HyperParams DNN

{ وَاتَّقُوا اللَّهَ وَيُعَلِّمُكُمُ اللَّهُ وَاللَّهُ بِكُلِّ شَيْءٍ عَلِيمٌ }

👨🏽‍💻 **github**: elma-dev

🔗 **linkden**: EL MAJJODI Abdeljalil

Params Initialization:

- Different initializations lead to very different results
- Random initialization is used to break symmetry and make sure different hidden units can learn different things
- Resist initializing to values that are too large!
- He initialization works well for networks with ReLU activations

He Initialization / glorot:

```
np.random.randn(,)*np.sqrt(2/len(l-1))
```

Regularization:

→ Deep Learning models have so much flexibility and capacity that **overfitting can be a serious problem**, if the training dataset is not big enough. Sure it does well on the training set, but the learned network **doesn't generalize to new examples** that it has never seen!

L2 Regularization

The standard way to avoid overfitting is called **L2 regularization**. It consists of appropriately modifying your cost function, from:

$$J = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right) \tag{1}$$

To:

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}} \tag{2}$$

```
L2_regularization_cost=(np.sum(np.square(W1))+np.sum(np.square(W2))+np.sum(np.s
```

We should to Add :

Implement the changes needed in backward propagation to take into account regularization. The changes only concern dW1, dW2 and dW3. For each, you have to add the regularization term's gradient ($\frac{d}{dW}(\frac{1}{2}\frac{\lambda}{m}W^2) = \frac{\lambda}{m}W$).

What you should remember:

the implications of L2-regularization on:

- The cost computation:
  - A regularization term is added to the cost.
- The back-propagation function:
  - There are extra terms in the gradients with respect to weight matrices.
- Weights end up smaller ("weight decay"):
  - Weights are pushed to smaller values.

Drop-out Technic :

→ **dropout** is a widely used regularization technique that is specific to deep learning. **It randomly shuts down some neurons in each iteration.**

→ When you shut some neurons down, you actually modify your model. The idea behind drop-out is that at each iteration, you train a different model that uses only a subset of your neurons. With dropout, your neurons thus become less sensitive to the activation of one other specific neuron, because that other neuron might be shut down at any time.

- We will not apply dropout to the input layer or output layer.

Divide $A[1]$ by keep_prob . By doing this you are assuring that the result of the cost will still have the same expected value as without drop-out

- A **common mistake** when using dropout is to use it both in training and testing. You should use dropout (randomly eliminate nodes) only in training.

What you should remember about dropout:

- Dropout is a regularization technique.
- You only use dropout during training. Don't use dropout (randomly eliminate nodes) during test time.
- Apply dropout both during forward and backward propagation.
- During training time, divide each dropout layer by keep_prob to keep the same expected value for the activations. For example, if keep_prob is 0.5, then we will on average shut down half the nodes, so the output will be scaled by 0.5 since only the remaining half are contributing to the solution. Dividing by 0.5 is equivalent to multiplying by 2. Hence, the output now has the same expected value. You can check that this works even when keep_prob is other values than 0.5.

Notes:

- Regularization will help you reduce overfitting.
- Regularization will drive your weights to lower values.
- L2 regularization and Dropout are two very effective regularization techniques.

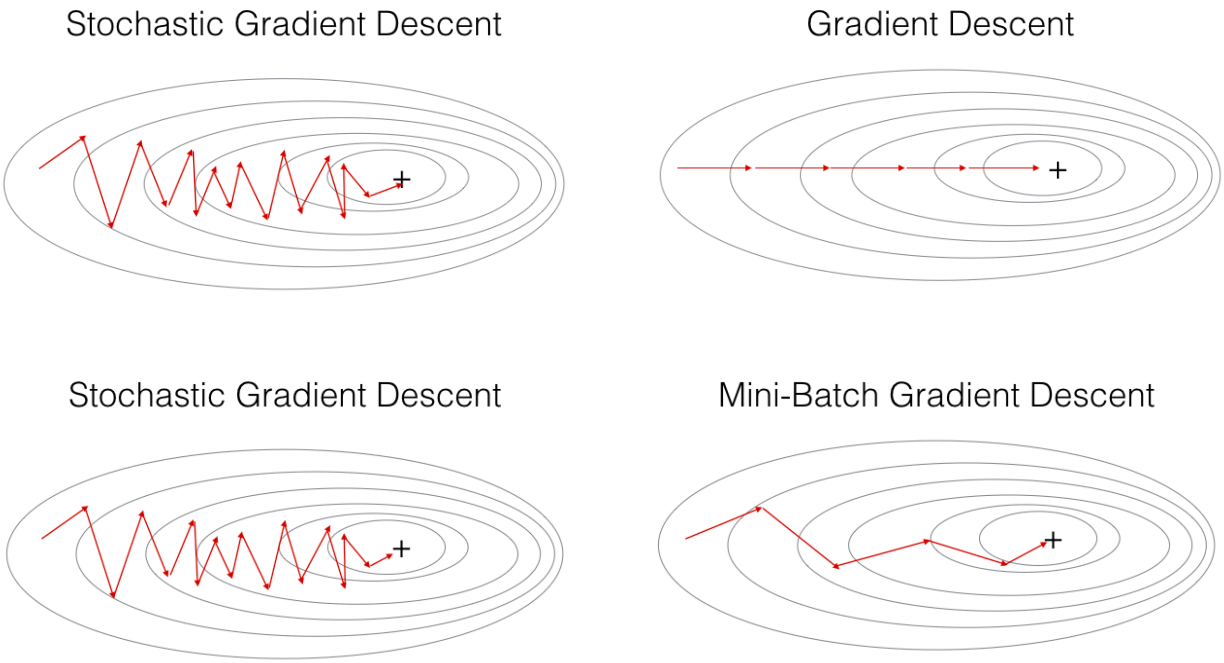
Gradient Checking

you can check if the gradient descent values is correct using gradient checking technic, this tech is very easy because we just use definition of gradient .

Optimization Algorithm

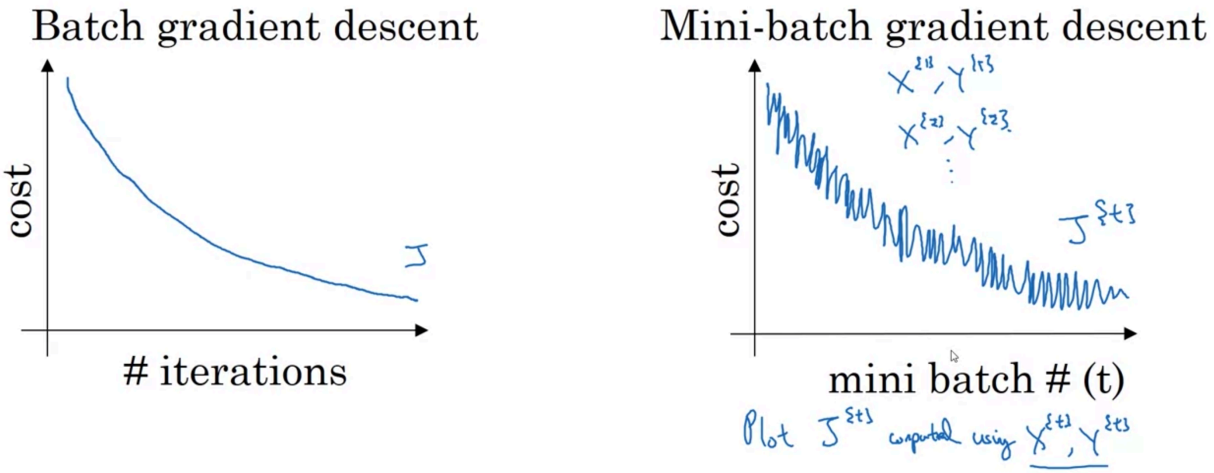
- Training NN with a large data is slow. So to find an optimization algorithm that runs faster is a good idea.
- Suppose we have m = 50 million. To train this data it will take a huge processing time for one step.
    - because 50 million won't fit in the memory at once we need other processing to make such a thing.

Mini-Batch Gradient Descent



→ **Batch gradient descent** we run the gradient descent on the whole dataset.

→ in **Mini-Batch gradient** descent we run the gradient descent on the mini dataset.




```
for b in range(numberOfBatches):
    AL,caches=forwardProp(X{b})
```

```
    cost=cost(AL,Y{b})
    grads=backProp(AL,caches)
    updateParams(grads)
```

→ Mini-batch gradient descent works much faster in the large datasets.

Chose Mini-Batch Size

$$miniBatchSize = m \Rightarrow BatchGradientDescent$$

- Batch Gradient Descent
  - too long per iteration (epoch)

$$miniBatchSize = 1 \Rightarrow StochasticGradientDescent$$

- Stochastic Gradient Descent
  - too noisy regarding cost minimization
  - won't ever converge

$$1 < miniBatchSize < m \Rightarrow Mini - BatchGradientDescent$$

- Mini Batch Gradient Descent
  - fast
  - not converge always

💡 ⇒ **Then What I will Chose !!**

- if TrainingSet small (<2000 examples) ⇒ Use Batch Gradient Descent
- else ⇒ Use Mini-Batch Gradient Descent
  - it has to be a power of 2 → (because of the way computer memory is layed out and accessed, sometimes your code runs faster if your mini-batch size is a power of 2): $64, 128, 256, 512, 1024, \ldots$

Exponentially weighted averages

→ There are a lot of optimization algorithms better than **gradient descent** but firstly we should learn about **exponentially weighted averages.**

EWA Equation

$$V(t) = beta * v(t-1) + (1 - beta) * theta(t)$$

Examples

→ if we have weather data of London like this:

theta_i
40
49
45
...
60
..

- If we plot this it will represent averages over $(1/(1-beta))$ entries:
  - beta = 0.9 will average last 10 entries
  - beta = 0.98 will average last 50 entries
  - beta = 0.5 will average last 2 entries

- theta small in winter and big in summer.
- if we ploting this data we get some noisy

⇒ **Calcuation of V(t):**

V0 = 0 V1 = 0.9 * V0 + 0.1 * t(1) = 4
# 0.9 and 0.1 are hyperparameters
V2 = 0.9 * V1 + 0.1 * t(2) = 8.5
V3 = 0.9 * V2 + 0.1 * t(3) = 12.15 ...

Bias correction in exponentially weighted averages

→ The bias correction helps make the exponentially weighted averages more accurate.

$$v(t) = (beta * v(t-1) + (1 - beta) * theta(t))/(1 - beta^t)$$

Gradient Descent With Momentum

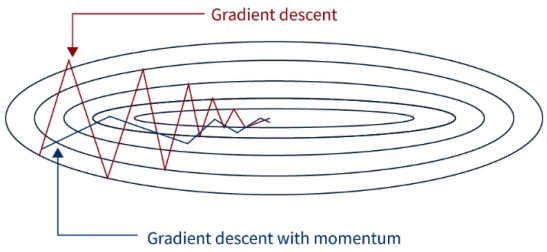
→ Momentum algorithm work faster than Gradient Descent.

→ Momentum is just a Gradient Descent algorithm with a little modifications.

→ With Momentum we calculate the Exponentially weighted of gradients and then update Weights with the new values.

```
#Pseudo Code
Vdw, Vdb=0,0
for i in range(iter):
    computeGradient_of_dw_db()
    Vdw=beta*Vdw+(1-beta)*dW
    Vdb=beta*Vdb+(1-beta)*db
    W=W-alpha*Vdw
    b=b-alpha*Vdb
```

→ With Momentum the Cost function going fast to the minimum.

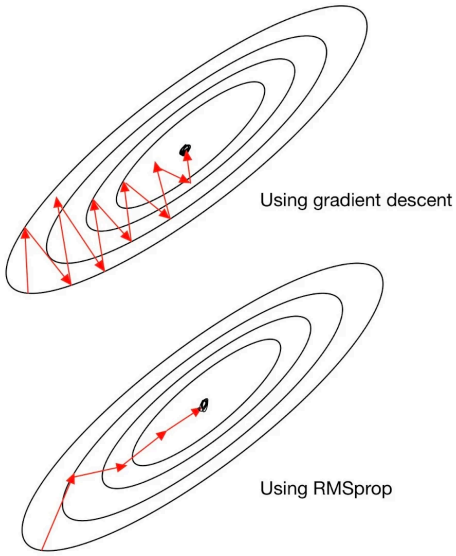


→ beta also is a hyperparametre but the common used value is 0.9

RMSprop

→ Also RMSprop speeds up the Gradient Descent.

→ This Algo make the cost function move **slower in vertical direction (b-axe)** and **fast in horizontal direction (w-axe)** .




```
sdW,sdb=0,0
for i in range(iterations):
    dW,db=compute_dw_db()
    sdW = beta * sdW + (1-beta) * dW^2 #^2 : element wise
    sdb = beta * sdb + (1-beta) * db^2
    W = W - alpha * (dw / sqrt(sdW + epsilon)) # add epsilon just to avoid zero
    b = b - alpha * (db / sqrt(sdb + epsilon))
```

→ Developed by Geoffrey Hinton and firstly introduced on Coursera.org course.

Adam Optimization AlgorithmNotes

→ Adam algorithm combine between **Momentum** and **RMSprop** .

```
vdW = 0, vdW = 0
sdW = 0, sdb = 0
on iteration t:
    # can be mini-batch or batch gradient descent
    compute dw, db on current mini-batch
```

```
vdW = (beta1 * vdW) + (1 - beta1) * dW # momentum
vdb = (beta1 * vdb) + (1 - beta1) * db # momentum
sdW = (beta2 * sdW) + (1 - beta2) * dW^2 # RMSprop
sdb = (beta2 * sdb) + (1 - beta2) * db^2 # RMSprop
vdW = vdW / (1 - beta1^t) # fixing bias
vdb = vdb / (1 - beta1^t) # fixing bias
sdW = sdW / (1 - beta2^t) # fixing bias
sdb = sdb / (1 - beta2^t) # fixing bias
#Update Params
W = W - learning_rate * vdW / (sqrt(sdW) + epsilon)
b = b - learning_rate * vdb / (sqrt(sdb) + epsilon)
```

- Hyperparameters for Adam:
  - Learning rate: needed to be tuned.
  - beta1 : parameter of the momentum - **0.9** is recommended by default.
  - beta2 : parameter of the RMSprop - **0.999** is recommended by default.
  - epsilon : **10^-8** is recommended by default.

Learning Rate Decay

→ It is a method to modify the learning rate alpha epoch after epoch.

→ As mentioned before mini-batch gradient descent won't reach the optimum point (converge). But by making the learning rate decay with iterations it will be much closer to it because the steps (and possible oscillations) near the optimum are smaller.

$$alpha = learning\_rate = (1/(1 + decay\_rate * epoch\_num)) * learning\_rate\_0$$

→ For **Andrew Ng,** learning rate decay has less priority.

Hyperparameter tuning

→ To get the best results we should to find the good hyperparameters.

→More important hyperparameters is:(sorted by importance)

i. Learning rate.
ii. Momentum beta.
iii. Mini-batch size.

iv. No. of hidden units.
v. No. of layers.
vi. Learning rate decay.
vii. Regularization lambda.
viii. Activation functions.
ix. Adam beta1 & beta2 .

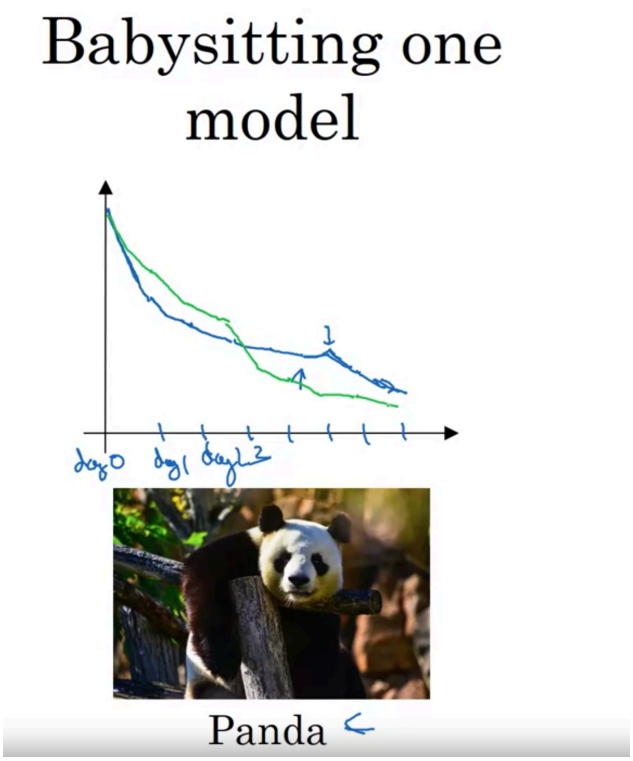
- It’s hard to decide which hyperparameter is the most important in a problem. It depends a lot on your problem.
- One of the ways to tune is to sample a grid with N hyperparameter settings and then try all settings combinations on your problem.
- Try random values: don't use a grid.
- You can use Coarse to fine sampling scheme :
    - When you find some hyperparameters values that give you a better performance - zoom into a smaller region around these values and sample more densely within this space.

Using an appropriate scale to pick hyperparameters

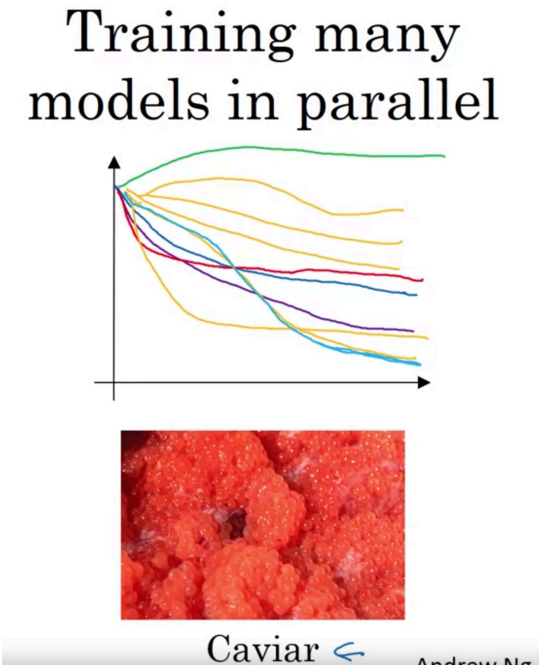
- When you have a range for a hyperparamter [a,b] it’s better to search using logarithmic scale:
    - logarithmic scale : [lag(a),log(b)]

⇒ When We don’t have a lot of calculations resources we can use tow methods to get great results, and this methods call:

⇒ **Babysittig One Model**: in this method we try to train one model and day after day we change the parameters.



⇒ **Training Many Models in Partallel**: with this method we train a lot of models and chose the best one.



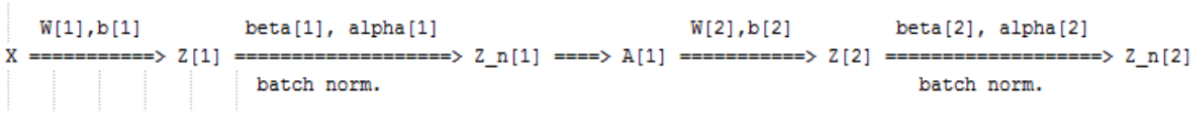
Normalizing activations in a network

- Batch Normalization is one of the important ideas in DL.
- With this algo we can speed up learning.
- Before we normalized input by subtracting the mean and dividing by variance. This helped a lot for the shape of the cost function and for reaching the minimum point faster.
- There are some debates in the deep learning literature about whether you should normalize values **before** the *activation function Z[l]* or **after** applying the *activation function A[l]* . In practice, **normalizing Z[l] is done much more** often and that is what Andrew Ng presents.

```
#Pseudo Code:
    Given Z[l] = [z(1), ..., z(m)] , i = 1 to m (for each input)
    Compute mean = 1/m * sum(z[i])
    Compute variance = 1/m * sum((z[i] - mean)^2)
    Then Z_norm[i] = (z[i] - mean) / np.sqrt(variance + epsilon)
 #(add epsilon for numerical stability if variance =0)
#Forcing the inputs to a distribution with zero mean and variance of 1.
    Then Z_tilde[i] = gamma * Z_norm[i] + beta
    #To make inputs belong to other distribution (with other mean and variance).
    #gamma and beta are learnable parameters of the model.
    #Making the NN learn the distribution of the outputs.
Note: if gamma = sqrt(variance + epsilon) and beta = mean then Z_tilde[i] = z[i]
```

Example:

⇒ Using batch norm in 3 hidden layers NN:



- Our NN parameters will be:
  - **W[1] , b[1] , ..., W[L] , b[L] , beta[1] , gamma[1] , ..., beta[L] , gamma[L]**
    beta[1] , gamma[1] , ..., beta[L] , gamma[L]
    are updated using any optimization algorithms (like GD, RMSprop, Adam)