A quality product by
Santi Paper Products
Plt no 358/477, Tamando,
Bhubaneswar,751030
Feedbacks and queries :
santipaperproducts16@gmail.com

PUT WASTE IN THE RIGHT PLACE

Size : 297X210 mm
MRP : ₹45.00
(Incl. of all taxes)
Exercise Book
140 Pages
(Total Pages Include Index & Printed Information)

029701140
8 906060 291933

Type of Ruling:
Single Line

SMART WRITE
writing is my passion

Notebook

INDEX

ML

NAME : Kush Pandya. SUB.: ____________

STD.: ____________ DIV. ____________ ROLL NO. ____________

S. NO.	DATE	TITLE	PAGE NO.	SIGN./REMARKS

S. NO.	DATE	TITLE	PAGE NO.	SIGN./REMARKS

Page :
Date :

(x, y) $x \in \mathbb{R}^{n_x}$ → a single column vector of n features.

$y \in \{0, 1\}$

m training example : $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots (x^{(m)}, y^{(m)})\}$

$$X = \begin{bmatrix} | & | & | & \\ x^{(1)} & x^{(2)} & x^{(3)} & \cdots \\ | & | & | & \end{bmatrix}$$

n_x ↕

m ↔

$n_x \times m$

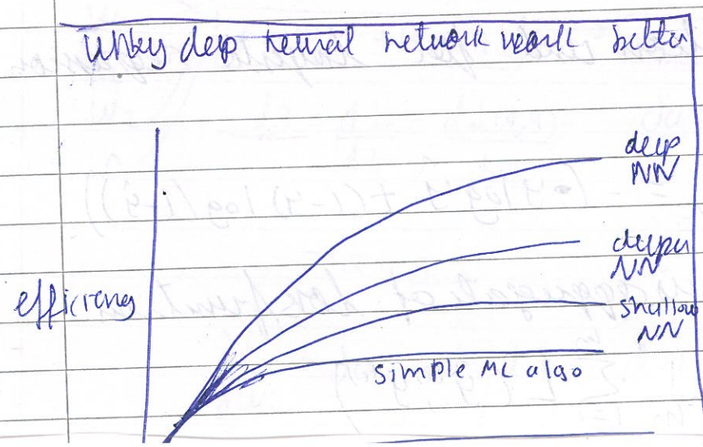
$X \subseteq \mathbb{R}^{n_x \times m}$

$Y = [y^{(1)}, y^{(2)} \ldots \ldots, y^{(m)}]$.

$y \in \mathbb{R}^{1 \times m}$

$y.shape = (1, m)$.

When deep neural network would better



logistic regression.

Given x want $\hat{y} = P(y=1|x)$.

$x \in \mathbb{R}^{n_x}$ $b \in \mathbb{R}$.

~~Parameter: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$.~~ Parameter $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

for linear regression → $w^T x + b$. →

for logistic regression.

output $\hat{y} = \sigma(w^T x + b)$ →

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

$\hat{y} = \sigma(w^T x + b)$ where $\sigma(z) = \frac{1}{1+e^{-z}}$

In logistic regression we do not use R^2 method. Since in gradient descent it gives multiple minimas. Which becomes hard to find the least minimum.

The loss function used for logistic regression

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

Cost function is aggregate of loss function.

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

Gradient ~~Descent method~~.

Gradient descent method.

$$w = w - \alpha \frac{dJ(w)}{dw} \} \rightarrow dw.$$

$$\Rightarrow w = w - \alpha dw.$$

and another one for b.

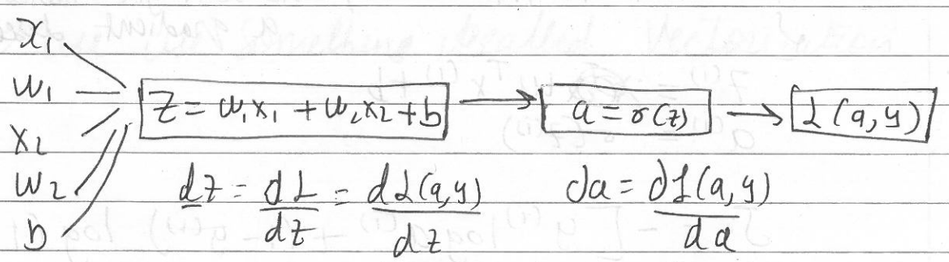
Naming convention.

d Final outvar → dvar
d var

↓

any previous node variable.

logistic regression derivatives.



$$dz = \frac{dL}{dz} = \frac{dL(a,y)}{dz} \qquad da = \frac{dL(a,y)}{da} = -\frac{y}{a} + \frac{1-y}{1-a}.$$

$$= a - y.$$

$$\frac{\partial L}{\partial w_1} = dw_1 = x_1 dz \qquad dw_2 = x_2 dz \qquad db = dz$$

Page :
Date :

logistic regression gradient descent.

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

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

$$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \underbrace{\frac{\partial L(a^{(i)}, y^{(i)})}{\partial w_i}}_{dw_1^{(i)}}$$

lets initailize

$J=0$, $dw_1=0$, $dw_2=0$, $db=0$

for $i=1$ to m // It's a single iteration of a gradient descent.

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J += -[y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log(1-a^{(i)})]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw_1 += x_1^{(i)} dz^{(i)}$ } n=2
$dw_2 += x_2^{(i)} dz^{(i)}$

Page :
Date :

$db += dz^{(i)}$

}

$J /= m$
$dw_1 /= m$
$dw_2 /= m$
$db /= m$

So here we have done this recursion.

$w_1 := w_1 - \alpha dw_1$
$w_2 := w_2 - \alpha dw_2$
$b := b - \alpha db$.

Since to implement the above code we will need so many loop's that will make our program less efficient.

So we use something called Vectorization.

Page :
Date :

Vectorization

Just get rid of for loop.

$z = np.dot(w, x) + b$

$w = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$ $x = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$ $b = [\,]$

avoid explicit for loop.

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

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

$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$ (n_x rows, m columns)

$w^T = \begin{bmatrix} w_1 & w_2 & \cdots & w_m \end{bmatrix}$

$Z = [z^{(1)}\ z^{(2)}\ \cdots\ z^{(m)}] = w^T X + [b\ b\ \cdots\ b]$

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

$z = np.dot(w^T, x) + b$ → a simple no

Page :
Date :

$A = [a^{(1)}\ a^{(2)} \cdots a^{(m)}] = \sigma(z)$

$Y = [y^{(1)} \cdots y^{(m)}]$

$dz = A - Y = [a^{(1)} - y^{(1)}\ a^{(2)} - y^{(2)} \cdots]$

$dz = [dz^{(1)}\ dz^{(2)} \cdots dz^{(m)}]$

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

$dw = \frac{1}{m} X dz^T$

The final implementation of logistic regression.

Put this in a for loop if you want to carry multiple iteration.

$z = w^T X + b$
$= np.dot(w^T, x) + b.$

Note. this is for 1 interation of gradient descent

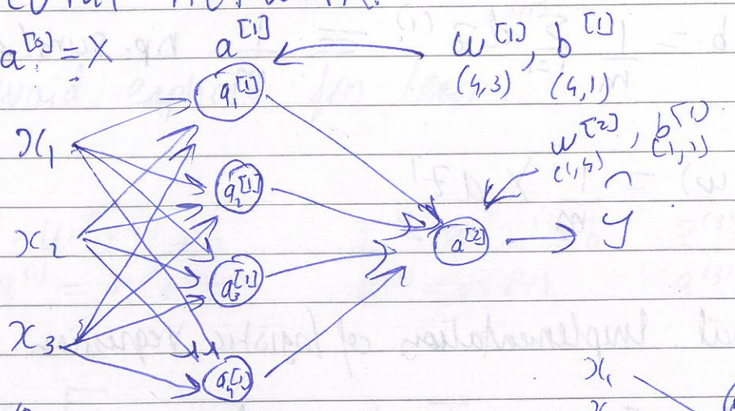
$A = \sigma(z)$
$dz = A - y$
$dw = \frac{1}{m} X dz^T$
$db = \frac{1}{m} np.sum(dz)$
$w = w - \alpha dw$
$b = b - \alpha db.$

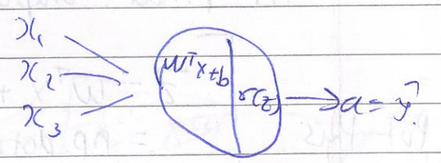
Page :
Date :

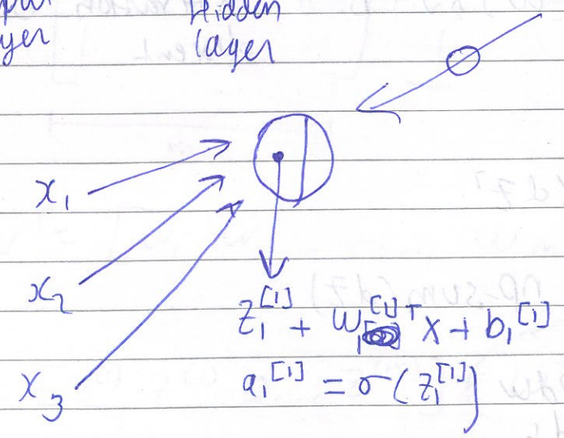
Broadcasting in python.

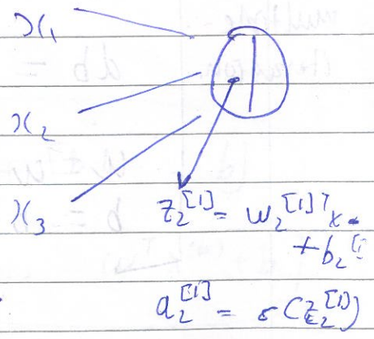
```
cal = A.sum(axis=0)
Percentage = 100*A/
```

Neural network.









Page :
Date :

Remember $A^{[0]}$ is also called Input.

Vectorisation.

$$z^{[1]} = \begin{bmatrix} — w_1^{[1]T} — \\ — w_2^{[1]T} — \\ — w_3^{[1]T} — \\ — w_4^{[1]T} — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} + b_1^{[1]} \\ w_2^{[1]T} + b_2^{[1]} \\ w_3^{[1]T} + b_3^{[1]} \\ w_4^{[1]T} + b_4^{[1]} \end{bmatrix}$$

$W^{[1]}$ $\quad$ $b^{[1]}$ $\quad$ $= \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = z^{[1]}$

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ \vdots \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]})$$

(i.e) $z^{[1]} = W^{[1]} x + b^{[1]}$
$a^{[1]} = \sigma(z^{[1]})$
$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$
$a^{[2]} = \sigma(z^{[2]})$.

In short. For the NN a Page before.

$x \longrightarrow a^{[2]} = \hat{y}$.
$x^{(1)} \longrightarrow a^{[2](1)} = \hat{y}^{(1)}$
$x^{(2)} \longrightarrow a^{[2](2)} = \hat{y}^{(2)}$
$x^{(m)} \longrightarrow a^{[2](m)} = \hat{y}^{(m)}$

$a^{[2](i)}$ — layer, example.

Page :
Date :

Un vectorized method

for $i=1$ to m;

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

Vectorization

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \quad (n_x, m)$$

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

} you may also call it forward Propagation.

$$Z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix}$$

$$= \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$$

Page :
Date :

If you need to understand more in details see video week 3 Justification for vectorized implementation.

Activation Func".

$s = \text{Sigmoid} = \dfrac{1}{1+e^{-z}}$ $\quad 0 \leq s \leq 1$

$t = \tanh = \dfrac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$ $\quad -1 \leq t \leq 1$

↓

It's better than sigmoid → The means of the f^h comes more closer to 0.

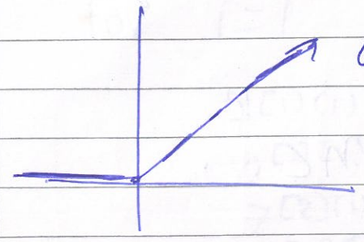
It centers your data.

It mostly strictly superior than sigmoid.

Use Sigmoid in output layer of binary classification.

The disadvantage of sigmoid and tanh if ~~is large~~ the input is large or small then the gradient derivative or slope of this function becomes very small. Which takes time getting the perfect descent.

R → Relu. → Rectified linear unit.



It's ~~deee~~ derivative is zero so it takes less time time.

Gradient descent for neural Network.

Parameters $w^{[1]}$ $(n^{[1]}, n^{[0]})$, $b^{[1]}$ $(n^{[1]}, 1)$, $w^{[2]}$ $(n^{[2]}, n^{[1]})$, $b^{[2]}$ $(n^{[2]}, 1)$.

$n_x = n_0$.

cost fn: $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{n} L(\hat{y}, y)$ ↑ $a^{[2]}$

Gradient descent.

Repeat {

Compute predict ($\hat{y}^{[i]}$, $i = 1 \dots m$)

$dw^{[1]} = \frac{dJ}{dw^{[1]}}$ $db^{[1]} = \frac{dJ}{db^{[1]}}$, ...

$w^{[1]} = w^{[1]} - \alpha dw^{[1]}$

$b^{[1]} = b^{[1]} - \alpha dw^{[1]}$

⋮

} ← backpropagation.

Back propagation. (for binary classification)

$dz^{[2]} = A^{[2]} - Y$

$dw^{[2]} = \frac{1}{m} dz^{[2]} A^{[1]T}$

$db^{[2]} = \frac{1}{m}$ np.sum($dz^{[2]}$, axis = 1, keepdims = True)

$dz^{[1]} = w^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$ ← any activation fn derivation

↑ element wise product.

$dw^{[1]} = \frac{1}{m} dz^{[1]} X^T$

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

Page :
Date :

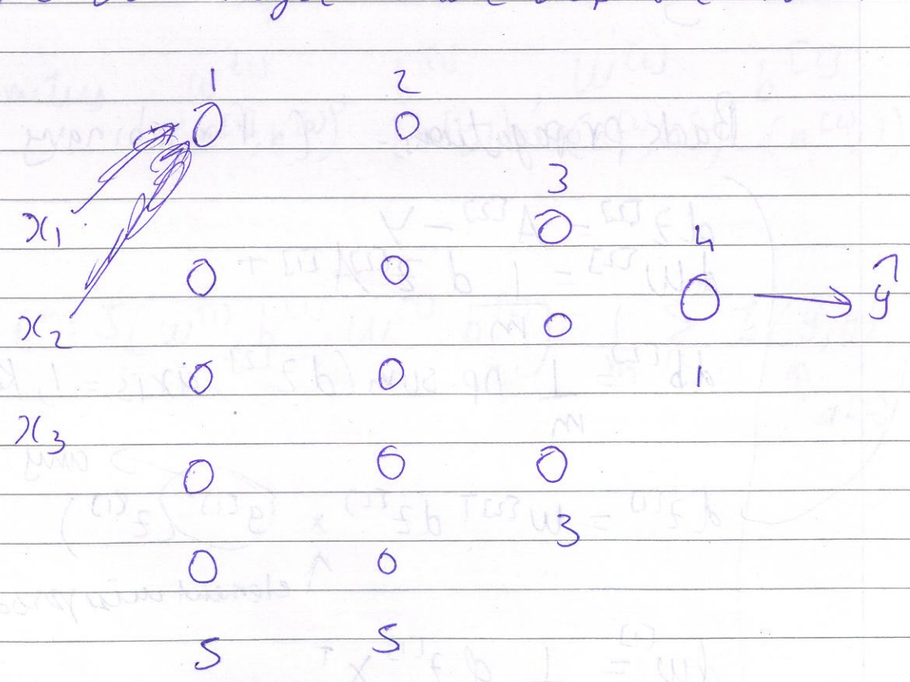
Deep neural Networks

$$\begin{cases} z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]} \\ a^{[l]} = g^{[l]}(z^{[l]}) \end{cases}$$

→ The basic of forward propogation

Deepness of NN.

More the layer, more deep the NN.



L = no of layer = 4

$n^{[l]}$ = no of units in layer l =

$n^{[1]}$ = $n^{[2]}$ = $n^{[3]}$ =

Page :
Date :

$a^{[l]}$ = activation in layer L.

$a^{[l]} = g^{[l]}(z^{[l]})$

$w^{[l]}$ = weight for $z^{[l]}$

$$X : z^{[1]} = w^{[1]} x + b^{[1]}$$

$$a^{[1]} = g^{[1]}(z^{[1]})$$

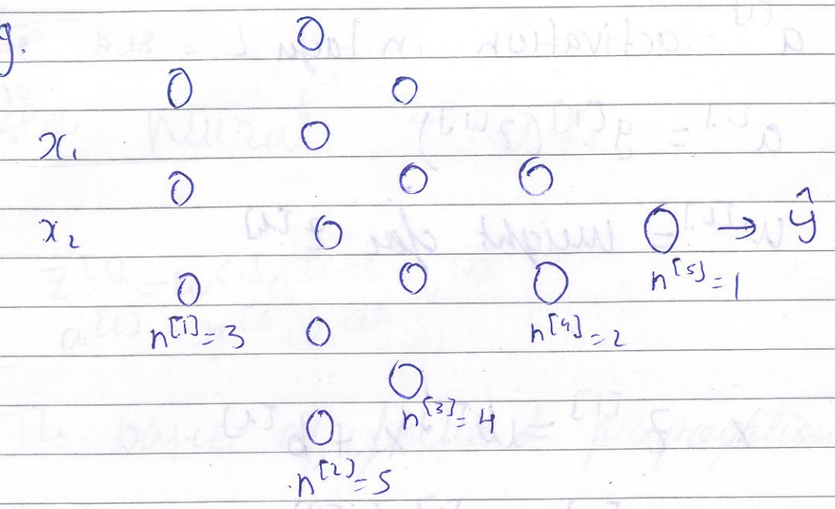
$$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$$

$$a^{[2]} = g^{[2]}(z^{[2]})$$

$$z^{[4]} = w^{[4]} a^{[3]} + b^{[4]}$$

$$a^{[4]} = g^{[4]}(z^{[4]}) = \hat{y}.$$

e.g.



$$z^{[1]} = w^{[1]} x + b^{[1]}$$

$(3,1) \quad (2,1)$

$(n^{[1]},1) \quad (n^{[0]},1)$

$$\begin{bmatrix} \cdot \\ \cdot \end{bmatrix} = \begin{bmatrix} \cdot & \cdot \\ \cdot & \cdot \end{bmatrix} \begin{bmatrix} \cdot \\ \cdot \end{bmatrix}$$

$$\text{dimension}(w^{[l]}) = (n^{[l]}, n^{[l-1]})$$

$$\text{dimension}(b^{[l]}) = (n^{[l]}, 1)$$

$$\text{dim}(dw^{[l]}) = (n^{[l]}, n^{[l-1]})$$

$$\text{dim}(db^{[l]}) = (n^{[l]}, 1)$$

One reason why we can't have a single layer of NN which has to many units instead of ~~deep~~ deep NN. is because, If we want both of this model to be equally efficient the shallow NN ~~would~~ should have exponentially large units Compared to input efficiency. And deep neural network would have a logrithmic efficiency (not sure).

Forward function and backward fn

layer → l.

Forward fn will require $w^{[l]}, b^{[l]}$

input → $a^{[l-1]}$

output → $a^{[l]}$

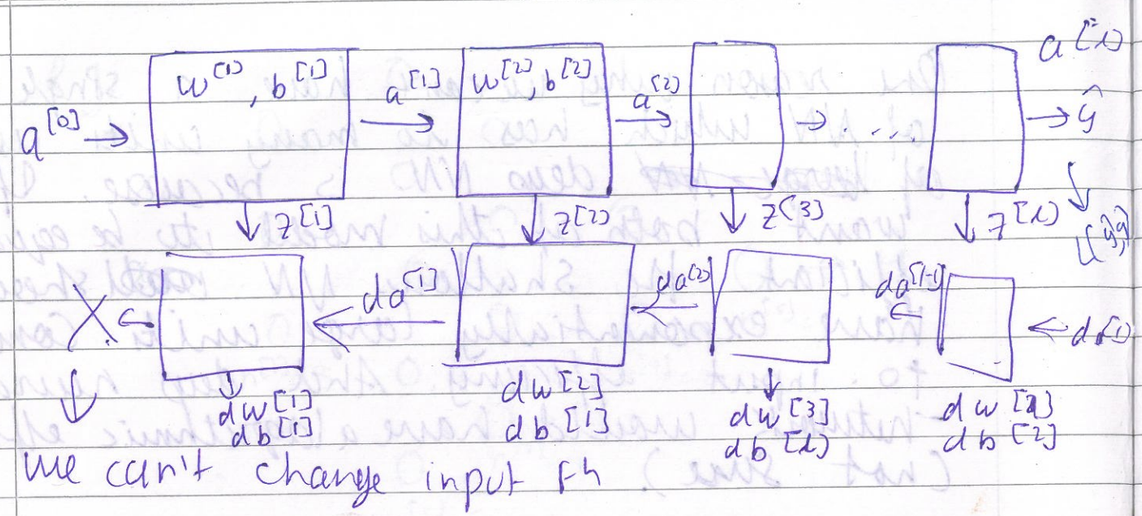
It will also cache(store) $z^{[l]}$

Back ward fn will require

input → $da^{[l]}$, ~~output~~

output → $da^{[l-1]}$, ~~dw~~ after this $dw^{[l]}, db^{[l]}$

It will



We can't change input fn

Forward fn

$$z^{[1]} = w^{[1]} a^{[1-1]} + b^{[1]}$$
$$a^{[1]} = g^{[1]}(z^{[1]})$$

backward fn

$$w^{[1]} = w^{[1]} - \alpha dw^{[1]}$$
$$b^{[1]} = b^{[1]} - \alpha db^{[1]}$$

Backward propagation for layer l

Steps.

$$dz^{[l]} = da^{[l]} * g^{[l]'}(z^{[l]})$$
$$dw^{[l]} = dz^{[l]} \cdot a^{[l-1]}$$
$$db^{[l]} = dz^{[l]}$$
$$da^{[l-1]} = w^{[l]T} \cdot dz^{[l]}$$

$$dz^{[l]} = dA^{[l]} * g^{[l]'}(z^{[l]})$$
$$dw^{[l]} = \frac{1}{m} dz^{[l]} \cdot A^{[l-1]T}$$
$$db^{[l]} = \frac{1}{m} np.sum(dz^{[l]}, axis=1, keepdims=True)$$
$$dA^{[l-1]} = w^{[l]T} \cdot dz^{[l]}$$

Parameters and hyperparameters

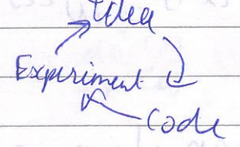
Parameter $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, w^{[3]}, b^{[3]}$

Hyperparameterisation → learning rate
→ iteration
→ hidden layer
→ hidden units $n^{[1]}, n^{[2]}$
→ choice of activation fn.

ie → momenteum, minibatch, regulaleion

Course - 2

We have to identify the this thing before making neural networks.
- layers
- hidden units
- learning rates
- activation function.



If you have small dataset try 70/30 train test split.

But if you have larger data set → The size of test can be reduced drastically.

Basic Recipe.

	Train set error	Dev set error
high Var	low	high
high bias	high	high (comparable to train set)
both bias & Var	high	super high

Regularization

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

omit.

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

L_2 regularization (lasso regres).

L_1 regularization $\frac{\lambda}{2m}\sum_{i=1}^{n_x}|w| = \frac{\lambda}{2m}\|w\|_1$ ↓ mostly is sparsed.

L_2 is used most often.

λ = regularization parameter.

In Neural Network.

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

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

"Frobenius norm"

Changes in backprop due to $dw^{[l]}$

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

$w^{[l]} = w^{[l]}$

$w^{[l]} = w^{[l]} - \alpha dw^{[l]}$

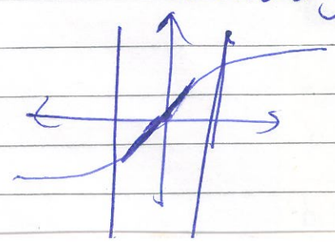
$\Rightarrow w^{[l]} = w^{[l]} - \alpha \left[(\text{from backprop}) + \frac{\lambda}{m} w^{[l]}\right]$

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

How does regularization help in reducing overfitting in NN.

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

So if $\lambda \uparrow$ $\therefore$ $w^{[l]}$ which is close to zero. in starting does not move much as λ amplifies it effect. We can see in graph below. that λ would remain in the linear region and $\therefore$ The NN will not be able make complecated decision boundary.



Dropout regularization

Inverted dropout

Here Illustrate with layer $l=3$ keep prob $= 0.8$

$d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep\_prob$

$a3 = np.multiply(a3, d3)$

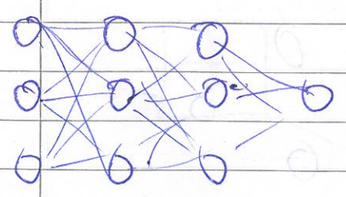
$a3 /= keep\_prob$

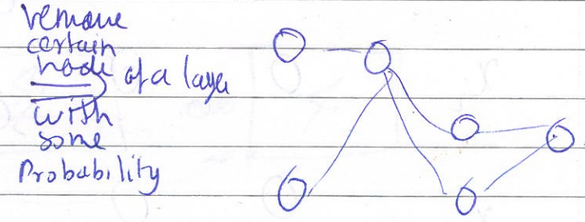
$z^{[4]} = w^{[4]} a^{[3]} + b^{[4]}$

↑ reduced by 20%. (assumed by above operations) So to make it expected by what it we bump it

At keep a_3 expected value same

What is dropout.





Page
Date

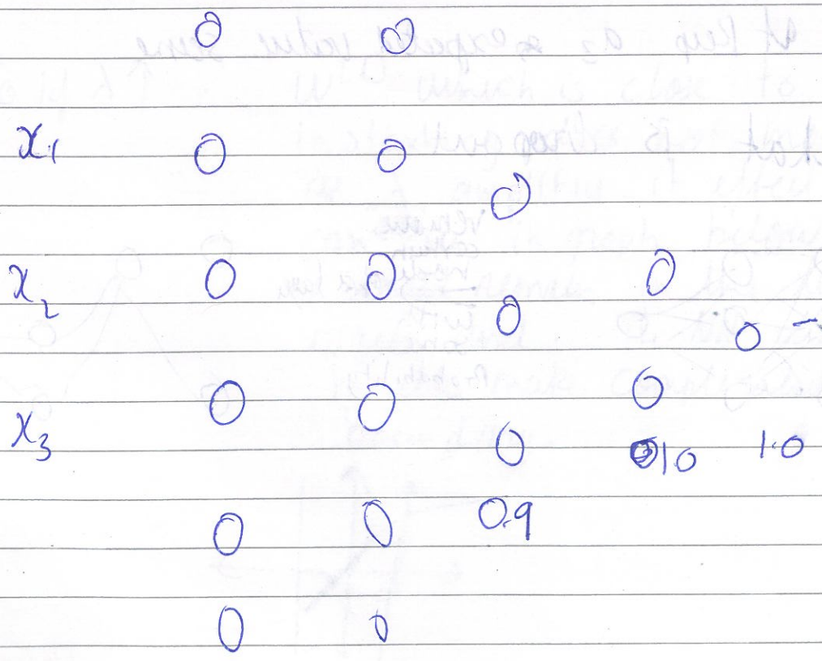
iteration during
↑
☆ For different training examples you zero out different hidden unit.

For making prediction.

We don't use drop out during that time. Since it will add noise to your prediction.

Intuition

If dropward prop a particular weight is large compared to other weights in that layer. The dropout will have a effect on it and reduce its influence. increasing other. But it will have a overall effect of shrinking it.



Page :
Date :

Keep prob can be different for different layers just remember that with larger layers (more nodes) we keep the "keep prob" low 3. while with small layer we keep them high.

We can put keep prob in Input layer but its value should be very high.

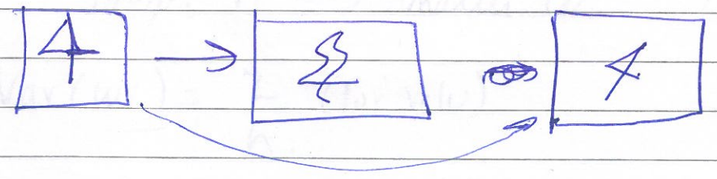
Disadvantage.

5- loss functions. (loosely defined).

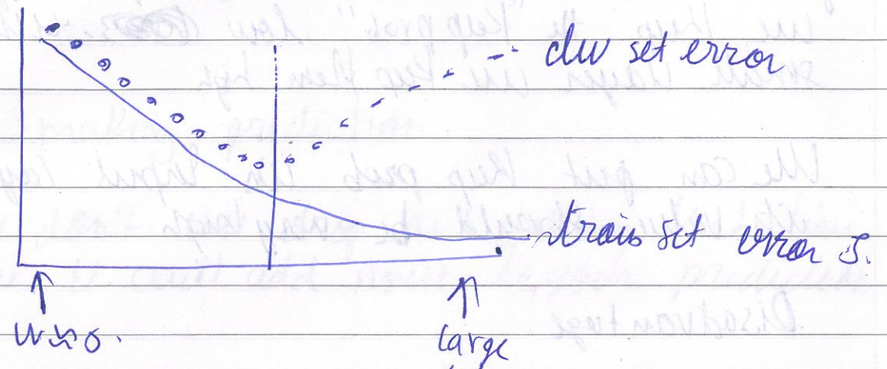
Data augmentation

Suppose we wants to make cat classifier but we don't have enough data. One thing we can do is we can flip the image. Take some random zoom ins. etc.

For digits we can put random rotation and distortions to it.



Early stoping



We just stop training the NN after some point. We can check whether the dev set error is decreasing or increasing

Normalising.

Importance → help your learning algorithm learn faster.

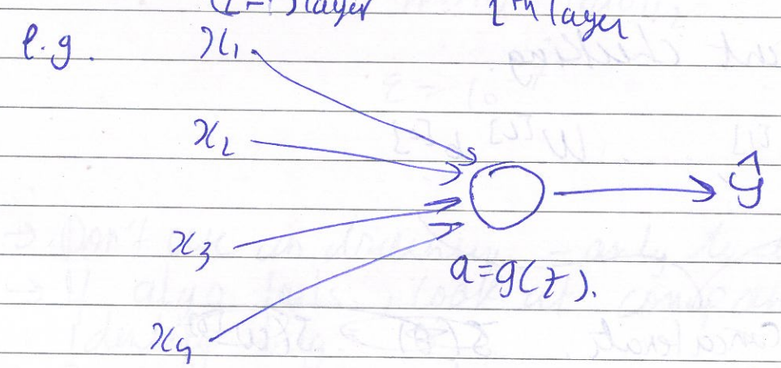
Meaning converting ~~mean and~~ variance of ~~each~~ of input attribute to same order of magnitude

Vanishing / exploding gradient

See video ~~for~~ for further explanation.

We see that if a very deep neural network starts working. Then we see that sometimes their is a vanishing and exploding arguments that really slows down the learning rate. ~~for the complementation~~

To avoid this problem we try to initialize weight more carefully.



$z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n$

larger n → smaller w_i (to reduce H_L z based on previous layer size)

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

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

Other Activation

tanh use $\sqrt{\frac{1}{n^{[l-1]}}}$

Xavier initialization

also $\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$

Numeric approximation of gradient.

Gradient checking.

Take $W^{[1]}, b^{[1]} \ldots W^{[L]}, b^{[L]}$

Concatenate. ~~$J(\theta) \to J(W^{[1]}$~~,

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

Take $dW^{[1]}, db^{[1]} \ldots dW^{[L]}, db^{[L]}$ and reshape into a big vector $d\theta$.

~~to do the grads~~

$J(dW^{[1]}, db^{[1]} \ldots) = J(d\theta)$

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

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

check $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$

↳ 10^{-7} - great

10^{-5} - OK (check component)

10^{-3} - wrong.

$\varepsilon = 10^{-7}$

→ Don't use in training - only to debug.
→ If algo fails., look at components to try to identify bug.
→ Remember regularization
→ Doesn't work with dropout
→ Run at random initialization: perhaps again after some training

Page :
Date :

Mini batch gradient descent.

$$X_{(n_x, m)} = \left[\underbrace{x^{[1]} \; x^{[2]} \; x^{[3]} \; \ldots}_{X^{\{1\}}} \; \underbrace{\ldots \; \ldots}_{X^{\{2\}}} \; \ldots \; x^{[m]} \right]$$

$$Y_{(1, m)} = \left[y^{[1]} \; y^{[2]} \; \ldots \; \ldots \; y^{[m]} \right]$$

Size of mini batch. $\rightarrow m \rightarrow$ batch learning (as previous)
$\rightarrow 1 \rightarrow$ Stochastic gradient descent.

Stochastic	In between (mini batch size not too small / large)	Batch gradient (mini batch gradient)
↓	↓	↓
less speedup	• Faster learning	high time.
lose vectorization advantage.	• Vectorization	
as 1 is being Processed at a time.	• Make progress without ~~Progress On~~ the whole training data.	

Choosing mini batch size.

If small trainset : Use batch gradient descent.
($m \leq 2000$)

Typical mini batch size
64, 128, 256, 512
2^6 2^7 2^8 2^9

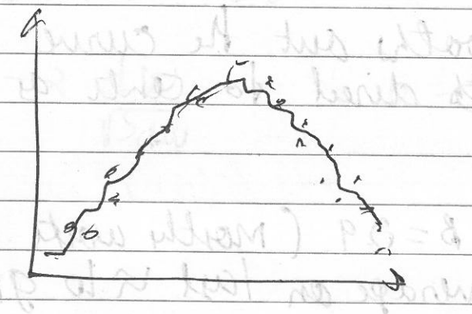
Page :
Date :

Make sure your mini batch fits in CPU/GPU memory

Exponentially weighted averages.

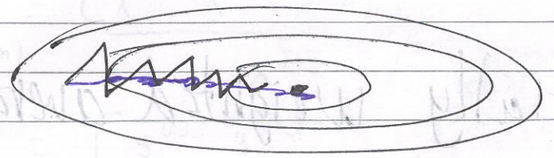
$\theta_1 = 40°F$
$\theta_2 = 49°F$

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t \quad / \quad V_0 = 0 \; (\text{const})$$



Page :
Date :

Gradient descent with momentum.



↕ slower learning
↔ Faster learning

Momentum
on iteration t
Compute dw, db on wrt mini-batch

initialise ~~0~~.
$V_{db}; V_{dw} = 0$ $V_{dw} = \beta V_{dw} + (1-\beta) dW$ } changing V_{dw}, V_{dB}
$V_{db} = \beta V_{db} + (1-\beta) db$ }

$w = w - \alpha V_{dw}$, $b = b - \alpha db$. } → remains same.

It smooths out the curve and takes a ~~smooth~~ direct to centre as shown in blue pen.

$\beta = 0.9$. (Mostly works well)
average on last ≒ 10 gradients

Sometime ~~the~~ the above 2 equation is sometime written

$V_{dw} = \beta V_{dw} + dW$
$V_{dB} = \beta V_{db} + db$

but don't use this one.

Page :
Date :

gradient descent with momentum always works better than regular gradient descent.

RMSprop

on iteration t.

compute dw, db on the current mini batch.

$S_{dw} = \beta S_{dw} + (1-\beta) \underbrace{dw^2}_{\text{element wise.}}$

$S_{db} = \beta S_{db} + (1-\beta) db^2$

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

Adam optimation Algo.

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

On iteration t

Compute dW, db using ↑ current mini batch

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

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

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

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

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

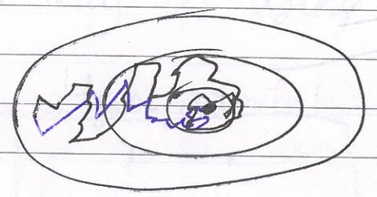
$$S_{dw}^{Corrected} = S_{db} / (1-\beta_2^t)$$

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

α : needs to be tune
β_1 : 0.9 (dw) recomended
β_2 : 0.999 (dw^2) recommeded
ε : 10^{-8} recommended

ADAM → Adaptive moment estimation.

Learning rate decay.



mini batch.

Slowly reduce α. so as you get near to trough you take small steps.

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

Epoch	α
1	0.1
2	0.67
3	0.05
.	.
.	.

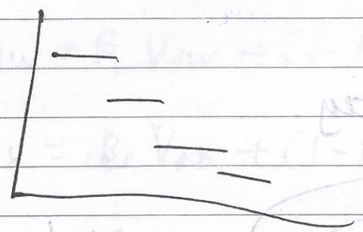
$\alpha_0 = 0.2$
decay rate = 1.

Other decay method

Formula.

$\alpha = 0.95^{epoch\_num} \alpha_0$ - emponentialy decay

$\alpha = \frac{k}{\sqrt{epoch\_num}} \alpha_0$ or $\frac{k}{\sqrt{t}} \alpha_0$



Manual decay.

Hyperparameter's tuning.

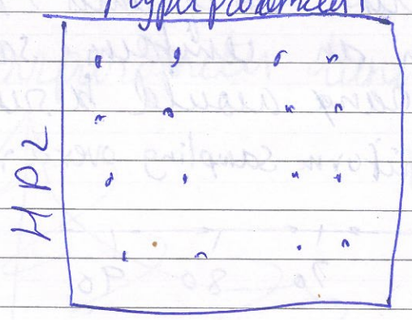
most important priority

Priority given ↓ ↑ α

β, #hidden units, mini batch size.
#layers, learning rate decay

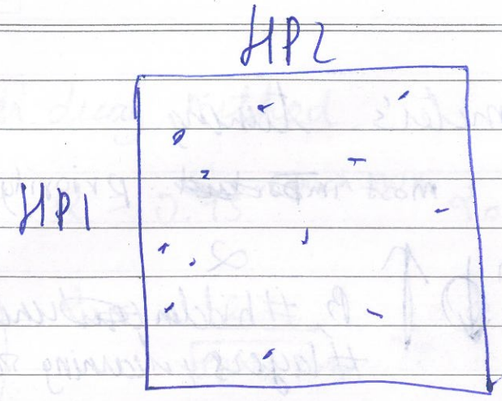
least priority.

One way is using random value.



Try each of the value indicated by dot's equispace between them. And choose the one which gives best result.

Best this works when no of HP are less. When more HP are their to tune. Then use random values given in next page

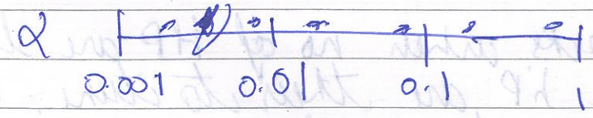


Using appropriate scale for choosing random hyperparameters.

for hyperparameter like no of units, no of layers. choosing in an uniform sampling in a particular range would be reasonable.

$n^{[l]}$ = uniform sampling over a range.

But for alpha try for ~~exponential~~ log scale.



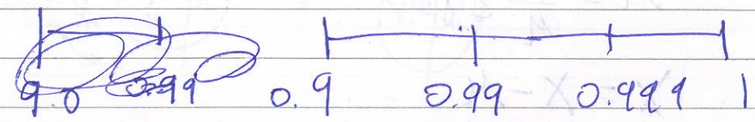
To implement in python.

$r = -4 * np.random.rand()$ ← $r \in [-4, 0]$

$\alpha = 10^r$

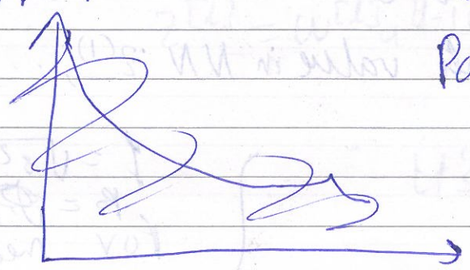
For β. (exponentialy weighted average)

It the same as before But it starts with some bias.



β : 0.900 → 0.9005 } averaging to last 10 value
β 0.999 → 0.9995 averaging to around 1000 value

Hyperparameter tuning practise.



Pandas → manualy done to tune it

Page :
Date :

Batch normalization.

In logistic regression.

$$\mu = \frac{1}{m}\sum x^{(i)}$$

$$X = X - \mu$$

$$\sigma^2 = \frac{1}{m}\sum x^{(i)2}$$

$$X = X/\sigma^2$$

Now we can do this for every layer after applying activation fn.

Give some intermidiate value in NN $z^{(1)}, \ldots z^{(m)}$

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

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

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

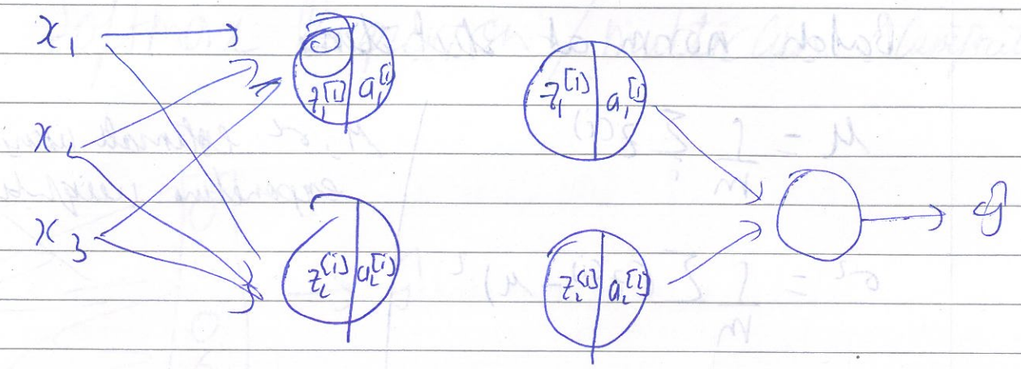
$\gamma = \sqrt{\sigma^2 + \varepsilon}$
$\beta = \mu$
For mean = 0
variance = 1

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

learnable parameters of model.

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

Page :
Date :



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

In batch normalization make the mean zero.

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

$\therefore$ It's not needed.

For t = 1 ... num Mini Batches
Compute forward prop on $X^{\{t\}}$ (mini batch)
In each hidden layer use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$
Use back prop & update $dw^{[l]}$, $d\beta^{[l]}$, $d\gamma^{[l]}$

you can also momentum, Adam, RMSprop.

Page :
Date :

Batch norm at test time

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

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

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

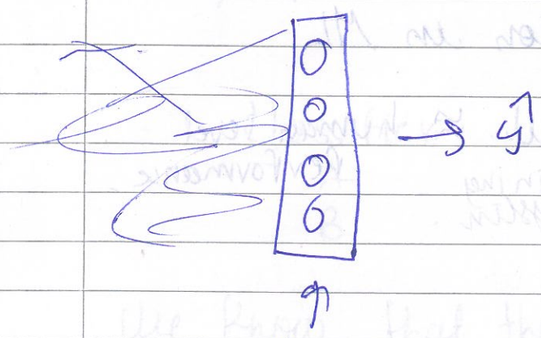
$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$$

μ, σ^2 estimate using exponential weighted average

(Watch the last video of Batch normalization)

Page :
Date :

Softmax Regression / multiclass classification.



last layer.

Now suppose we want to classify 4 things. ⇒ 4 nodes in last layer.

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

Activation fn.

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

$$a^{[L]}_{(4,1)} = \frac{e^{z^{[L]}}}{\sum_{j=1}^{4} t_i} \Rightarrow a_i^{[L]} = \frac{t_i}{\sum_{i=1}^{4} t_i}$$

~~This gives probabilities. out a vector of shape.~~

The softmax gives out a vector (not a value like ReLU, sigmoid, tanh) of probabilities for each layer L.

Orthogonalization.

chain of assumption in ML

→ fit training set well. ≈ human level performance.

g train bigger algo, use better optimizing system.

→ fit dev set well

g. regularization. ↓
Bigger training set

→ fit test set well.

g get bigger dev set ↓

→ Perform well in real world.

g Change the dev set & test set.

As per author early stopping is not used by him because it affects both the 1 & 2 chain in above figure.

Does not mean you can't use it. But it takes away orthogonalization.

Setting up your goal.

→ single number evalution metric →

classifier	Precision	Recall
A	95%	(90%)
B	(98%)	85%

We know that their is a tradeoff between precision and recall. But what if the circled one above (i.e Precision of B & Recall of A) are good. Then how to know which is better

∴ Use F_1 score. (useful iterating over other ML algo).

~~Now suppose your algo~~

Algo	Wrong % → US	Canada	India	China	Avg
A	3%	7%	5%	9%	6%
B					
C					
D					

In above just take Avg. of all. Assuming Avg is a good way.

Now you can also combine time as attributes.

→ Structuring train/dev/test set.

Choose dev & test set such that it should have same distributions.

→ Going away from certain attributes

$$\text{Cost Fn} = \frac{1}{m_{dev}} \sum_{i=0}^{m_{dev}} w^{(i)} \, \text{Loss}\left(y_{pred}^{(i)}, y^{(i)}\right)$$

where $w^{(i)} = \begin{cases} 1 & \text{if } x^{(i)} \text{ is not } y \text{ attribute} \\ 100 & \text{if } x^{(i)} \text{ is } y \text{ attribute} \end{cases}$

where y attribute is something we want to avoid.

Footnote → If your model is not doing well on real world application than change the test set.

Comparing to human level performance.

→ [redacted]

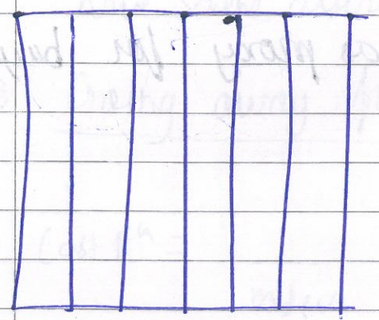
→ The best possible error (i.e least possible error) on a set is called bayes optimal error.

Human level error is used as proxy for bayes error.

Convolutional Neural Network

Page :
Date :

Edge detection.



6×6 image.

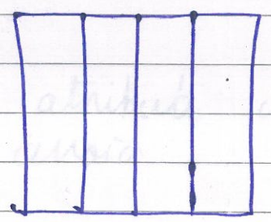
Vertical edge detection

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

filter

$$*$$

$$=$$



This is called Convolutional.

tf.nn.conv2d.

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \rightarrow \text{Vertical}$$

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \rightarrow \text{Horizontal.}$$

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \rightarrow \text{Sober Filter (Vertical)}$$

$$\begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix} \rightarrow \text{Sober Filter}$$

Sometimes for a complicated image we can use all the elements of a filter as a parameter

~~train the parameter~~

$$\begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix}$$

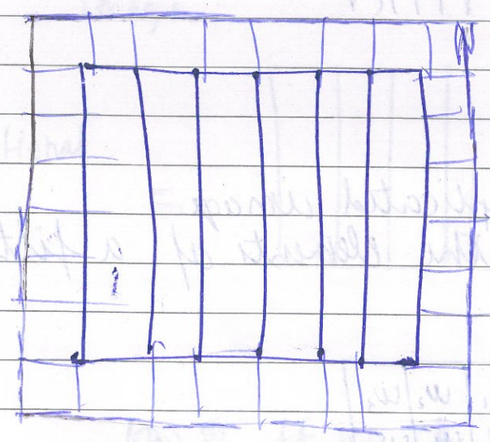
Padding

Suppose you have image of $n \times n$
& filter of $f \times f$

Then you will have a output of
$(n-f+1) * (n-f+1)$

∴ Every time you apply convolution ~~xxxxx~~ your image ~~shrink~~ shrink.

· Also many edge number in image gets multiplied by very filter.



P=1 (Padding layers)

$P = \frac{f-1}{2}$

we mostly initialize padded to zero

6x6 image padded 1 block from edge making multiplication output with filter as 6x6 matrix

Valid and same convolution.

"Valid" → no padding.

"same" → output matrix same as input

f is ~~mostly~~ usually odd

→ Since it means it has a central filter. which can be useful

→ And also due to the formula $P = \frac{f-1}{2}$
which give whole number when f = odd.

Strided Convolution

Just see the video. Hard to explain in books

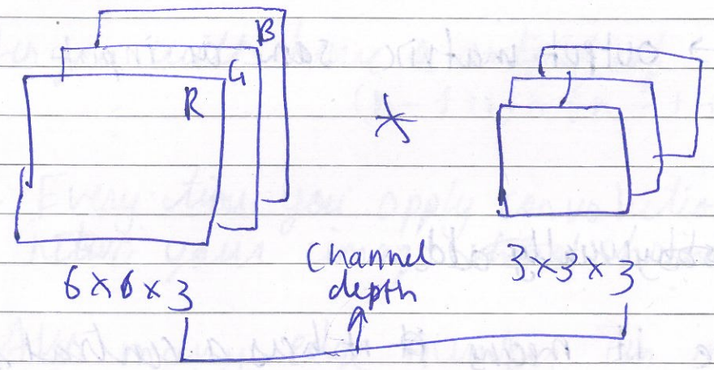
n x n → image
f x f → filter
Padding → p
stride → s

output matrix = $\left(\frac{n+2p-f+1}{s}, \frac{n+2p-f+1}{s} \right)$

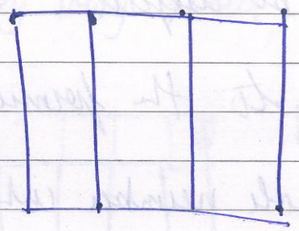
if not fraction then use

For RGB Image

We use 3×3×3 filter.

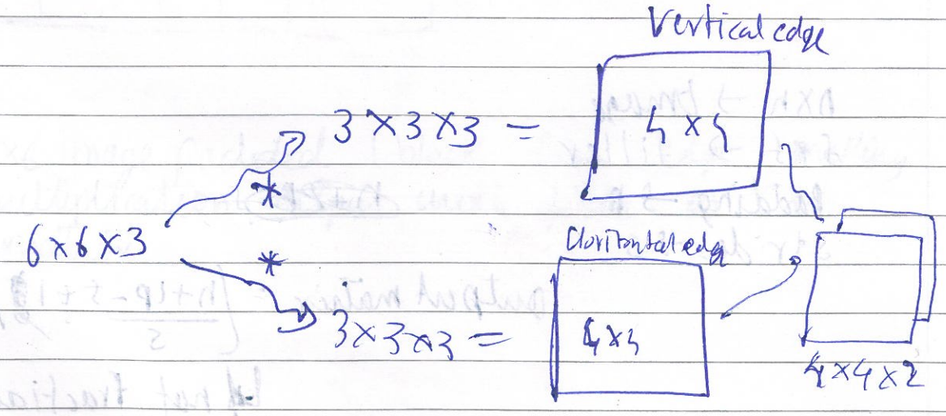


=



4×4 → 2d

Multiple ~~vodur~~ filter

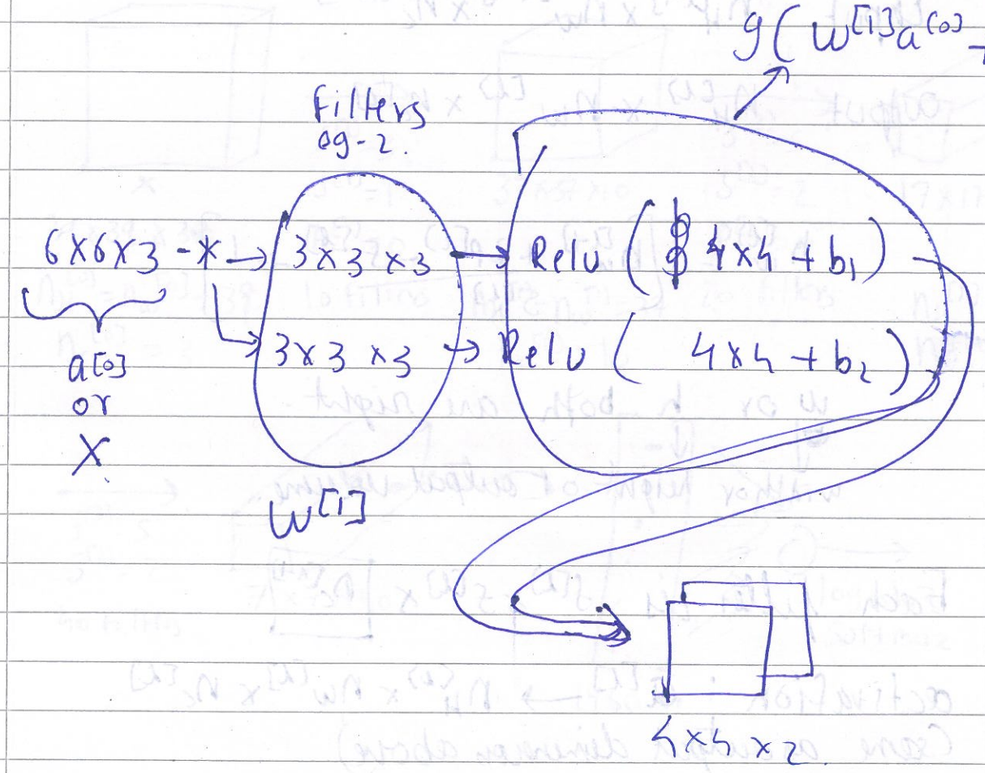


Summary.

$n \times n \times n_c \quad * \quad f \times f \times n_c \rightarrow \underline{n-f+1} \times n-f+1 \times n_c'$

$6 \times 6 \times 3 \qquad 3 \times 3 \times 3 \qquad 4 \times 4 \times 3$

One layer of CNN



notation

If layer l is a convolution layer:

$f^{[l]}$ = filter size
$p^{[l]}$ = padding
$s^{[l]}$ = stride
$n_c^{[l]}$ = num of filters

Input $n_H^{[l-1]} \times n_w^{[l-1]} \times n_c^{[l-1]}$

output $n_H^{[l]} \times n_w^{[l]} \times n_c^{[l]}$

$$n_w^{[l]} = \left\lfloor \frac{n_w^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

w or h both are right.
↓ ↓
width or height of output volume.

Each filter is $f^{[l]} \times f^{[l]} \times \boxed{n_c^{[l-1]}}$

activation: $a^{[l]} \rightarrow n_H^{[l]} \times n_w^{[l]} \times n_c^{[l]}$
(same as output dimension above)

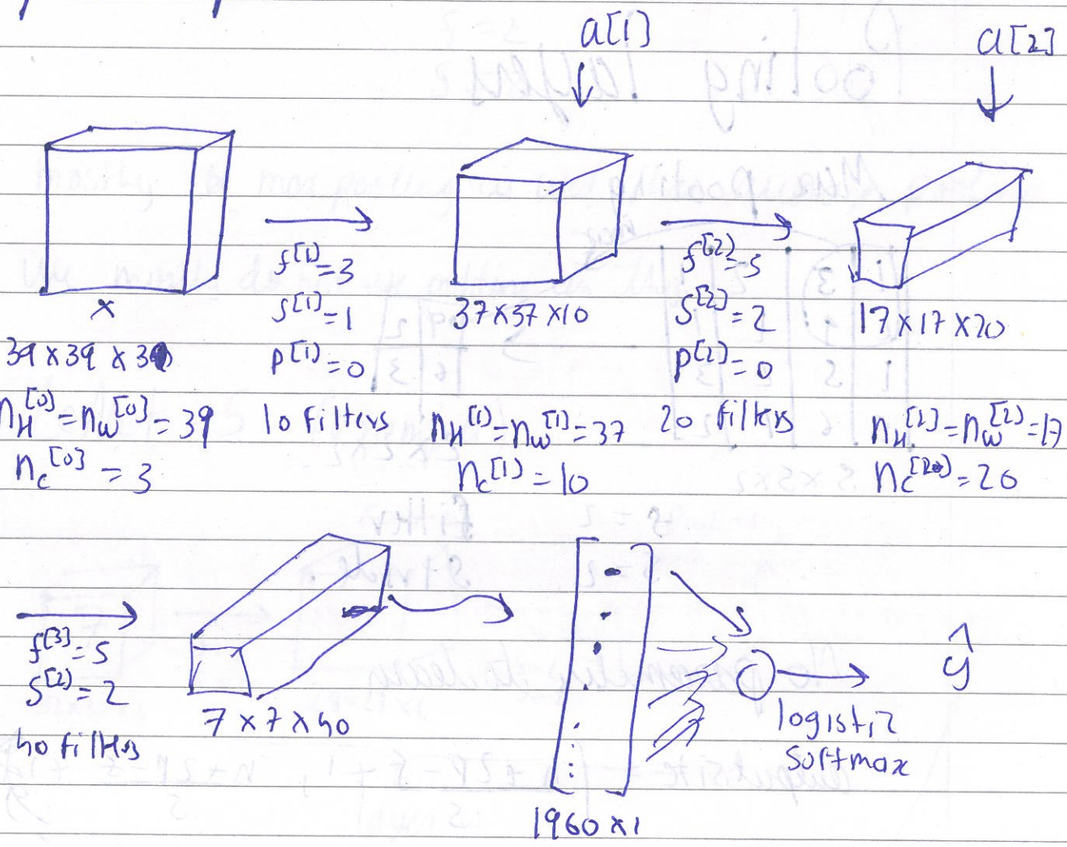
For ~~vectori~~ batch/mini batch gradient descent your output

$A^{[l]} \rightarrow m \times n_H^{[l]} \times n_w^{[l]} \times n_c^{[l]}$ → (not universal)
↘ no of training example

Weights: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$
↑ no of filter.

bias: $n_c^{[l]}$ (i.e $(1,1,1,n_c^{[l]})$

Example



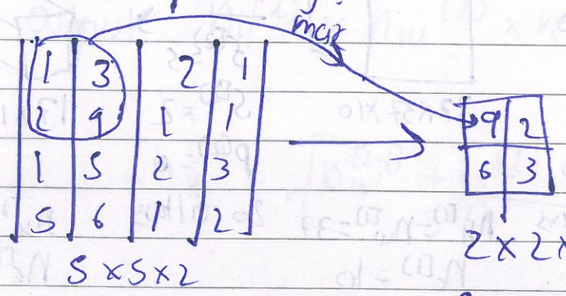
As you see height & width of decreases layer by layer and depth increases.

Types of layer in Convolution network

- convolution (Conv) ←
- Pooling (Pool) ←
- Fully connected (FC) ←

Pooling layers

Max pooling.



5 × 5 × 2 — 2 × 2 × 2.

f = 2 filter

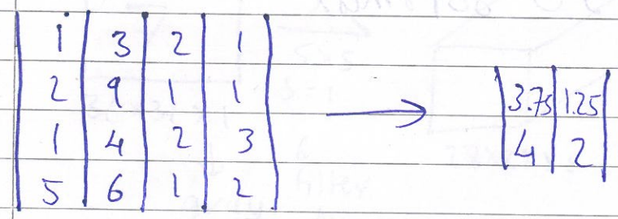
S = 2. Stride.

No parameters to learn

$$\text{output size} = \left(\frac{n + 2p - f}{s} + 1, \frac{n + 2p - f}{s} + 1 \right)$$

, same as previous

Average pooling.

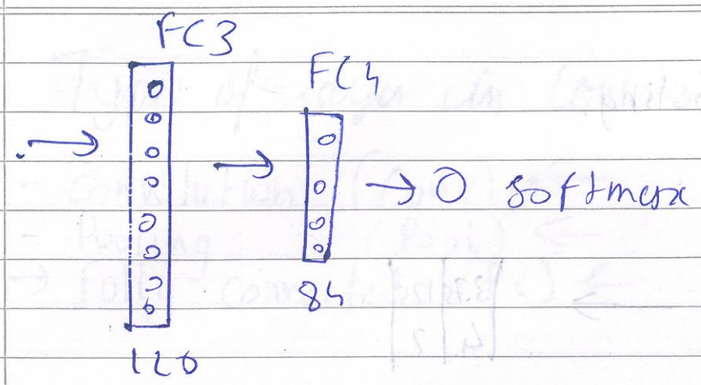


f = 2
S = 2

mostly in max pooling is used than average pooling

We mostly do not use padding in this.

LeNet-5 (example)



$n_H, n_w \downarrow$ $n_c \uparrow$ Trend

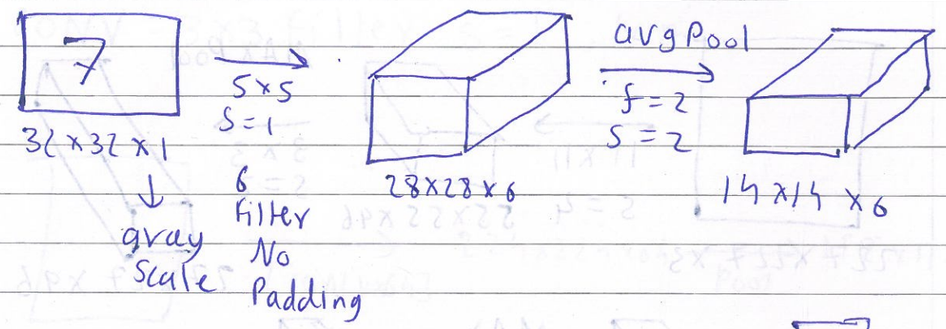
conv pool - conv - pool - FC - FC - FC - softmax

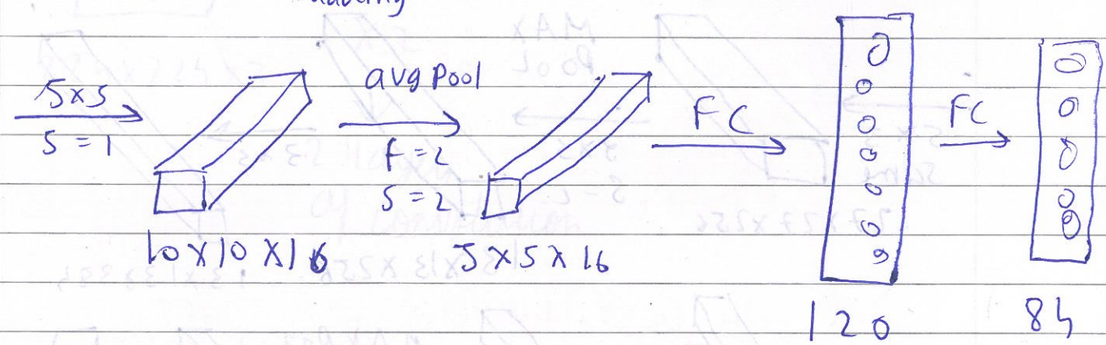
Why Convolution.

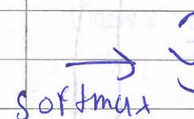
Parameter sharing → A feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image

Sparsity of connection → In each layer, each output value depends only on small number of input.

Lenet - 5







$n_H, n_w \downarrow$ $n_c \uparrow$.

deeper into the Pattern