Playlist-2

Divide Data in | Train | Dev | Test |

Bias ∝ train data
variance ∝ dev/test data

High Bias → Big network, Train longer (try diff NN architecture) } try
(training data problem)

High variance → More data, Regularisation } Try
(dev set performance) (try diff NN architecture)

⇒ Regularisation

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

(λ: Regularisation parameter; $\frac{\lambda}{2m}b^2$ ~ doesn't contri much)

L_2 regularisation | $\|w\|^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$ | euclidian norm

L_1 regularisation | we use, $\frac{\lambda}{2m}\|w\|_1 = \frac{\lambda}{2m}\sum_{i=1}^{n_x}|w|$

This makes w more sparse, it may take less memory.

For NN,

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

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

↑ "Frobenius Norm"

Now,

$$dw^{[l]} = \frac{dJ}{dw^{[l]}}$$

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

$$w^{[l]} := w^{[l]} - \alpha\, dw^{[l]} - \alpha\,(\text{back prop})$$

weight decay

weight decay is also called L2 regularisation

If $\lambda \uparrow$, $w^{[l]} \downarrow$

$\therefore z^{[l]} \downarrow$ as $z^{[l]} = w^{[l]} x + b^{[l]}$

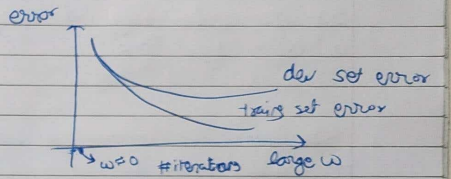
So it will become linear and not have good boundaries

⇒ Dropout Regularisation

Train small-small parts of NN, based on probability we train some & choose to not train some

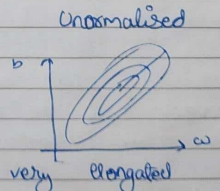
Downside: J is not well defined.

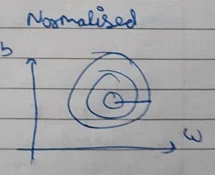
Early stopping



⇒ Normalization.

It is scaling the data to increase the variance if the scale of the inputs is not similar





For NN,

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

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

↑ "Forbenius Norm"

Now,

$$dw^{[l]} = \frac{dJ}{dw^{[l]}}$$

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

$$w^{[l]} := w^{[l]} - \alpha\, dw^{[l]} - \alpha\,(\text{back prop})$$

weight decay

weights decay is also called L2 regularisation

If $\lambda \uparrow$, $w^{[l]} \downarrow$

$z^{[l]} \downarrow$ so $z^{[l]} = w^{[l]} x + b^{[l]}$

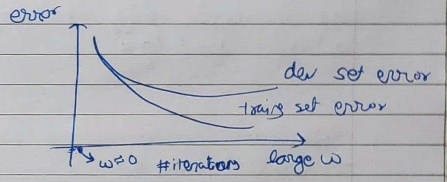
So it will become linear and not how good boundaries

⇒ Dropout Regularisation.

Train small-small parts of NN, based on probability we train some & choose to not train some.

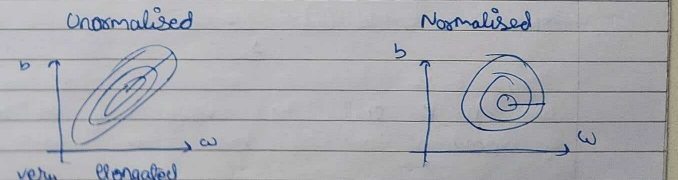
Downside: J is not well defined.

Early stopping



⇒ Normalization.

It is scaling the data to increase the variance if the scale of the inputs is not similar



→ Weight initialization

$z = w_1 x_1 + w_2 x_2 \cdots + w_n x_n + b$

↑ n we want 'w' to be small so that z is small.

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

$Var(w_i) = \frac{1}{n}$ for Relu $Var(w_i) = \frac{2}{n}$

~~tanh~~ to $np.sqrt\left(\frac{2}{n^{[l-1]}}\right)$

So we need diff. ini. of weight for diff. activation func ($g(z)$)

→ Gradient checking.

$J(\theta) = J(\theta_1, \theta_2, \dots)$

for each i:

$d\theta_{approx}[i] = \lim_{\varepsilon \to 0} \frac{J(\theta_1, \theta_2, \dots, \theta_i + \varepsilon, \dots) - J(\theta_1, \theta_2, \dots, \theta_i - \varepsilon, \dots)}{2\varepsilon}$

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

Keep $\varepsilon \approx 10^{-7}$

We check this for every i, if the matrix of $d\theta_{approx}$ and $d\theta$ have same dimensions and are almost same, we are correct.

To check, $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2} \approx$ ~~d~~ 10^{-7} great, 10^{-5}, 10^{-3} — wrong

→ Gradient checking implementation notes

- dont use ~~it~~ in training – only to debug
- it is very slow
- If algo fails grad check, look at all components (w, b, etc) to try to identify bug.
- Remember Regularisation

eg: if, $J(\theta) = \frac{1}{m} \sum \mathcal{L}(\hat{y}, y) + \frac{\lambda}{2m} \sum_l \|w^{[l]}\|_F^2$

$d\theta$ = gradient of J wrt θ, also include this term

- It doesn't work with dropout
- Run random ini., perhaps after training, just to be sure that, initially $w \& b \neq 0$

WEEK-2

→ Mini-Batch Grad. descent / ~~Stochastic gradient descent~~

Good for $m \geq 2000$

It is very helpful for large data set, so here we ~~diffe~~ divide data into mini-batches, for ~~gra~~ each step of grad. descent we need to process data, so it make sense if we already start grad. dest. of mini-batches of processed data.

eg:

$(n_x \times m)$ $X = [x^{(1)} \dots x^{(1000)} \dots x^{(m)}]$

$(1 \times m)$ $Y = [y^{(1)}, \dots, y^{(1000)}, \dots y^{(m)}]$

for m ↑↑ eg: 5,000,000, it makes sense to make a batch of let say 1000

$\therefore x^{(1)} \dots x^{(1000)} = X^{\{1\}}$

$\{j\}$ ⇒ is for mini batch

∴ for t = 1 to 5000 :

Forward prop on $X^{\{t\}}$

$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$

$A^{[1]} = g^{[1]}(Z^{[1]})$

.

.

.

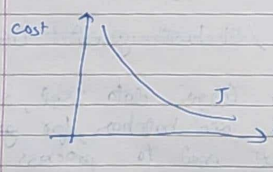
} vertical implementation (1000 eg)

Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{1000} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2(1000)} \sum \|w^{[l]}\|_F^2$

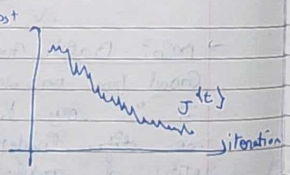
Back prop to compute gradients cost $J^{\{t\}}$ using $(X^{\{t\}}, Y^{\{t\}})$

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

Batch grad. desc



Mini Batch



If mini batch size = 1000 then it is batch,

If size = 1, it is called stochastic grad. descent

Not recommended

Recommend to keep size in 2^n nCr

→ Exponentially weighted ~~average~~ avg.

eg: $V_t = \beta V_{t-1} + (1-\beta)\theta_t$ → ∴ It is an exponential function

~~β = 0 depicts~~

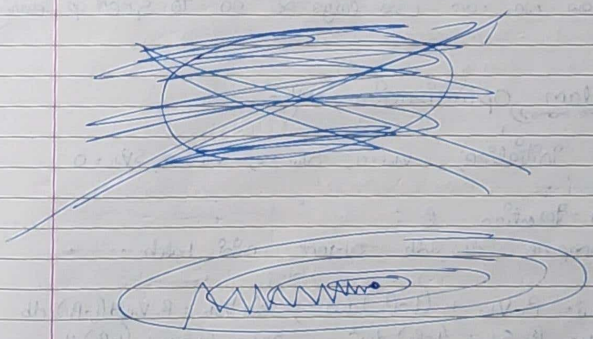
turns out $V_t \approx \frac{1}{1-\beta}$ days avg temp

∴ for $\beta = 0.98$

V_t = 50 days avg temp

Bias Correction is important as in early period (i.e. t≈0) the function doesn't work as expected, although later it becomes correct.

→ Gradient Descent with momentum



more ↓ slower the learning rate

more ↔ Faster

→ Implementation

On iteration t:

Compute dw, db on current mini batch

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

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

(default β = 0.9)

~~Reorder~~

→ RMSprop (root mean sq. propagation)

For it assume b ↕ & w ↔

On iteration t :

Compute dw, db on current mini batch

$S_{dw} = \beta S_{dw} + (1-\beta_2) \underbrace{dw^2}_{\text{small}}$ (element wise), $S_{db} = \beta_2 S_{db} + (1-\beta_2) \underbrace{db^2}_{\text{large}}$

$w := w - \alpha \frac{dw}{\sqrt{S_{dw}+\varepsilon}}$, $b := b - \alpha \frac{db}{\sqrt{S_{db}+\varepsilon}}$

(just to make sure denominator ≠ 0; ε is very small ie 10^{-8})

∴ Now we'll get large w (i.e. ↔) & small b (i.e. ↕)

Now we can use large α so to speed up learning

→ Adam (Adaptive moment) Optimization Algo

We initialize, $V_{dw}=0$, $S_{dw}=0$, $V_{db}=0$, $S_{db}=0$

On iteration t :

Compute dw, db current mini batch

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

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

Corrected

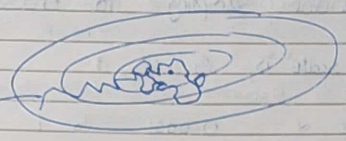
$V_{dw} = V_{dw} / (1-\beta_1^t)$ $V_{db} = V_{db} / (1-\beta_1^t)$

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

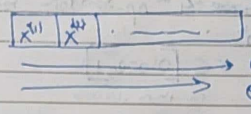
$w := w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}}+\varepsilon}$ $b := w - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}}+\varepsilon}$

α - needs to be tune, β_1 - generally 0.9, β_2 - 0.999 (dw^2), $\varepsilon = 10^{-8}$

→ learning rate decay



with cons. α it may never converge



1 epoch = 1 pass through data

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

eg: epoch	α
1	1
2	0.67
3	0.5
...	...

Let $\alpha_0 = 0.2$, decay rate = 1

Using learning rate decay it will initially take large step then smaller & converge

Other methods — $\alpha = 0.95^{\text{epoch num}} \alpha_0$ (exponential)

or $\alpha = \frac{k}{\sqrt{\text{epoch num}}} \alpha_0$

or discrete staircase

or decrease α manually

WEEK 3

- Use random sampling to choose hyperparameters
- But If the scale is too big,

eg: for α 0.0001 to 1

then we use logarithm for better sampling

$r = -4 * np.random.rand()$

$\alpha = 10^r$

where $r \in [0.0001, 1]$

- For exponentially weighted values

eg: $\beta = 0.9 \ldots$ to 0.999

$1-\beta = 0.1 \ldots$ to 0.001

10^{-1} to 10^{-3}

$r \in (-3, -1)$

$1-\beta = 10^r$

$\beta = 1 - 10^r$

Because we use, $\frac{1}{1-\beta}$

There care be 2 approaches

Babysitting one model (low compute) → like single child full attention making sure it survives

Training many parallel models (more compute) → multiple child, & getting the fastest one

→ Batch Normalization

If this aplies to your algo, it helps you reduce time for hyperparameters search.

Given some intermediate values in NN $z^{(1)} \ldots z^{(m)}$

$z^{(1)}, \ldots, z^{(m)}$

$\mu = \frac{1}{m} \sum z^{(i)}$ $\quad \sigma^2 = \frac{1}{m} \sum (z_i - \mu)^2$

$z^{(i)}_{normalized} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$

we normalize to $z^{(i)}$ to have mean = 0, & var = 1

But we dont want all z to have it,

so,

$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$

γ, β: learnable parameters of model

Now we'll use $\tilde{z}^{(i)}$ instead of $z^{(i)}$

we use to normalize x & $z^{(1)}$

Now we can regulate mean & var by regulating γ & β so we are in control.

$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{Batch Norm (BN)}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \rightarrow z^{[2]} \ldots$

Parameters: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]} \ldots$

$\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \ldots$

In practise BN is usually performed in mini-batches

As mean of $z^{[l]} = 0$, so in its eq^

$$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$$

$b^{[l]}$ is eliminated as it gets cancelled to make mean = 0

Parameters : $w^{[l]}$, $\beta^{[l]}$, $\gamma^{[l]}$

Implementing Grad descent.

for t = 1 to no. of mini batches :
Compute forward prop on $X^{\{t\}}$
In each hidden layer use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$
Use backprop to compute $w^{[l]}$, $\beta^{[l]}$, $\gamma^{[l]}$

Update them :

→ As we normalize X so that there is not much variance in input so it fastens the process, BN does the same this to $z^{[l]}$ as $z^{[l]}$ is kinda input to next hidden layer so it fastens the process

→ Also we know that the output of a layer depends on the output of all layers behind it so changing values of any layer before can make great change in current one. So what batch norm does is it ensure no matter the values change, their mean & variance doesn't change that much making learning process easy.

→ Each minibatch is scaled by mean/variance computed on just that mini batch.

→ This adds some noise to values $z^{[l]}$ within that minibatch. so similar to dropout, it adds some noise to each hidden layer's activations

→ This has a slight regularisation effect.

By reducing batch size noise increases & so does regularisation effect.

→ Softmax Regression

C = # classes

It is used if we want multiple o/p not just 0 or 1

So here instead of one output unit at end there will be C units

& $\hat{y}$ will be $(C \times 1)$ instead (1×1)

eg: let C = 4,

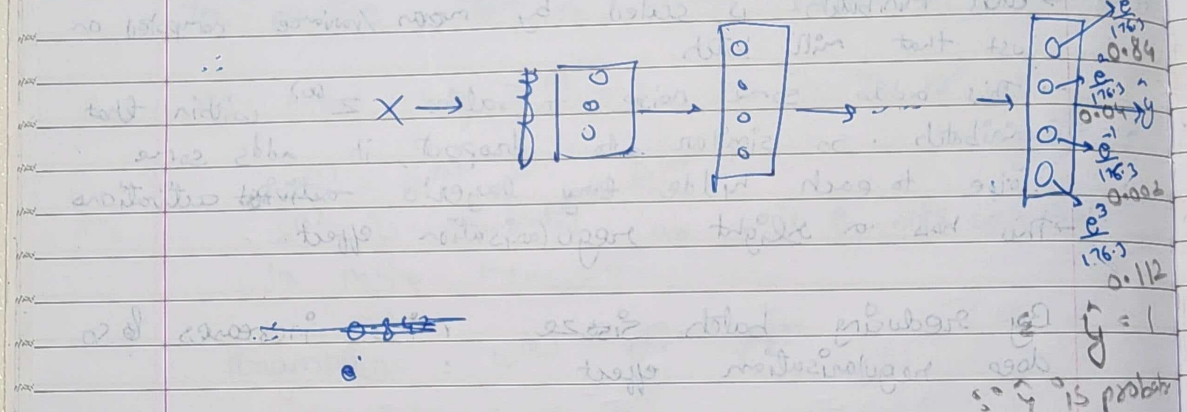
$$z^{[l]}_{(4\times1)} = w^{[l]} a^{[l-1]} + b^{[l]}$$

Activation function :

$$temp = e^{(z^{[l]})}$$

$$a^{[l]}_{(4\times1)} = \frac{e^{(z^{[l]})}}{\sum_{j=1}^{4} temp_i} \quad \Rightarrow \quad a^{[l]}_i = \frac{t_i}{\sum_{j=1}^{4} t_j}$$

eg, $z^{[l]} = \begin{bmatrix} 5 \\ -1 \\ 3 \end{bmatrix}$, $t = \begin{bmatrix} e^5 \\ e^{-1} \\ e^3 \end{bmatrix}$, $t = \sum_{j=1}^{4} t_j = 176.3$, $a^{[l]} = \frac{t_i}{176.3}$


X
0.864
0.084
0.0092
0.112