Deep Learning Notes

By Divyajeet Pala

Logistic Regression

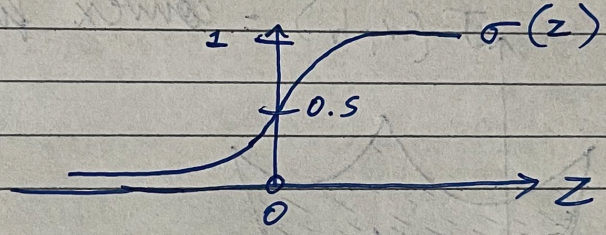
→ Input : x , $x \in \mathbb{R}^{n_x}$ i.e., x is an n_x dimensional vector

Parameters : $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$
(n_x dim. vector) (a real number)

Output : $\hat{y} = P(y=1 \mid x)$ i.e., probability $y=1$ given x

→ $\hat{y} = \sigma(w^T x + b)$

→ Sigmoid function



→ $\sigma(z) = \frac{1}{1+e^{-z}}$

→ Training examples : $\{(x^{(1)}, y^{(1)}), \ldots (x^{(m)}, y^{(m)})\}$,
want $\hat{y}^{(i)} \approx y^{(i)}$.

Note // Superscript in rounded bracket (i) denotes the i^{th} training example out of the total m examples.

→ Loss function : $\mathcal{L}(\hat{y}, y) = -[y \log \hat{y} + (1-y) \log(1-\hat{y})]$
(convex func.)

→ measures how well you're doing on a single training example

Teachers Sign

→ Cost function (measures performance w.r.t. whole dataset):

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

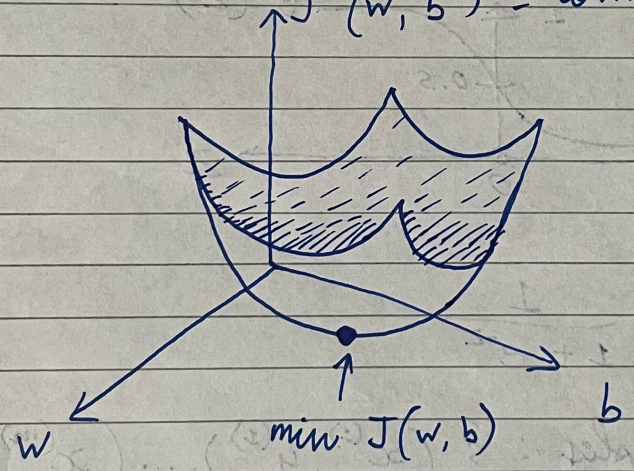
Note the -ve sign ↑

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

→ Gradient descent

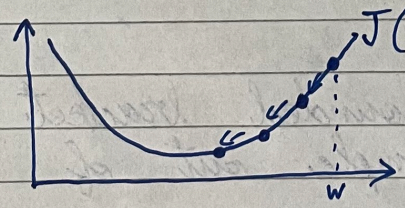
→ ~~Wannt~~ Want to find w, b that minimize $J(w, b)$

→ $J(w, b)$ = convex function



3-D representation

→



Repeat { ↓ learning rate

$w := w - \alpha \frac{d[J(w)]}{dw}$ ↑ slope

}

[:= denotes updating the value]

→ $w := w - \alpha \frac{\partial J(w,b)}{\partial w}$

$b := b - \alpha \frac{\partial J(w,b)}{\partial b}$

→ In coding, $\frac{\partial J}{\partial w}$ is assigned to 'dw' variable & $\frac{\partial J}{\partial b}$ will be 'db' variable.

Teachers Sign

Logistic Regression

→ Imp formulas:

→ $z = w^T x + b$

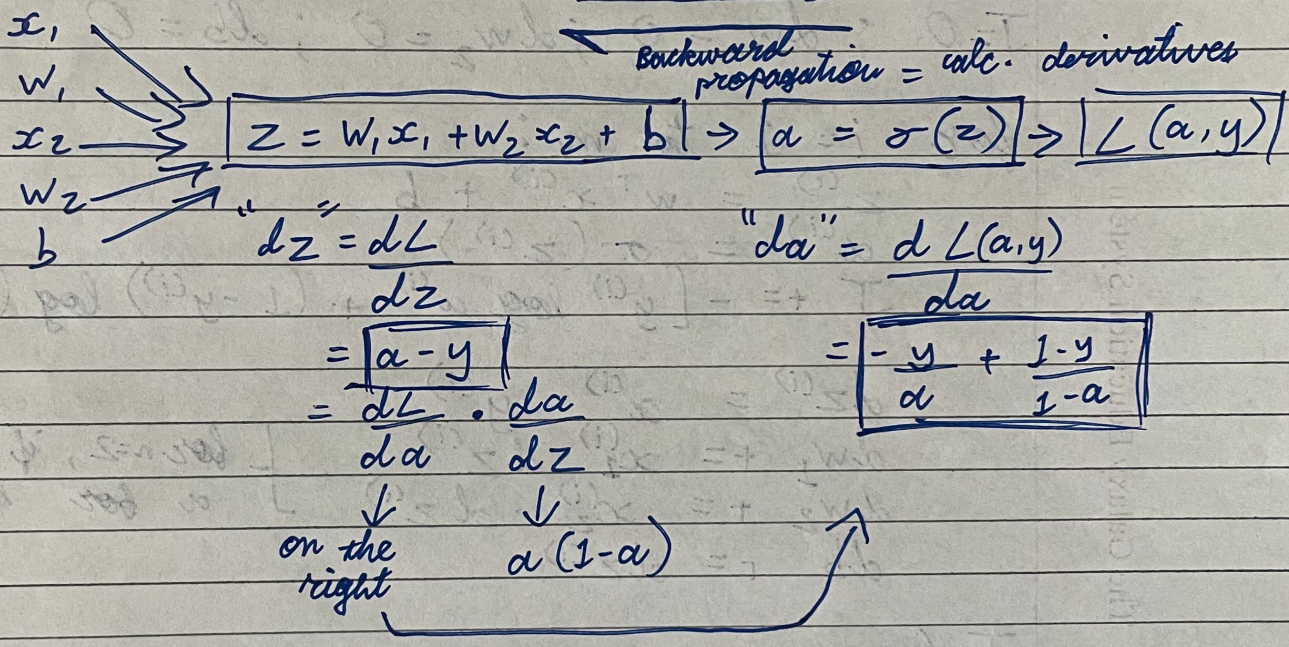
→ $\hat{y} = a = \sigma(z)$

→ $\mathcal{L}(a, y) = -[y \log(a) + (1-y) \log(1-a)]$

→ Computational graph

Forward propagation = calculating values

Backward propagation = calc. derivatives



"dz" $= \frac{d\mathcal{L}}{dz} = \boxed{a - y} = \frac{d\mathcal{L}}{da} \cdot \frac{da}{dz}$

"da" $= \frac{d\mathcal{L}(a,y)}{da} = \boxed{-\frac{y}{a} + \frac{1-y}{1-a}}$

$\frac{da}{dz}$ ↓ $a(1-a)$

$\frac{d\mathcal{L}}{da}$ ↓ on the right

→ $\frac{\partial \mathcal{L}}{\partial w_1} = "dw_1" = \frac{\partial \mathcal{L}}{\partial z} \cdot \frac{\partial z}{\partial w_1} = dz \cdot x_1$

"dw_1" ↑ coding var. name

dz ↑ coding var.

Similarly,

→ $\frac{\partial \mathcal{L}}{\partial w_2} = x_2 \cdot dz = "dw_2"$

→ $\frac{\partial \mathcal{L}}{\partial b} = "db" = "dz"$

→ Steps

① Calc. z, then a, then $\mathcal{L}(a, y)$.

② Calc. ~~for, then~~ $dz^{(i)}$, then dw_1, dw_2, and db

③ Update $w_1 := w_1 - \alpha\, dw_1$; $w_2 := w_2 - \alpha\, dw_2$

$b := b - \alpha\, db$

④ Repeat

Teachers Sign

→ Implementation using for-loop

→ Just for understanding. In actual practice, vectorized form is implemented ~~as~~ cause for-loops slow af.

→ Code

$J=0 \; ; dw_1 = 0 \; ; dw_2 = 0 \; ; db = 0$

for $i = 1$ to m:

$$z^{(i)} = w^T x^{(i)} + b$$
$$a^{(i)} = \sigma(z^{(i)})$$
$$J \mathrel{+}= -\left[y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log (1-a^{(i)})\right]$$

$$dz^{(i)} = a^{(i)} - y^{(i)}$$
$$dw_1 \mathrel{+}= x_1^{(i)} \cdot dz^{(i)}$$
$$dw_2 \mathrel{+}= x_2^{(i)} \cdot dz^{(i)}$$
$$db \mathrel{+}= dz^{(i)}$$

} for n=2, if more, apply a for loop.

$J \mathrel{/}= m$

$dw_1 \mathrel{/}= m$ $\quad w_1 := w_1 - \alpha dw_1$

$dw_2 \mathrel{/}= m$ $\quad w_2 := w_2 - \alpha dw_2$

$db \mathrel{/}= m$ $\quad b := b - \alpha db$

→ dw_1, dw_2, db are accumulators. At the end we take their average. Hence they don't have superscript (i).
But $dz^{(i)}$ is ~~with~~ w.r.t. just one e.g. hence the superscript (i).

Logistic Regression

→ Vectorizing Logistic Regression.

→ We need to calculate:

$z^{(1)} = w^T x^{(1)} + b$ $\quad z^{(2)} = w^T x^{(2)} + b$ $\quad z^{(3)} = w^T x^{(3)} + b$

$a^{(1)} = \sigma(z^{(1)})$ $\quad a^{(2)} = \sigma(z^{(2)})$ $\quad a^{(3)} = \sigma(z^{(3)})$

i.e., z & a for each training example

→ What we have is X,

where $X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}$

i.e., all the training inputs stacked together in different columns.

shape $= (n_x, m)$

$X \in \mathbb{R}^{n_x \times m}$

↳ means X is a $n_x \times m$ dimensional matrix

→ To get the z for all m-training inputs, a row vector is created of dim. $1 \times m$.

$[z^{(1)} \; z^{(2)} \dots z^{(m)}] = w^T X + [b \; b \dots b]$ ← also a $1 \times m$ vector

$[w_1 \; w_2 \dots w_m] \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}$

which will give us,

$[w^T x^{(1)} \quad w^T x^{(2)} \dots w^T x^{(m)}]$

then add $[b \; b \dots b]$ we get

$[z^{(1)} \; z^{(2)} \dots z^{(m)}] = [w^T x^{(1)} + b \quad w^T x^{(2)} + b \dots w^T x^{(m)} + b]$

Teachers Sign

→ Python code for the above,

$$z = np.dot(w.T, x) + b \quad \text{——— [I]}$$

→ In the code, $b = \mathbb{R}$, But python will automatically expand it to fit the matrix obtained by the np.dot() operation i.e., $1 \times m$ matrix. This is called broadcasting.

→ To get 'a' of all the 'z',

Code

$$A = sigmoid(z) \quad \text{——— [II]}$$

where z is the vector calculated in [I] and sigmoid() is a user-defined function and A will be a matrix of dimension similar to that of z.

→ Calculating derivatives with vectorization

→ $dz^{(1)} = a^{(1)} - y^{(1)}$; $dz^{(2)} = a^{(2)} - y^{(2)}$

we want $dz = [dz^{(1)} \; dz^{(2)} \cdots dz^{(m)}] \rightarrow \mathbb{R}^{1 \times m}$

→ We got $A = [a^{(1)} \cdots a^{(m)}]$ & $Y = [y^{(1)} \cdots y^{(m)}]$

→ So, $dz = A - Y$ ∈ python code

→ As previously seen, $db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$

so python code for that,

$db = \frac{1}{m}$ np.sum(dz)

Logistic Regression

→ Calculating derivatives with vectorization

→ dw was $\frac{1}{m} \sum_{i=1}^{m} x^{(i)} dz^{(i)}$

→ $dw = \frac{1}{m} X dz^T$

$$= \frac{1}{m} \begin{bmatrix} | & & | \\ x^{(1)} & \cdots & x^{(m)} \\ | & & | \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$$

$$= \frac{1}{m} \left[x^{(1)} dz^{(1)} + \ldots + x^{(z)} dz^{(z)} \right]$$

Code

→
```
dw = 1/m * np.dot(X, dz.T)
```

→ Complete code for vectorized logistic regression

```
z = np.dot(w.T, X) + b
A = sigmoid(z)

dz = A - Y
dw = 1/m * np.dot(X, dz.T)
db = 1/m * np.sum(dz)

w := w - α dw
b := b - α db
```

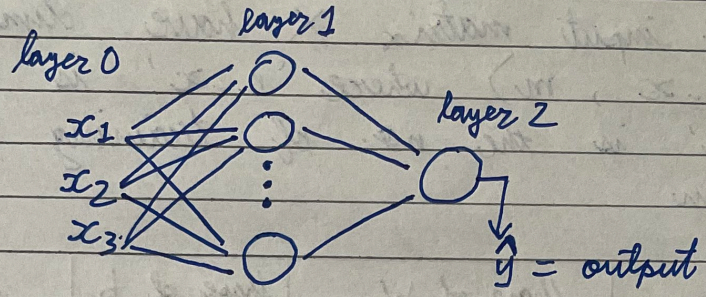
Note: The above code denotes one-iteration of gradient descent. For multiple iterations, a for-loop is necessary.

Teachers Sign

Deep Neural Network

(1) Initialization of parameters

→ 2-layer neural network



	n_x	n_h	n_y
Size of:	Input layer	Hidden layer	Output layer

Parameters : W1, b1, W2, b2

Dimensions of parameters :

→ W1 = (n_h, n_x)

→ b1 = (n_h, 1)

→ W2 = (n_y, n_h)

→ b2 = (n_y, 1)

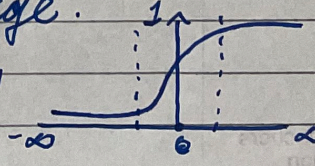
where Wi corresponds to weight matrix of layer i and bi corresponds to vector of bias for layer i.

⇒ Code (Done within function "initialize_parameters(n_x, n_h, n_y)")

```
W1 = np.random.randn(n_h, n_x) * 0.01
b1 = np.zeros((n_h, 1))
W2 = np.random.randn(n_y, n_h) * 0.01
b2 = np.zeros((n_y, 1))
```

Note '0.01' is multiplied to W1 so that the gradient remains large.

E.g. In sigmoid function,



we want to have no. between the dotted lines to have larger gradient and hence have better learning.

Teachers Sign

→ L - layer neural network

→ Let input matrix X have dimensions $n^{[0]}$, (n_x, m) where 'n_x' is no. of features & 'm' is the no. of training examples. Then

	Shape of W	Shape of b	Activation
Layer 1	$(n^{[1]}, n\_x)$	$(n^{[1]}, 1)$	$Z^{[1]} = W^{[1]}X + b^{[1]}$
Layer 2	$(n^{[2]}, n^{[1]})$	$(n^{[2]}, 1)$	$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$
⋮			
IMP → Layer L	$(n^{[L]}, n^{[L-1]})$	$(n^{[L]}, 1)$	$Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$

Shape of $Z^{[L]} = (n^{[L]}, m)$

where $n^{[i]}$ refers to size no. of i^{th} layer

IMP → → In general, $W_i = (n^{[i]}, n^{[i-1]})$ & $b_i = (n^{[i]}, 1)$

→ The no. of units in different layers are stored in a variable 'layer_dims'. For e.g. 'layer_dims' for the 2-layer NN would be $[n\_x, n\_h, n\_y]$.

Code

```
def initialize_parameters_deep(layer_dims):
    parameters = {}
    L = len(layer_dims) # no. of layers in NN

    for l in range(1, L):
        parameters["W" + str(l)] = np.random.randn
        (layer_dims[l], layer_dims[l-1]) * 0.01
        parameters["b" + str(l)] = np.zeros((layer_dims[l]
        , 1))

    return parameters
```

The Galaxy Education System

Teachers Sign

(2) Functions for Forward propagation

→ → Linear Forward

→ The linear forward module (vectorized over all the examples) ~~not~~ will compute :

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$

where $A^{[0]} = X$

→ The values of matrices / vectors W, A, b will be stored in a python tuple for computing the backward pass efficiently in future.

Code

```
def linear_forward (A, W, b):
    Z = np.dot (W, A) + b

    cache = (A, W, b)

    return Z, cache
```

→ Linear Activation Forward

→ Applying the activation function sigmoid or relu to the Z.

→ The function to be coded will return ~~to~~ $\sigma(z)$ or relu(z) depending on choice, and a activation cache that stores Z.

E.g. of activation function code

Code

```
def sigmoid (z):

    A = 1 / (1 + np.exp (-z))

    return A , Z
```

⇒ Hence the sigmoid () will return two items: the activation value "a" and a "cache" containing "Z" (which will be fed to the backward function).
Can be called by ' A, activation_cache = sigmoid(z)'

⇒ Mathematical equation : $A^{[l]} = g(Z^{[l]}) = g(W^{[l]} A^{[l-1]} + b^{[l]})$
where g can be sigmoid or relu

Code

```
def linear_activation_forward(A_prev, W, b, activation):

    if activation == 'sigmoid':
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)

    elif activation == 'relu':
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)

    cache = (linear_cache, activation_cache)

    return A, cache
```

[redacted]

Deep Neural Networks

→ L-layer model

→

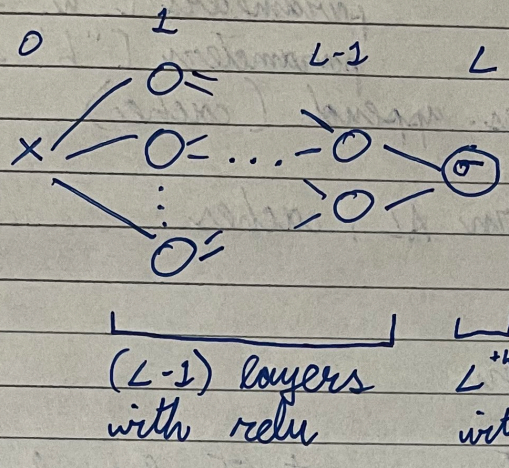


Fig. [LINEAR → RELU] × (L-1) → LINEAR → SIGMOID

→ Implementing forward propagation for the [linear → Relu] * (L-1) → linear → sigmoid computation

Code

```
def L_model_forward(X, parameters):
    caches = []
    A = X
    L = len(parameters) // 2    # no. of layers

    # (Linear → Relu) * (L-1)
    # loop starts from 1 as layer 0 is the input
    for l in range(1, L):
        A_prev = A

        A, cache = linear_activation_forward(
                    A_prev, parameters["W" + str(l)],
                    parameters["b", str(l)], "relu")
        caches.append(cache)


    # (linear → sigmoid) for final layer
```


Teachers Sign

```
AL, cache = linear_activation_forward(A,
            parameters["W" + str(L)],
            parameters["b" + str(L)], "sigmoid")
caches.append(cache)

return AL, caches
```

(5) Cost Function

⇒ Cross-entropy cost J formula:

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

Code

```
def compute_cost(AL, Y):

    m = Y.shape[1]  # no. of training examples

    cost = -1/m * ((np.dot(Y, np.log(AL.T)))
           + (np.dot(1-Y, np.log(1-AL.T))))

    cost = np.squeeze(cost)  # To make
                             sure [[17]] turns
                             to 17.

    return cost
```

Deep Neural Network

⑥ Backward propagation

→ np.sum(axis = 0/1) → 0 = column-wise, 1 = row-wise

→ Linear Backward

→ For layer l, the linear parts is: $Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$ (followed by an activation)

Suppose we already have $dZ^{[l]} = \frac{\partial \mathcal{L}}{\partial Z^{[l]}}$.
We want $(dW^{[l]}, db^{[l]}, dA^{[l-1]})$.

→ Mathematical formulas:

→ $dW^{[l]} = \frac{\partial J}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} (A^{[l-1]}.T)$ ← Transpose of $A^{[l-1]}$

→ $db^{[l]} = \frac{\partial J}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)}$

→ $dA^{[l-1]} = \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]}$ ← Transpose

Code

```
def linear_backward(dZ, cache):

    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = 1/m * np.dot(dZ, A_prev.T)
    db = 1/m * np.sum(dZ, axis = 1, keepdims = True)
    dA_prev = np.dot(W.T, dZ)

    return dA_prev, dW, db
```

The Galaxy Education System

Teachers Sign

→ Linear Activation Backward

→ Two functions need to be user defined namely 'sigmoid_backward' & 'relu_backward'. The functions implement backward propagation for sigmoid & relu unit respectively.

→ Called by:

dZ = sigmoid_backward(dA, activation_cache)

OR

dZ = relu_backward(dA, activation_cache)

→ If $g(.)$ is the activation function, the sigmoid_backward & relu_backward compute:

$$dZ^{[l]} = dA^{[l]} * g'(Z^{[l]})$$

Code

```
def linear_activation_backward(dA, cache, activation):
    linear_cache, activation_cache = cache

    if activation == "relu":
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_
                                          cache)

    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear
                                          :cache)

    return dA_prev, dW, db
```


Teachers Sign

Deep Neural Network

→ L-Model Backward

→ In L-model-forward(), a cache was stored containing (X, W, b & Z). Those will be used to compute gradients.
In L-model-backward(), we'll iterate through all the hidden layers backward.

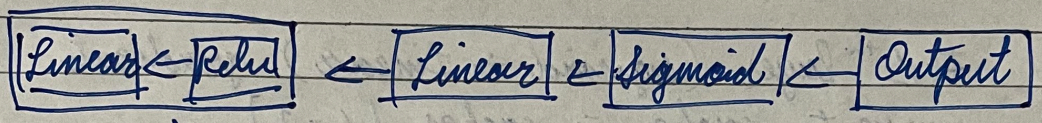


Fig. Backward Pass

→ We know $A^{[L]} = \sigma(Z^{[L]})$. We need $dAL = \frac{\partial L}{\partial A^{[L]}}$.

Loss $= -[y \log a + (1-y) \log(1-a)]$ hence, $\frac{\partial L}{\partial a}$ would be $\frac{\partial L}{\partial a} = \left[-\frac{y}{a} + \frac{1-y}{1-a}\right]$

Therefore, dAL = -(np.divide(Y, AL) - np.divide(1-Y, 1-AL))

→ Each value of dA, dW, and db will be stored in the 'grads' dictionary.

Code

```
def L_model_backward(AL, Y, caches):
"""
    'caches' is list of caches containing:
    - every cache of linear_activation_forward()
      with "relu" (it's caches[l], for l in range(L-1))
    - the cache of linear_activation_forward()
      with "sigmoid" (it's caches[L-1])
"""
```

Teachers Sign

```
grads = { }
L = len(caches)
m = AL.shape[1]
Y = Y.reshape(AL.shape)

dAL = -(np.divide(Y, AL) - np.divide(1-Y, 1-AL))

# Lth layer (sigmoid -> Linear) gradients
current_cache = caches[L-1]
dA_prev_temp, dW_temp, db_temp =
 linear_activation_backward(dAL, current_cache, "sigmoid")
grads["dA" + str(L-1)] = dA_prev_temp
grads["dW" + str(L)] = dW_temp
grads["db" + str(L)] = db_temp

# loop for l = L-2 to l=0
for l in reversed(range(L-1)):
    current_cache = caches[l]
    dA_prev_temp, dW_temp, db_temp =
    linear_activation_backward(~~dAL~~
    grads["dA" + str(l+1)], current_cache, "relu")

    grads["dA" + str(l)] = dA_prev_temp
    grads["dW" + str(l+1)] = dW_temp
    grads["db" + str(l+1)] = db_temp

return grads
```

Deep Learning

→ Updating parameters

→ Formula :

→ $W^{[l]} = W^{[l]} - \alpha dW^{[l]}$

$b^{[l]} = b^{[l]} - \alpha db^{[l]}$

→ After updating parameters, stored in the 'parameters' dictionary.

Code

```
def update_parameters(params, grads, l_r):
    parameters = params.copy()
    L = len(parameters) // 2

    for l in range(L):
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] - l_r * grads["dW" + str(l+1)]
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - l_r * grads["db" + str(l+1)]

    return parameters
```

Deep Learning
Regularisation Notes

By Divyajeet Pala

Deep Learning Regularization

→ Ways to solve a neural network overfitting on the data i.e., high variance problem :

→ Regularization
→ Get more training data

→ Understanding Regularization based on Logistic regression.

→ Cost function J for Logistic regression :

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

where $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$, m = no. of training e.g.

→ When we want to regularize, we modify the equation as :

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

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

& λ = regularization parameter

→ $\|w\|_2$ refers to the L_2 norm or Euclidean norm, where :

$$\|a\|_2 = \sqrt{\sum_{i=1}^{n} a_i^2}$$

Hence, the regularization used above is called L_2 regularization.

Teachers Sign

→ L_1 regularization

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

where $||w_1||$ is sum of the absolute value of weights.

⇒ On using L_1 reg., w will end up being sparse i.e., w vector will have a lot of zeros in it.

Note In practice, L_2 regularization is used more often.

→ Regularization in terms of Neural Network

→ For neural network,

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

where $||w^{[l]}||_F^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{ij}^{[l-1]})^2$ $\because w : (n^{[l]}, n^{[l-1]})$

i.e., sum of square of all weights

→ $||w||_F^2$ is known as the "Frobenius norm" of the matrix W.

Deep Learning Regularization

→ Implementing Gradient Descent with Regularization

⇒ Previously, we would calculate $dW^{[\ell]}$, which is $\frac{\partial J}{\partial W^{[\ell]}}$ through the process of back-propagation.

⇒ Then update:

$$W^{[\ell]} := W^{[\ell]} - \alpha\, dW^{[\ell]}$$

⇒ Now after implementing regularization i.e., adding the $\frac{\lambda}{2m}\sum_{\ell=1}^{L} \|W^{[\ell]}\|_F^2$ to the cost function, we ~~update~~ modify the equation for gradient descent as follows:

$$\Rightarrow dW^{[\ell]} = (\text{value from backprop.}) + \frac{\lambda}{m} W^{[\ell]}$$

$$\Rightarrow W^{[\ell]} := W^{[\ell]} - \alpha\, dW^{[\ell]}$$

$$= W^{[\ell]} - \alpha\left(\text{backprop value} + \frac{\lambda}{m} W^{[\ell]}\right)$$

$$= W^{[\ell]} - \frac{\alpha\lambda}{m} W^{[\ell]} - \alpha(\text{backprop value})$$

$$= \left(1 - \frac{\alpha\lambda}{m}\right) W^{[\ell]} - \alpha(\text{backprop value})$$

⇒ For this reason, L_2 regularization is also known as "weight decay" since we are decreasing the original weights by multiplying with a number little less than 1.

Deep Learning
Vanishing Gradient Notes

By Divyajeet Pala

Vanishing Gradient Problem

→ Using more layers in a neural network causes more activation functions. And as we increase that, increases the gradients of loss function which leads to zero.
What this causes is that the weights initial layers will not update much (i.e.) will update by infinitesimally small amount. So the network won't improve much

→ Partial solution
→ One of the ways in which it can be partially solved is by initializing the weights in a different manner.

→ $z = w_1 x_1 + w_2 + x_2 + \dots$
We want z to not blow up and nor become too small.

→ For that, larger the n (no. of layers), w_i should be smaller. Since $z = \sum_{i=1}^{n} w_i x_i$, if we are adding more number of $w_i x_i$ terms due to more no. of layers, we want each of the $w_i x_i$ terms to be smaller.

→ To solve the issue (partially), we can set the variance of w_i to $1/n$ where n = no. of input features going into the neuron.

$$\text{Var}(w_i) = \frac{1}{n}$$ where n = no. of input features going into the neuron

To do this, initialize as follows:

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

Teachers Sign

→ If the activation function used is Relu, $Var(w_i) = \frac{2}{n}$ works better. It can be achieved by:

$$w^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{2}{n^{[l-1]}}\right)$$

→ For activation function tanh:

$$Var(w_i) = 1/n$$

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

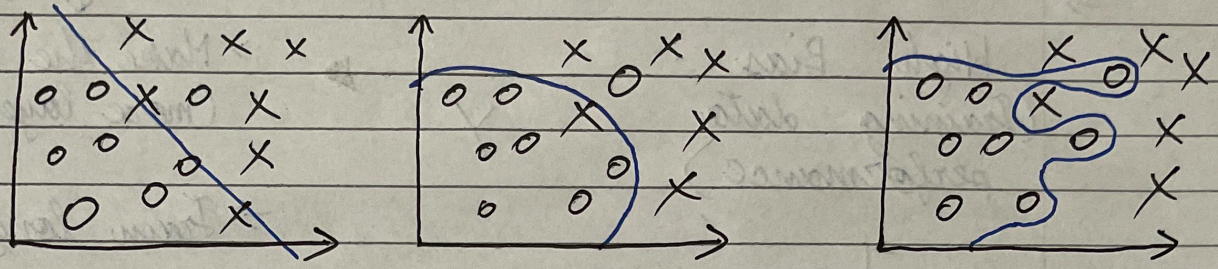
Xavier initialization

Deep Learning
Bias and Variance Notes

By Divyajeet Pala

DL -> Bias and Variance

Name ..Divyajeet Pala..
Subject ..NN..... Chapter .....
Date ..... Page No. ..1.....



High bias
"Underfitting"

Just right

High Variance
"Overfitting"

Train set error:	1%	15%	15%	0.5%
Dev set error:	11%	16%	30%	1%
Assuming optimal error to be 0%:	High Variance	High Bias	High bias & high variance	Low bias & variance

-> The train set error gives a sense of the bias problem and the dev set error gives a sense of variance problem.

-> Note that if the optimal error for the above e.g. was 15%, the second model would be best and would be considered as low bias & low variance.

-> To solve the issue of high bias and/or high variance, a few things can be done. Basic recipe of ML ~~summary~~ tells a few of those ways.

⇒

High Bias
(Training data performance)

Y → Σ most helpful
→ Make the NN bigger (more layers / hidden units)
→ Train longer

N ↓

High variance
(Dev set performance)

Y → → Get more data & train again
→ Regularization

N ↓

Done

Deep Learning
Gradient Checking Notes

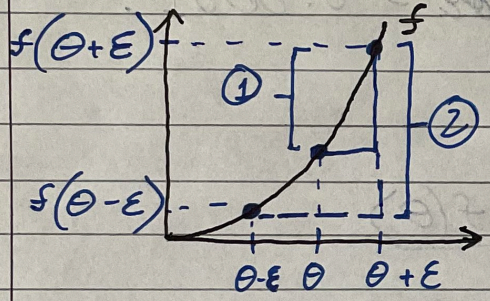
By Divyajeet Pala

Checking derivative computation

⇒ Derivatives are calculated when implementing back propagation in a neural network to update the weight matrices and biases.

⇒ There is a test called gradient checking that can help make sure that the implementation of back prop is correct.

⇒ E.g. $f(\theta) = \theta^3$



if $\varepsilon = 0.01$,

$\theta = 1$

$\theta + \varepsilon = 1.01$

$\theta - \varepsilon = 0.99$

① ⇒ The method with which gradient is actually calculated.

② ⇒ Method proposed for checking the gradient

⇒ ② $= f(\theta+\varepsilon) - f(\theta-\varepsilon)$

⇒ Width of the bigger Δ is $[\theta+\varepsilon - (\theta-\varepsilon)] = 2\varepsilon$.

$$\Rightarrow \text{Gradient} = \frac{\text{Height}}{\text{width}} = \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx f'(\theta)$$

$$= \frac{(1.01)^3 - (0.99)^3}{2(0.01)} = 3.0001$$

$$\Rightarrow g(\theta) = f'(\theta) = 3\theta^2 = 3(1)^2 = 3$$

Approx. error ⇒ $3.0001 - 3 = 0.0001$

⇒ As the error is very less, the two-sided difference gradient checking can be used to evaluate the derivatives.

The Galaxy Education System

Teachers Sign

→ Why two-sided difference $(\theta+\varepsilon, \theta-\varepsilon)$ is better than $(\theta, \theta+\varepsilon)$ one-sided difference?

$$f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$$ ← Formal definition of derivative

The error for above is $\overset{\text{big-O}}{\downarrow} O(\varepsilon^2)$ where ε is a value tending to 0 (zero) i.e., very small value.

E.g. if $\varepsilon = 0.01$, error $= 0.0001$.

For one-sided diff.,

$$f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}$$

the error tends to be $\overset{\text{big-O}}{\downarrow} O(\varepsilon)$.

E.g. For $\varepsilon = 0.01$, error $= 0.01$

which is significantly more than 0.0001.

→ Therefore, one-sided difference is a much less accurate approximation when compared to two-sided difference.
For the same reason, two-sided difference is used when doing gradient checking.

→ By comparing the derivative from back-prop with the calculated two-sided difference, the accuracy can be checked of the back prop.

Gradient Checking

⇒ Implementation

→ $W^{[i]}$ refers to the weight matrix of layer i.
$b^{[i]}$ refers to bias of layer i.

→ Take $W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}$ and reshape into a big vector θ. θ will contain all the values of $W^{[1]} \ldots b^{[L]}$.

Therefore, the cost function J which earlier was a function of all the different weights & bias, will now be a function of θ.

Earlier : $J(W^{[1]}, b^{[2]}, \ldots, W^{[L]}, b^{[L]})$
Now : $J(\theta)$

→ In the same way as before, concatenate $dW^{[1]}, db^{[1]}, \ldots, dW^{[L]}, db^{[L]}$ and reshape into a big vector $d\theta$.

Shape of θ & $d\theta$ will be same.

⇒ Now checking whether $d\theta$ is the gradient of $J(\theta)$.

→ Gradient checking

Code (Pseudo)

for each i in θ : # iterating over the elements of vector θ

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

⇒ Basically getting a vector $d\theta_{approx.}$ where the i^{th} element is two-sided difference of $J(\theta)$ in which the i^{th} element is added or subtracted with ε & then two-sided difference is calculated.

Teachers Sign

→ Formula :

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

so, $d\theta_{approx.}[i] \approx \frac{\partial J}{\partial \theta_i}$.

Dimension of $d\theta_{approx.}$ will be same as that of $d\theta$.

Aim is to check whether $d\theta_{approx}$ & $d\theta$ are almost equal or not.
To do that euclidean distance between them is calculated.

→ Check $$\frac{||d\theta_{approx.} - d\theta||_2}{||d\theta_{approx}||_2 + ||d\theta||_2} \quad ---\ (i)$$

where → $||d\theta_{app.} - d\theta||_2$ is the L2 norm i.e., sum of squares of element-wise difference and then taking the root of ~~answer~~ answer.

→ The numerator is the euclid. distance and denominator is used to normalize the values.

→ In practice, $\varepsilon = 10^{-7}$ will be good.
so with $\varepsilon = 10^{-7}$,
→ if the formula (i) gives 10^{-7} or smaller, the derivative approximation is correct ✓.
→ if (i) $\approx 10^{-5}$, okayish (can work but double-checking necessary.
→ if (i) $\approx 10^{-3}$, the implementation of back prop has a bug.

Teachers Sign

Deep Learning
Optimisation Algorithms Notes

By Divyajeet Pala

Optimization Algorithms

Exponentially moving averages ~~SAL~~ (EMA)

→ A moving average shows an average of data-points for a certain number of time periods. It's called 'moving' because each data-point is calculated using data from previous X number of time periods. By averaging prior data, MA smooth the price data to form a trend following indicator. They don't predict price direction, but rather define the current direction.

→ E.g. Let $\theta_1, \theta_2, \theta_3 \ldots$ be temperatures of a place where θ_i is the temperature on i-th day.

Calculating moving averages:

→ Initializing $V_0 = 0$.

→ $V_1 = 0.9 V_0 + 0.1 \theta_1$ ← That day's temperature
(↑ previous value)

→ $V_2 = 0.9 V_1 + 0.1 \theta_2$

→ $V_3 = 0.9 V_2 + 0.1 \theta_3$

⋮

→ $V_t = 0.9 V_{t-1} + 0.1 \theta_t$ ← general formula

→ Formula

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

→ In the e.g. $\beta = 0.9$.

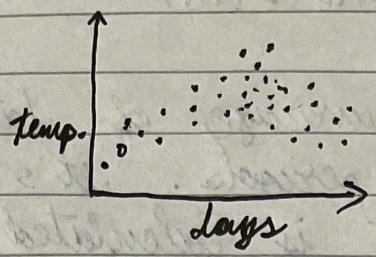
→ Here, V_t averages over $\frac{1}{1-\beta}$ number of days' temperature.

So, when $\beta = 0.9$, V_t would approximately average over 10 days of temperatures.

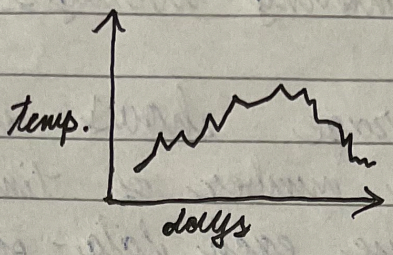
→ If $\beta = ~~0.09~~ 0.98$, $V_t \approx 50$ days.

Teachers Sign

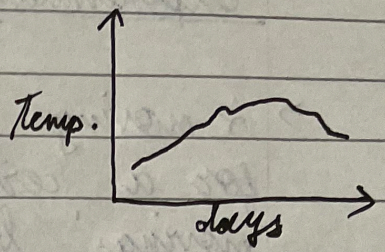
→



Scatter plot of temp V/s days



V_t graph when $\beta = 0.9 \approx 10$ days average



V_t when $\beta = 0.98 \approx 50$ days avg.

→ As seen, when β is high, the graph is smoother. Downside of having high β is that the EMA formula, adapts more slowly to the temperature changes.

The reason:

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

when β is large, it gives more weight to V_{t-1} i.e., the previous values, and a smaller weight to the θ_t i.e., current temperature.

→ Intuition behind ~~EAT~~ EMA in the next page.

Exponentially moving averages (EMA)

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

$\Rightarrow$ Writing the equations for V_t in decreasing order of t with $\beta = 0.9$:

$\Rightarrow V_{100} = 0.9 V_{99} + 0.1 \theta_{100}$
$V_{99} = 0.9 V_{98} + 0.1 \theta_{99}$
$V_{98} = 0.9 V_{97} + 0.1 \theta_{98}$
...

$\Rightarrow$ Substituting V_{99} in the eq. of V_{100} we get,

$$V_{100} = 0.1\theta_{100} + 0.9(0.9 V_{98} + 0.1\theta_{99}) \quad \text{--- (i)}$$

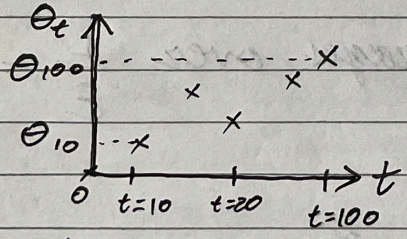
$\Rightarrow$ Subst. V_{98} in (i),

$$V_{100} = 0.1\theta_{100} + 0.9[0.1\theta_{99} + 0.9(0.1\theta_{98} + 0.9 V_{97})]$$

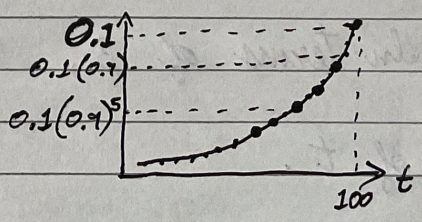
$\Rightarrow$ Expanding, we get:

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

$\Rightarrow$ θ_t v/s t scatter plot



Scatter plot of the datapoints



An exponentially decaying graph for the $0.1(0.9)^x$ part of the equation i.e., co-efficients

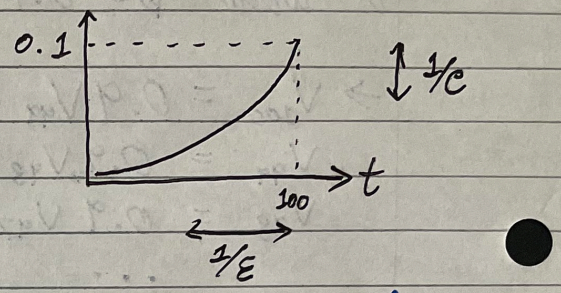
$\Rightarrow$ For calculating, ~~V_{100}~~, V_{100}, sum up the element-wise product of

the two functions.

⇒ The sum of the co-efficients (0.1×0.9^x part of the term), is approx. 1.

⇒ Let $\beta = 1 - \varepsilon$,

$$(1-\varepsilon)^{1/\varepsilon} = \frac{1}{e}$$



Basically, the equation tells us that it takes $1/\varepsilon$ days for the height of the graph to decay to $1/e \sim 1/3$ of the peak.

So, for $\beta = 0.9$, $\varepsilon = 0.1$. Hence it takes $1/\varepsilon = 10$ days for the peak of the graph to come down to $1/e$ or approx. $1/3$ of the peak.

$$(0.9)^{1/0.1} \Rightarrow 0.9^{10} \approx 1/e$$

Hence we can say that ~~after~~ values before 10 days, contribute a very less weight $<< 1/e$(peak), to the current moving average value. Hence we say that $\beta = 0.9$, averages ~~up~~ 10 days of temperature.

In terms of β, V_t averages over $\frac{1}{\varepsilon} = \frac{1}{1-\beta}$ values of t.

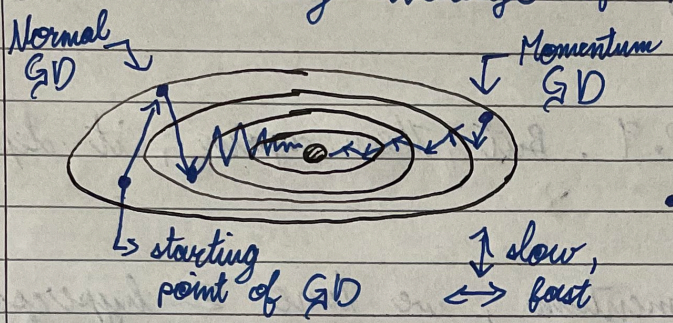
⇒ We could just add the values of 10 days and divide with 10 to get average, but that is computationally and memory-wise really expensive though more accurate. EMA is more efficient when no. of variables is high.

Teachers Sign

Optimization Algorithms

Gradient Descent with Momentum

→ Idea: The basic idea is that instead of updating weights (w) & biases (b) with dW & db, we update them with the exponentially moving average of the gradients dW & db.



● = ~~minimum~~ minimum

→ = one step/iteration of GD

We want learning to be slower vertically, & faster horizontally.

→ Applying momentum:

On iteration t:

Compute dW, db on the current mini-batch/batch:

~~$V_{dw} = \beta$~~

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

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

where α = learning rate, β = ~~EMA~~ EMA term

→ Intuition:

Think of the function as a bowl shaped function. We roll down a ball, the terms dW, db act as acceleration, and V_{dw}, V_{db} (being multiplied β) act as velocity. So due to acceleration, our ball moves faster & faster towards the center instead of going here-and-there much. β acts as friction and

Teachers Sign

prevents the ball from speeding infinitely.

So due to momentum, gradient descent algorithm now doesn't take the steps independently of the previous steps. Instead previous steps momentum helps in making the ball reach the minima point faster.

⇒ Usually, $\beta = 0.9$. But, then again, it depends on the problem.

⇒ When using momentum, we tune 2 hyperparameters namely, α & β. Where α is the learning rate and β is the momentum term.

Optimization Algorithms

RMS prop

⇒ Idea : Updating params W & b by dW & db divided by a ~~numbers~~ terms that would be small, if the gradient of the term (W or b) would favour in reaching the minima and would be large if the direction of the gradient of the term (W or b) would not favour reaching the minima.

⇒ On iteration t :
compute dW, db for the mini-batch / batch :

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2)\, dW^2 \quad \leftarrow \text{element-wise squaring}$$
$$S_{db} = \beta_2 S_{db} + (1-\beta_2)\, db^2 \quad \leftarrow$$

$$W := W - \alpha \frac{dW}{\sqrt{S_{dw}}}, \quad b := b - \alpha \frac{db}{\sqrt{S_{db}}}$$

naming convention

Note // β_2 is just β, the β_2 just separates the β of RMS prop from that of momentum.

⇒ Suppose, the direction of gradients of W favour in reaching the minima. Then, while calculating S_{dw}, $(dW)^2$ will turn out to be small. Hence dW will be divided with a relatively small no.. so the updates in W direction will be larger as the update (dW) is divided with a small number ($\sqrt{S_{dw}}$).

If direction of gradient of b is not favourable, S_{db} will be a larger number as $(db)^2$ will be large.
Hence, the update will be smaller as the denominator ($\sqrt{S_{db}}$) will be large.

Teachers Sign

Optimization Algorithms

Adam Optimization

⇒ Idea : Combining momentum with RMSprop.

⇒ Note : β_1 corresponds to β of momentum, and β_2 to β of RMSprop.

⇒ Initialize $V_{dw} = 0$, $S_{dw} = 0$, $V_{db} = 0$, $S_{db} = 0$

On iteration t :

Compute dW, db using current mini-batch

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

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

Bias correction for all terms

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

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

Updating W and b

$$W := W - \alpha \frac{V_{dw}^{corr.}}{\sqrt{S_{dw}^{corr.}} + \varepsilon}$$

$$b := b - \alpha \frac{V_{db}^{corr.}}{\sqrt{S_{db}^{corr}} + \varepsilon}$$

Note // ε is a small value which prevents the term from exploding in case the $\sqrt{S^{corr}}$ term becomes very small.

Teachers Sign

→ Hyperparameter tuning :

→ α (learning rate) : Needs to be tuned according to the problem

→ β_1 (momentum parameter) : 0.9 (by default) but can be changed

→ β_2 (RMS prop parameter) : 0.999 (by default) is optimum but can be changed

→ ε : 10^{-8} (Doesn't contribute much to model's performance)

→ Learning Rate Decay

Idea : As the GD gets closer to the minima, the learning rate decreases so as to provide a smoother slope and prevent over-shooting.

Formula : $$\alpha = \frac{1 \times \alpha_0}{1 + (\text{decay-rate} \times \text{epoch-num})}$$

E.g. $\alpha_0 = 0.2$, decay = 1

Epoch	α
1	0.1
2	0.067
3	0.05
⋮	⋮