Neural Networks & Deep learning

ReLU → Rectified linear Unit
(a function that is either 0 or +ve linear)
↳ eliminates -ve values

Densly Connected Layers
all the o/ps from the previous layer are connected to each and every node in the present layer.

Supervised learning
when the i/p and o/p of an application are clearly stated and the model is trained using i/p and expected o/ps as training data

Structured & Unstructured Data → images, text, speech etc.

databases with tables

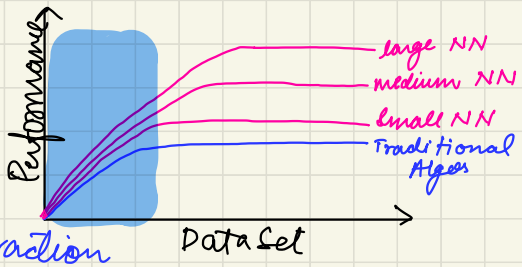
Traditionaly computers were unable to process unstructured data but that has changed with the introduction of Neural Networks.

* Neural Nets have also improved structural data understanding

Performance v/s Dataset Size.

* NN have become more and more prominent due to the rise in amounts of data as only they can give performance gains

For low amounts of data the relative ordering depends on feature extraction

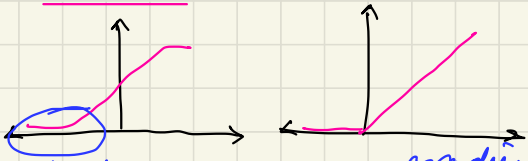


* Earlier the advancements is NN were based on Data & Computation advancements but now algorithmic transformations are all the rage.

↳ latest advancement

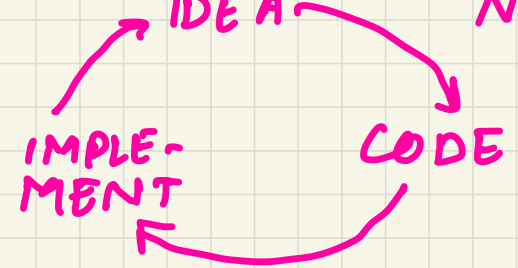
Replacement of Sigmoid with ReLU

Reason



gradient is very low in this region ∴ progress is very slow.

gradient is 1 ∴ gradient descent works faster



NN training is an iterative Problem

Faster training ⇒ more time for ideation

Binary Classification

Image Representation

3 channels for pixel values for Red, green and Blue.

For an image of size h x w, the dimensions are h x w x 3

They are stacked in a Column vector X for NN input.

$$X = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix}_{n \times 1}$$

For m training examples, X becomes.

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}_{n \times m}$$

i/p to a neural network

o/p is represented as.

$$Y = [y^{(1)} \; y^{(2)} \; y^{(3)} \cdots y^{(m)}]_{1 \times m}$$

Logistic Regression

Let $\hat{y} = P(x = 1 / x)$ (probability of $y = 1$)

Let W be a weight matrix of size n where n is the number of features

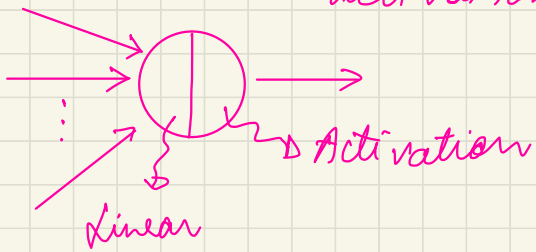
$$W = \mathbb{R}^{n \times 1}$$

Represents one node of a neural network.

$$\hat{y}^{(i)} = \sigma(W^T x^{(i)} + b)$$

σ: sigmoid; $W^T x^{(i)} + b$: linear relationship

NODE = Linear Relationship + Activation function



Now, Maximum likelihood estimation is

$$= \prod_{i=0}^{c} P(y=i|x) \qquad (c = \#\text{ of classes})$$

For Binary classification

$$MLE = P(y=0|x) \cdot P(y=1 \cdot x)$$

$$= (1-\hat{y})^{1-y} \cdot (\hat{y})^{y}$$

we need to maximize MLE for accurate results.

Taking log on both ends.

$$\log(MLE) = (1-y)\log(1-\hat{y}) + y\log(\hat{y})$$

We can also achieve max likelihood by maximizing log(MLE) or minimizing $[-\log(MLE)]$. So, $-\log(MLE)$ can be our cost

$$\alpha(\hat{y}^{(i)}, y^{(i)}) = -\left[(1-y)\log(1-\hat{y}) + y\log\hat{y}\right]$$

over the entire data set

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} \alpha(\hat{y}^{(i)}, y^{(i)})$$

Here, $J(W,b)$ is a convex function and hence can be reduced by gradient descent.

Gradient Descent

Aim: Take small steps to reach global min

Repeat {

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

(α: learning rate; $\frac{\partial J(w,b)}{\partial w_j}$: slope)

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

}

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

$$\frac{\partial \mathcal{L}(\hat{y}^{(i)}, y^{(i)})}{\partial w_j} = \frac{\partial \mathcal{L}(\hat{y}^{(i)}, y^{(i)})}{\partial \hat{y}^{(i)}} \cdot \frac{\partial \hat{y}^{(i)}}{\partial w_j}$$

$$\frac{\partial \mathcal{L}(\hat{y}^{(i)}, y^{(i)})}{\partial \hat{y}^{(i)}} = -y^{(i)} \frac{1}{\hat{y}^{(i)}} + (1-y^{(i)}) \frac{1}{(1-\hat{y}^{(i)})}$$

$$= \frac{\hat{y}^{(i)} - y^{(i)}}{\hat{y}^{(i)}(1-\hat{y}^{(i)})}$$

$$z = w^T x + b$$

$$\frac{\partial \hat{y}^{(i)}}{\partial w_j} = \frac{\partial \hat{y}^{(i)}}{\partial z^{(i)}} \cdot \frac{\partial z^{(i)}}{\partial w_j}$$

$$= \frac{-1}{(1+e^{-z^{(i)}})^2} \times e^{-z^{(i)}} \times -1 \cdot x_j^{(i)}$$

$$\frac{\partial \hat{y}^{(i)}}{\partial w_j} = \frac{e^{-z^{(i)}}}{(1+e^{-z^{(i)}})^2} \cdot x_j = \frac{(1+e^{-z^{(i)}}) - 1}{(1+e^{-z^{(i)}})^2} x_j^{(i)}$$

$$= \frac{1}{1+e^{-z^{(i)}}} \cdot \frac{(1+e^{-z^{(i)}}) - 1}{1+e^{-z^{(i)}}} \cdot x_j^{(i)}$$

$$= \hat{y}^{(i)} \cdot (1 - \hat{y}^{(i)}) \, x_j^{(i)}$$

$$\frac{\partial \mathcal{L}(\hat{y}^{(i)}, y^{(i)})}{\partial w_j} = \frac{\hat{y}^{(i)} - y^{(i)}}{\hat{y}^{(i)}(1-\hat{y}^{(i)})} \cdot x_j^{(i)} \cdot \hat{y}^{(i)}(1-\hat{y}^{(i)})$$

$$= [\hat{y}^{(i)} - y^{(i)}] \, x_j^{(i)}$$

$$\frac{\partial J(w,b)}{\partial w_j} = \frac{\sum_{i=1}^{m} [\hat{y}^{(i)} - y^{(i)}] \cdot x_j^{(i)}}{m}$$

VECTORIZATION

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}_{n \times m} \qquad W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}_{n \times 1}$$

$$Y = [y^{(1)} \; y^{(2)} \; \cdots \; y^{(m)}]_{1 \times m}$$

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

$$\hat{Y} = \sigma(W^T X + b)$$

$$\frac{\Delta J}{W} = X(\hat{y} - y)^T \qquad \frac{\Delta J}{B} = \Sigma(\hat{y} - y)$$

$$W = W - \frac{\alpha}{m} X\left(\sigma(W^T X + b) - Y\right)^T$$

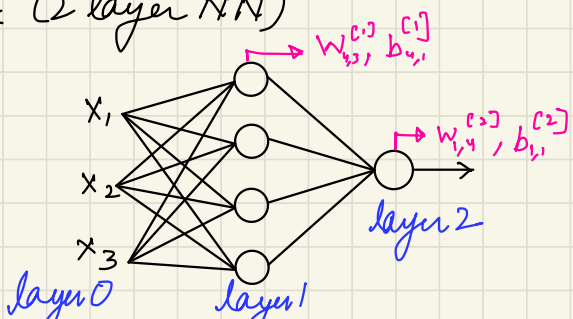
$$b = b - \frac{\alpha}{m} \Sigma\left(\sigma(W^T X + b) - Y\right)$$

Shallow Neural Networks

Superscript $[i] \rightarrow$ layer i

example (2 layer NN)

$a^{[0]} = X_{(n,m)}$



* dimensions of $W^{[i]} = (i_{o/p}, i_{i/p})$ for any layer i

* dimensions of $b^{[i]} = (i_{o/p}, 1)$ for any layer i

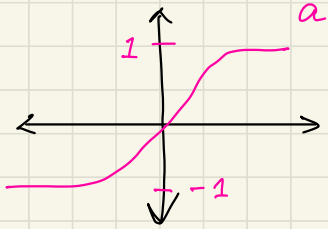
Let $g(z)$ be the activation function for the neural net so,

$$z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$$
$$a^{[1]} = g(z^{[1]})$$

$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$
$$a^{[2]} = g(z^{[2]})$$

Activation Functions

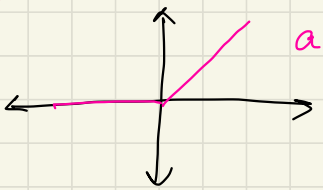
$a = \tanh(z)$



$a = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ (shifted sigmoid)

can be better for hidden layers as data is equally spread and mean is closer to 0.

$a = ReLU(z)$



$a = \max(0, z)$ learns faster

use when unsure about what to choose or want regression output

$a = LeakyReLU(z)$

$a = \max(0.01z, z)$ slightly better performance than ReLU

* use sigmoid for last layer only in case of Binary classification.

Why linear Functions are not used

Because multiple linear layers can be compressed to a single linear layer.

→ can be used as o/p layer with regression problems.

Derivatives

$g(z) = \frac{1}{1+e^{-z}}$ (sigmoid)

$$g'(z) = \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right) = g(z)(1 - g(z))$$

$g(z) = \tanh(z)$

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$g'(z) = \frac{(e^z + e^{-z})^2 - (e^z - e^{-z})^2}{(e^z + e^{-z})^2}$$

$$= 1 - (\tanh(z))^2$$

$$g'(z) = 1 - [g(z)]^2$$

$g(z) = ReLU(z)$

$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

(should be undefined at 0, but doesn't matter in software implementation)

Gradient Descent

Repeat {

$$dW^{[1]} = \frac{\partial J}{\partial W^{[1]}}, \quad db^{[1]} = \frac{\partial J}{\partial b^{[1]}}$$

$$dW^{[2]} = \frac{\partial J}{\partial W^{[2]}}, \quad db^{[2]} = \frac{\partial J}{\partial b^{[2]}}$$

$$W^{[1]} := W^{[1]} - \alpha dW^{[1]}$$
$$B^{[1]} := B^{[1]} - \alpha db^{[1]}$$

$$W^{[2]} := W^{[2]} - \alpha dW^{[2]}$$
$$B^{[2]} := B^{[2]} - \alpha db^{[2]}$$

}

CODE

Foward Propogation (Binary Classification)

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$
$$A^{[1]} = g(Z^{[1]})$$
$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$
$$A^{[2]} = g(Z^{[2]}) = \sigma(Z^{[2]})$$

Back Ward Propogation

$$dZ^{[2]} = A^{[2]} - Y$$
$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$
$$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis=1, keepdim=True)$$
$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]'}(Z^{[1]})$$

↖ element wise product

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$
$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis=1, keepdim=True)$$

Why weights should not be 0?

if all weights are zero hidden units in one layer are symmetric.

∴ initialize with small random numbers.

if W is too large, the activation is saturated.

Deep Neural Networks

L = # layers

$n^{[l]}$ = # units in layer l

$n^{[0]} = n_x$ $\quad$ $n^{[L]} = 1$

Generalized F. P.

$$Z^{[l]} = W^{[l]} A^{[l-1]} + B^{[l]}$$
$$A^{[l]} = g^{[l]}(Z^{[l]})$$

dimensions

$$W^{[l]} = (n^{[l]}, n^{[l-1]})$$
$$B^{[l]} = (n^{[l]}, 1)$$
$$A^{[l]} = (n^{[l]}, m)$$
$$Z^{[l]} = (n^{[l]}, m)$$

Why deep neural networks work?

example: For face detection earlier layers may detect edges, followed by edge combinations to detect facial features, followed by part of face detections and so on.

Deep Learning → Builds a system from primitive level to upwards toward complex

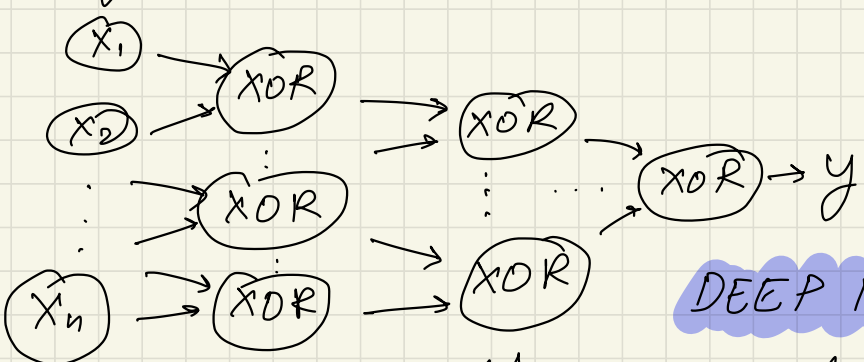
* DL are modelled against brain as brain does the same (starting from base to building up)

Circuit Theory

if a digital circuit is modelled using large NN, it will require less number of nodes per layer

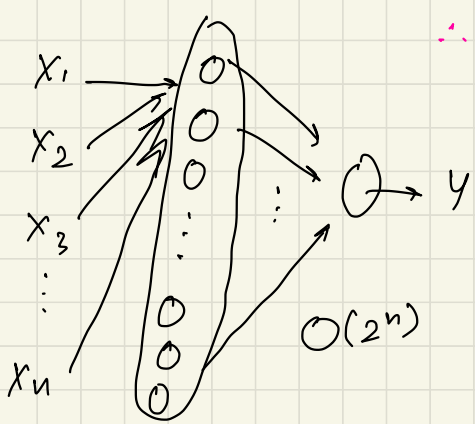
But if the NN is shallow, the number of nodes increases exponentially

eg $y = x_1 \text{ XOR } x_2 \text{ XOR } x_3 \cdots \text{XOR } x_n$



DEEP NETWORK

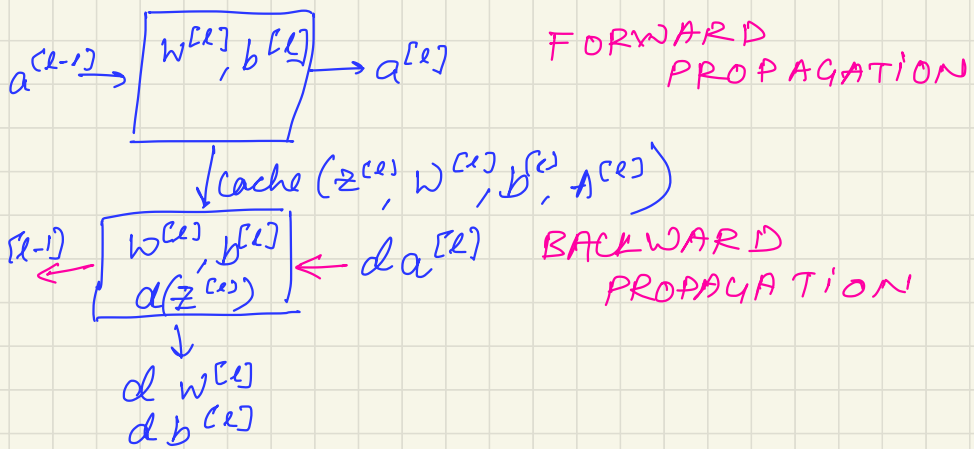
∴ DNN can learn more complex systems & functions easily



SHALLOW NETWORK

Propagation

For any layer l



Foward

$$Z^{[l]} = W^{[l]} \cdot A^{[l-1]} + B^{[l]}$$

$$A^{[l]} = g_{[l]}(Z^{[l]})$$

Backward

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

$*$ → element wise product

$$dW^{[l]} = \left[dZ^{[l]} \cdot (A^{[l-1]})^T\right] / m$$

$$db^{[l]} = \frac{1}{m} \text{np.sum}(dZ^{[l]}, \text{axis} = 1, \text{keepdims} = \text{True})$$

$$dA^{[l-1]} = (W^{[l]})^T \cdot dZ^{[l]}$$

FOR LAST LAYER WITH SIGMOID

$$dA^{[L]} = -Y / A^{[L]} + (1-Y) / (1 - A^{[L]})$$

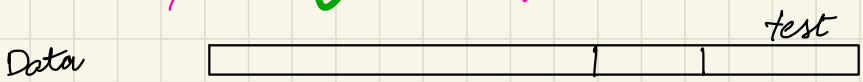
Hyperparameters

* learning rate
* # iterations
* # layers (L)
* hidden units $n_{[1]}, n_{[2]}, \cdots n_{[L-1]}$
* choice of activation function

Hyperparameters are the parameters that influence the model architecture and learning of actual computable parameters

Improving Deep Neural Nets

Train, Dev and Test Sets →



In case of a smaller data set
60-20-20 is a good breakup

- Hold out cross validation
- Development set
- "dev"

But as the data sets become larger we use smaller %

eg. For example in a dataset with 1,000,000 data points dev and test sets can just be 10,000 points as that is enough to get an idea of the system.

98-1-1

Dev, test sets should be from the same distribution as the training set, or else the results may be inaccurate.

if a non-biased evaluation of the model is not required its okay to not have a test set.

Train Set → training the parameter
Dev Set → Fine tuning the hyper parameters
Test Set → Gives an un-biased evaluation of the model.

Bias & Variance

High Bias ~ the model fit to the data is extremely simple. (model is biased towards simplicity)
[underfitting]

High Variance ~ the model fit to the data is extremely complicated. (model surrenders to the variance in data)
[overfitting]

Train Error ↓ & Dev Error ↑ → High Variance
Train Error ↑ & Dev Error ↑ → High Bias
Train Error ↑ & Dev Error ↑↑ → High Bias & Variance
Train Error ↓ & Dev Error ↓ → Low Bias & Variance

wrt to Baye's Error.

Basic Steps for DL optimization

① High Bias (visualize training data performance.
So!: Bigger Network
learn longer
Change NN architecture

② After Bias Reduction, Check for High Variance

Sol: More data
Regularization
Change NN architecture

Regularization

Logistic Regression

L2 Regularization :- Adding $\frac{\lambda}{2m}\|w\|_2^2$ to cost

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

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

L1 Regularization :- Adding $\frac{\lambda}{m}\|w\|_1$ to cost

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{m}\|w\|_1$$

$$\|w\|_1 = \sum_{j=1}^{n_x} |w_j|$$

* L1 regularization can lead to w being sparse.
⇒ Better in compressing a model.

$\lambda \rightarrow$ regularization parameter

Neural Network Regularization

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

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

↳ FORBENIUS NORM

Now,

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

↳ Reason why L2 is also known as weight decay

$$W^{[l]} = W^{[l]} - \alpha \left[\text{derivative from back prop} + \frac{\lambda}{m} W^{[l]} \right]$$

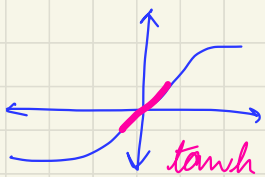
$$W^{[l]} = \left(1 - \frac{\alpha\lambda}{m}\right) W^{[l]} - \alpha(\text{derivative})$$

Why Regularization Reduces Overfitting?

with large enough λ, $W \approx 0$
some hidden units have less impact making model more general.

$$\lambda \uparrow \Rightarrow W^{[l]} \downarrow \Rightarrow z^{[l]} \downarrow \Rightarrow a^{[l]} \approx z^{[l]}$$
$$(z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]})$$

∴ layers are roughly linear
Hence, even deep models become somewhat simpler.



Dropout Regularization

Removing some nodes from each layer based on a dropout probability.

if keep-probability for a layer l = kp

```
dl = np.random.rand(al.shape[0], al.shape[1]) < kp
al = np.multiply(al, dl)   # sets some values to 0.
al /= kp                   # increases the impact by chance of node removal.
```

↳ evens out the node removal

eg if kp = 0.8
al is increased by 20%.

Different units should be removed at different iterations of gradient descent.

during test → dropout should be eliminated to avoid noise in results

division by KP makes sure test results aren't required to be scaled as dropout is missing in test.

Why dropout works?

since features to a NN node are eliminated at random, weight doesnot depend on any one feature
∴ weights are well spread out.
Has similar impact as L2.

Keep probability can be different for diff layers.

KP = 1 (for last layer)

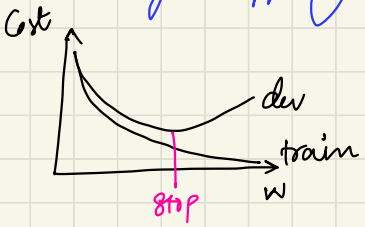
KP can also be used for inputs but the number should be close to 1.

Other ways to Regularize Data

Data Augmentation :→ modify data set slightly to create more data points

eg Flip images horizontally or zoom them a little
Distorting a little helps too

Early Stopping → Plot Train & Dev Set Costs. and stop training half way.



DISADVANTAGE → We compromize on Cost Optimization.
L2 Regularization is a better fit.

Normalization

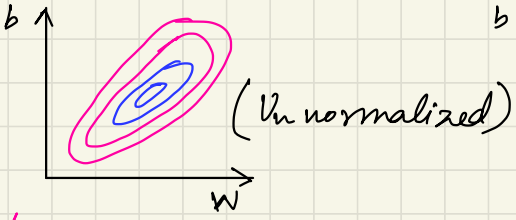
1. Modify for mean to be zero
2. Modify for variance normalization

(1) $\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$ (2) $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} **2$

$x = x - \mu \Rightarrow x = x/\sigma$

INTUITION

For normalized data the weights for each feature are within the same range which ensures that the cost function is symmetric to some extent. This means, larger learning rates can be chosen to train the system.



Vanishing / Exploding Gradients

The gradients either become too big or too small.

if $W > I$ then over the layers the gradient of $\hat{y}$ will explode to a large number for multiple layers

if $W < I$ the gradient will become quite small.

To rectify the vanishing / exploding gradient $Var = \frac{1}{n}$ for DNN and $\frac{2}{N}$ for ReLU layers weights

For non-ReLU layers

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

For ReLU layers

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

variance $\frac{2}{n^{[l-1]} * n^{[l]}}$ can also be used for tanh

Gradient Checking

① Concatenate $W^{[1]}, b^{[1]}, W^{[2]}, \ldots, W^{[L]}, b^{[L]}$ into a vector θ.
② Concatenate $dW^{[1]}, db^{[1]}, \ldots, dW^{[L]}, db^{[L]}$ into a vector $d\theta$.

$J(W^{[1]}, \ldots, b^{[L]}) = J(\theta)$

for each i:

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

$d\theta_{approx}[i] \approx d\theta[i]$

Check $\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2} \approx 10^{-7}$ for $\epsilon = 10^{-7}$ (10^{-5} is also good)

* Should not be used in training (only debugging)
* Remember to use regularization
* Doesn't work with dropout

Optimization Algorithms

mini-batch gradient descent

eg $X_{(n_x, m)} = [x^{(1)}\ x^{(2)}\ x^{(3)}\ \dots\ x^{(1000)} \mid x^{(1001)}\ \dots\ x^{(2000)} \mid \dots \mid \dots\ x^{(m)}]$

$X^{\{1\}}$ $X^{\{2\}}$ $X^{\{5000\}}$

$m = 5,000,000$
$batch\_size = 1000$

for $t = 1$ to 5000
  Forward Propagation $(X^{\{t\}})$
  BackWard Propagation $(X^{\{t\}}, Y^{\{t\}})$
  update parameters.

We basically run GD on smaller batches of dataset.

epoch → 1 pass through the entire dataset

batch size = m ⟶ Gradient Descent
batch size = 1 ⟶ Scholastic G.D.
(can be extremely noisy)
(we loose the progress by vectorization)

Optimal batch size is somewhat noisy as well but vectorization can be used to reduce time. → Typically a power of 2.
$2^6, 2^7, 2^8, \dots$

Exponentially Weighted Average

moving average of data over a time period

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

⇒ if $\beta = .9$ (averaging over last 10 values)
$\beta = .98$ (averaging over last 50 values)

Bias Correction

we start from $V_0 = 0$
due to which, the initial values are biased towards 0.
∴ for initial values
Corrected weighted avg $= \frac{V_t}{1-\beta^t}$

Gradient Descent with momentum

① Compute dW, dB on current mini batch
② $V_{dW} = \beta V_{dW} + (1-\beta) dW$
$V_{dB} = \beta V_{dB} + (1-\beta) dB$

③ $W = W - \alpha V_{dW}$; $B = B - \alpha V_{dB}$

*$\beta = 0.9$ works best mostly but can also be tuned

Root Mean Square Prop (RMS Prop)

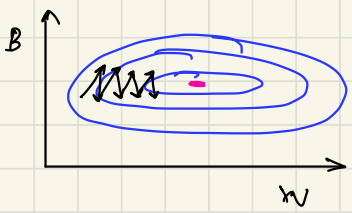
① Compute dW, db on the current mini-batch
② $S_{dW} = \beta S_{dW} + (1-\beta) dW^2$ ← element wise
$S_{dB} = \beta S_{dB} + (1-\beta) dB^2$ ← element wise

③ $W = W - \alpha \frac{dW}{\sqrt{S_{dW}}}$; $B = B - \alpha \frac{dW}{\sqrt{S_{dW}}}$

Intuition

(Horizontal) dw^2 ← small ⟶ dividing by avg increases horizontal shifting of the gradient

(Vertical) db^2 ← large ⟶ decreases vertical movement



* element with larger sq. in general gets dampened and that with smaller values are scaled to ensure equal movement in both directions

α can be large as the chances of movement getting skewed are decreased.

We add $\epsilon = 10^{-8}$ to the denominator to ensure non-zero denominators

$$w = w - \alpha \frac{dw}{\sqrt{S_{dw}} + \epsilon} \; ; \; b = b - \alpha \frac{db}{\sqrt{S_{db}} + \epsilon}$$

Adam Optimization (Momentum + RMS Prop)

On iteration t:

① compute dw, db using current mini-batch

② $V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw$; $V_{db} = \beta_1 V_{db} + (1-\beta_1) db$

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

③ $V_{dw}^{corrected} = V_{dw} / (1-\beta_1^t)$; $V_{db}^{Corrected} = V_{db} / (1-\beta_1^t)$

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

(4)

$$W = W - \alpha \frac{V_{dw}^{corr}}{\sqrt{S_{dw}^{corr}} + \epsilon} \quad ; \quad b = b - \alpha \frac{V_{db}^{corr}}{\sqrt{S_{db}^{corr}} + \epsilon}$$

Hyperparameters (by Adam Paper)

$\alpha \rightarrow$ tuned
$\beta_1 \rightarrow 0.9$
$\beta_2 \rightarrow 0.999$
$\epsilon \rightarrow 10^{-8}$

AdaM → Adaptive Moment estimation

Learning Rate Decay

as learning reaches closer to minimum, smaller learning rate helps us achieve results faster rather than oscillating about the solution.

$$\alpha = \frac{1}{1 + \text{decay\_rate} * \text{epoch\_num}} \alpha_0$$

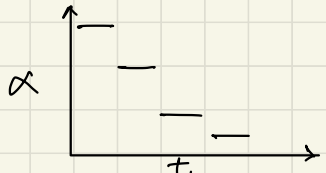
Exponential decay

$$\alpha = (0.95)^{\text{epoch}} \alpha_0$$

Sqroot decay

$$\alpha = \frac{k}{\sqrt{\text{epoch\_num}}} \cdot \alpha_0$$

discrete staircase.



Local Optima

in lower dimensional spaces, local minimas have 0 derivatives.
But for high dimensional spaces, saddle points may have 0 derivatives

Plateau

learning (der) ≈ 0 for a large time

↳ RMS prop, Momentum and Adam can help.

Hyper Parameter Tuning

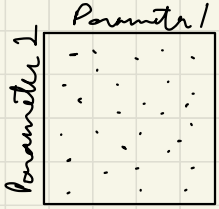
α → most important

β
#hidden layer units } → some what important.
mini-batch size

#layer
learning rate decay } → least important among the lot.

How to:

1. choose a region
2. Pick random points in that region
3. Create models for those points
4. If some points in a sub-region are better performers, choose that region and increase point density.
5. Repeat 3.



Scale

linear scale → α, # hidden units

logarithmic scale → α (0.0001 → 1)

```
r = -4 * np.random.rand()
α = 10^r
```

Sample $(1-\beta)$ on logarithmic range. (0.9 → 0.999)

```
r = (-3, -1)
β = 1 - 10^r
```

PANDA APPROACH → Train 1 model and modify hyperparameters each day

CAVIAR APPROACH → Train multiple models in parallel and then select the best model

Batch Normalization

normalizing $Z^{(l)}$ in hidden layers to make the model train faster for $W^{(l)}$, $B^{(l)}$

$$\mu = \frac{1}{m} \sum_i z^{(i)}$$

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

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

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

(γ, β ← learnable parameters)

Use $\tilde{z}^{(i)}$ instead of $z^{(i)}$

$X \xrightarrow{W^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \longrightarrow a^{[1]} \xrightarrow{W^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[BN]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \longrightarrow a^{[2]} \ldots$

parameters

$W^{[1]}, \cancel{b^{[1]}}, W^{[2]}, \cancel{b^{[2]}}, \ldots$

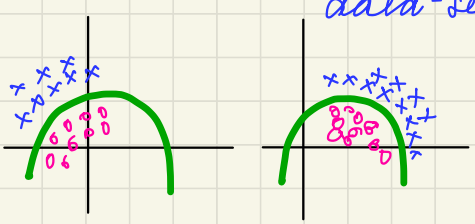
$\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \ldots$

we can eliminate $b^{[l]}$ as batch normalization zeros out the mean

$$z^{[l]} = W^{[l]} a^{[l-1]}$$

β and γ can also be updated using GD, GD with momentum or Adam.

Covariate Shift → the ground truth stays the same but the distribution of data-set changes



Batch Normalization stabalizes the layer laters' variance and mean which ensures that even if the distribution changes for the first layer, the model still adapts to it.

Batch norm Solves Covariate Shift

* Batch norm has slight regularization effects as mean & variance are computed per min-batch.

Testing

① Compute μ and σ^2 using exponentially weighted avg during the training process for all mini batches.

② Use this avg value during test as values of σ^2 and μ.

Multiclass classification

$C \rightarrow$ # of classes

$n^{[L]} = C$

Last layer uses softmax to implement this

Activation Function

$$t = e^{(z^{[L]})}$$

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{i=1}^{C} t_i}$$

$$a_i^{[L]} = \frac{t_i}{\sum_{j=1}^{n} t_j}$$

Loss function

$$\mathcal{L}(\hat{y}, y) = \sum_{j=1}^{C} y_j \log \hat{y}_j$$

Tensor Flow:

① Define variables tf.variable(0, dtype)

② Initialize gradient tape (records forward prop)

③ Set trainable variables list

④ compute gradients tape.gradient(cost, trainable_v)

⑤ optimizer.apply_gradients(zip(grad, trainable_va))

for one step of training.

OR

1. Define variables
2. Define optimizer — tf.keras.optimizers.Adam(0.1)
3. define cost-function()
4. call optimizer.minimize(cost-function, [variables]) for one step of training.