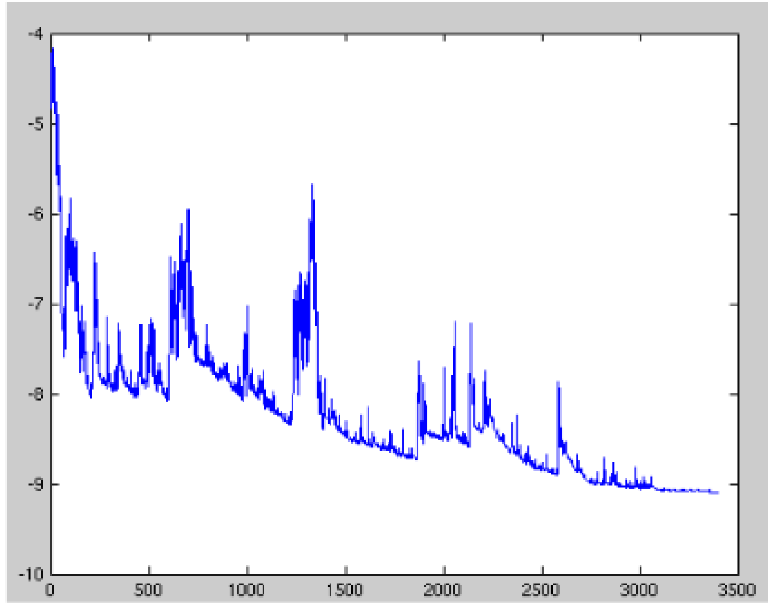
Computes the gradients of the objective function J(θ) w.r.t. the parameters θ **of 1 training sample at a time**:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

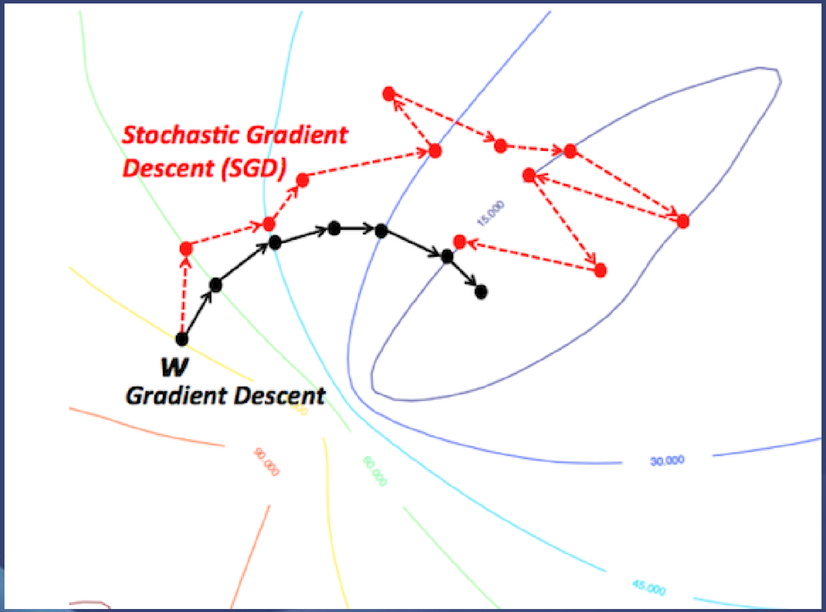
Drawbacks:

- the objective function have fluctuations
- complicate convergence to the exact minimum (overshooting)

Stochastic Gradient Descent



SGD fluctuations source



source

Mini-batch Gradient Descent (SGD)

Computes the gradients of the objective function J(θ) w.r.t. the parameters θ **of 1 mini-batch at a time**:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

code:

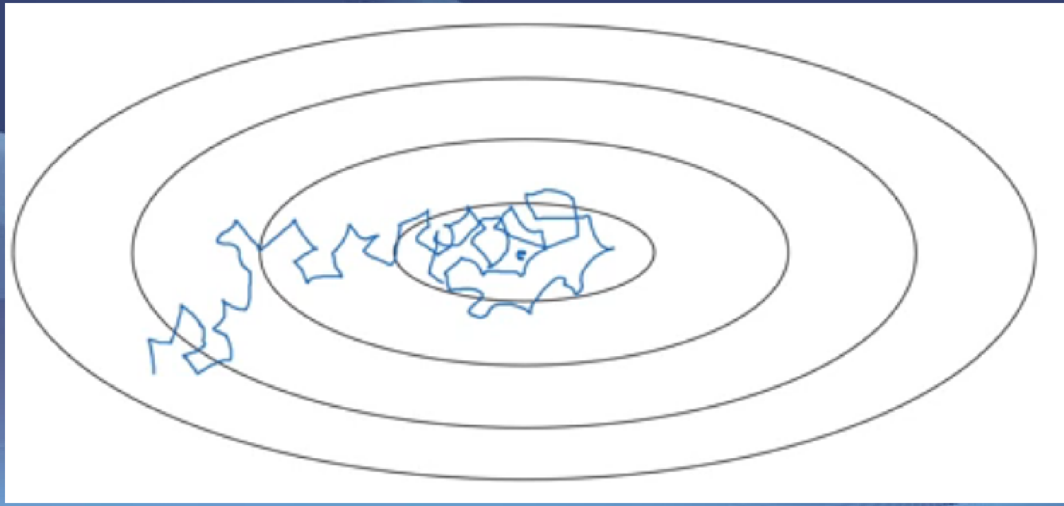
```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Pros:

- reduces the variance of the parameter update
- can make use of high-optimized matrix optimization libraries

Challenges

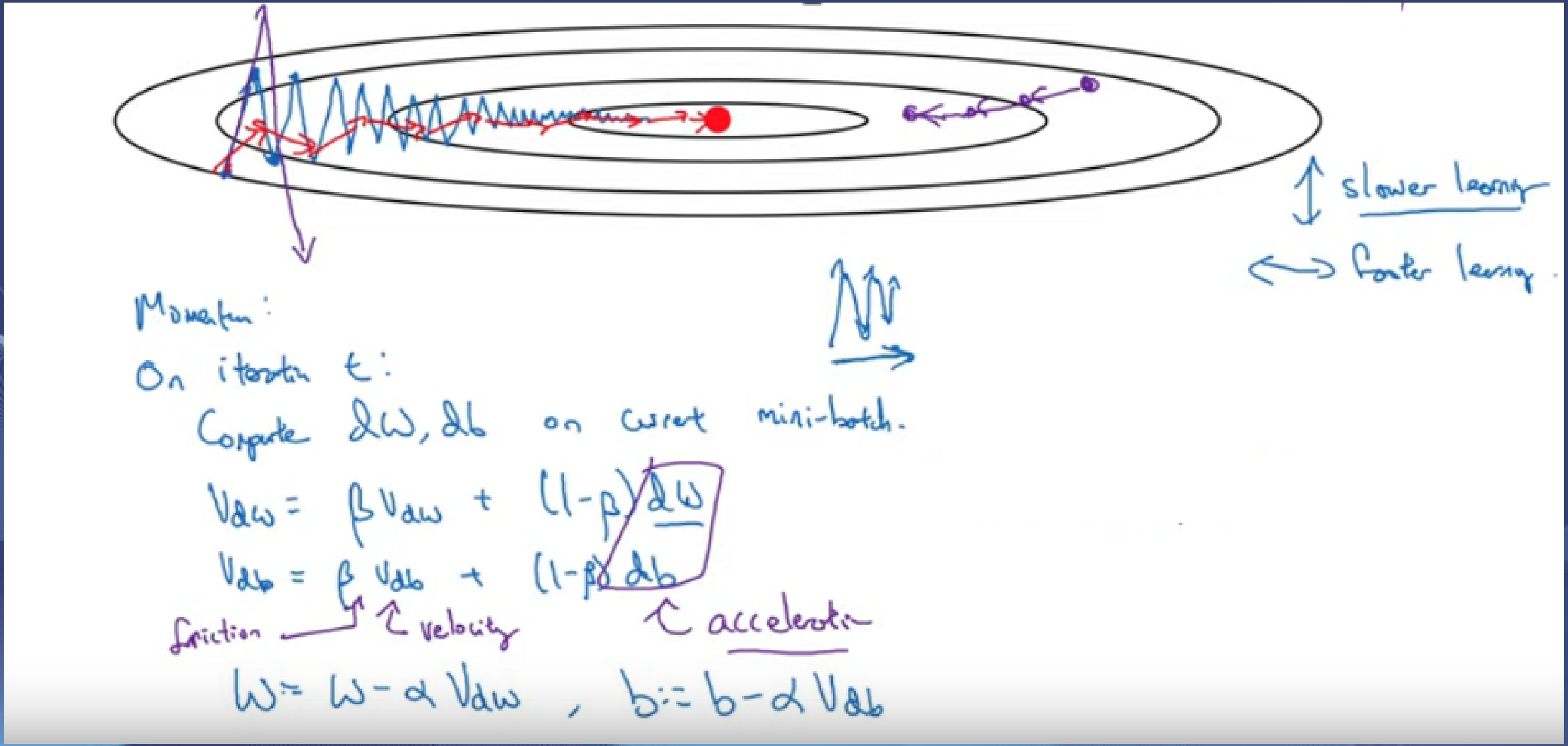
- choosing a proper learning rate → define a learning rate schedule
- schedules and thresholds for learning rate schedules
- getting trapped in sub-optimal local minima (saddle points, plateau)
- wandering around and never converge



source

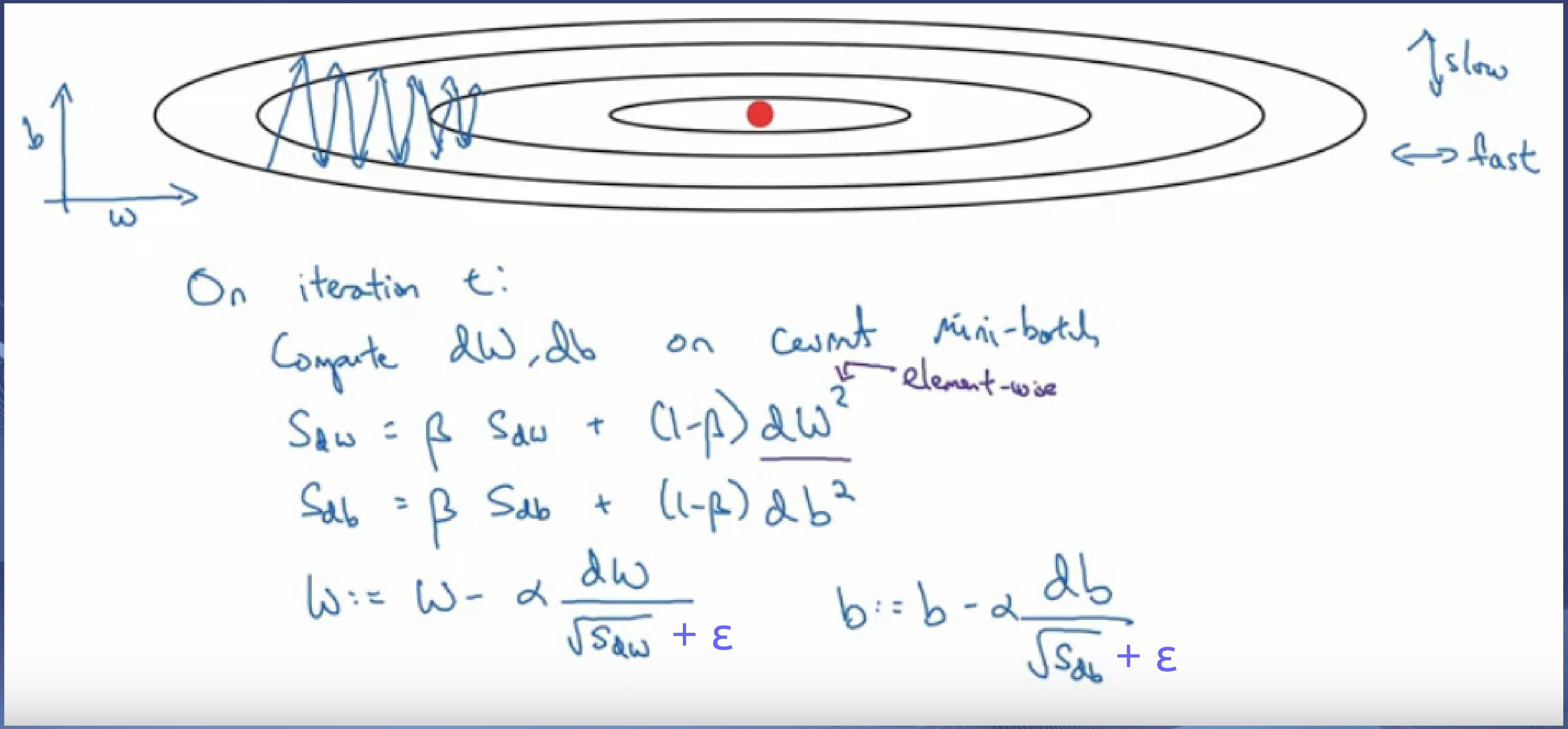
How to overcome these challenges

Momentum



How to overcome these challenges

RMSProp (Root Mean Square Prop)



source

- dW^2 is small and db^2 is large

How to overcome these challenges

ADAM (Adaptive Moment Estimation)

= momentum + RMSProp

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

On iteration t:

Compute dW, db using current mini-batch

$$\boxed{V_{dw} = \beta_1 V_{dw} + (1-\beta_1)\,dW, \quad V_{db} = \beta_1 V_{db} + (1-\beta_1)\,db}$$ momentum

$$\boxed{S_{dw} = \beta_2 S_{dw} + (1-\beta_2)\,dW^2, \quad S_{db} = \beta_2 S_{db} + (1-\beta_2)\,db}$$ RMSProp

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

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

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

source

How to overcome these challenges

ADAM (Adaptive Moment Estimation)

→ α : needs to be tune

→ β_1 : 0.9 (dw)

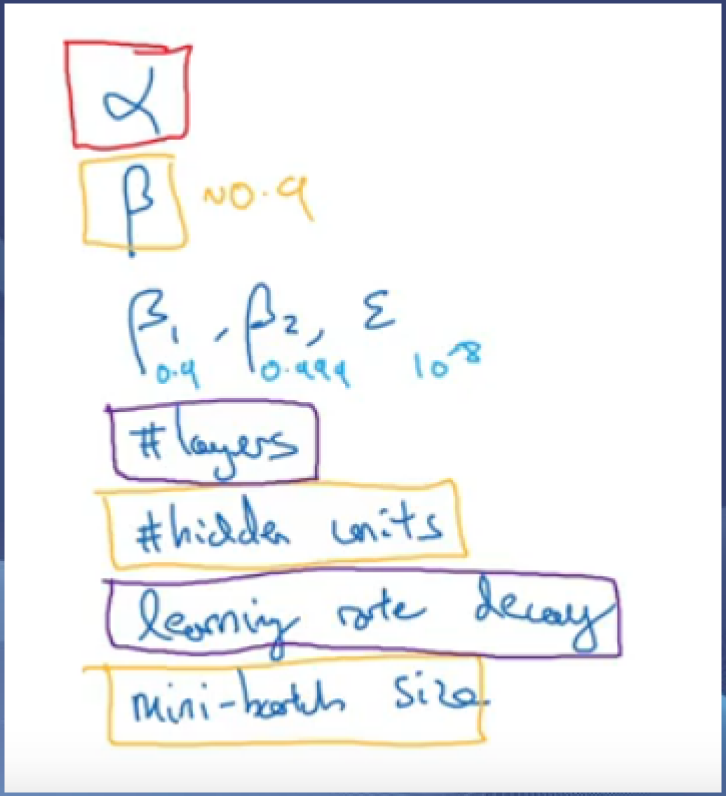
→ β_2 : 0.999 (dw^2)

→ ε : 10^{-8}

source

Hyperparameters to tune

According to Andrew Ng...



source

Red = most important
Orange = second importance
Blue = third importance

Thank you!