→ In modern real world supervised learning is applied more & has vast no of applications.

Supervised Learning part - 1

→ takes set of inputs which has correct outputs, learns from it & will become capable enough to predict the outputs for new inputs.

Example: house price prediction

→ This generally fits under regression which is a major part of supervised learning another major one is called classification.

Supervised Learning part - 2

→ Classification algorithms learn based on the dataset provided & classifies the inputs into classes or categories; depending on type of dataset we can have two to any n no of classes.

→ Classification predicts certain fixed values of output unlike regression which can predict infinite different possibilities. → Here we draw some sort of boundary to classify objects into diff categories.

→ We can have multiple sets of input too in classification or regression algorithms.

Unsupervised learning Part-1

→ Here there are no output labels, instead the algorithm tries to find the patterns in the dataset.

→ Eg if the algorithm decides that there is two groups forming after assessing the dataset, such algorithm is called clustering algorithm, this classifies data into clusters.

→ The dataset here do not have any labels, the job of algorithm is to find patterns or group the datasets which are similar.

Unsupervised learning part - 2 (USL)

→ Another type of USL is called anomally detection, which detects when anything unusual occurs Eg fraud detection algorithms detect & flag very large & suspicious transactions.

→ " " " " " " dimensionality reduction, it takes a big dataset & compresses with minimal data loss.

Jupytex notebook

→ Take a look at them when possible.

Linear regression model part-1

→ Terminology

x	y
Size in feet^2	Prices

x = input variable or feature
y = output " " target variable
m = total no of training examples
$(x^{(i)}, y^{(i)})$ = i^{th} training example values

Linear regression model part 2

training set (inputs, outputs are fed and trained.)
↓
learning algorithm
↓
x (test data) → f (hypothesis function) → $\hat{y}$ (predicted y)

Representing f

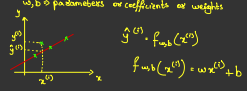
$f_{w,b}(x) = wx + b$ (when f is a straight line)
or
$f(x) = wx + b$

this is a Linear regression with one variable or univariate linear regression

Cost function formula

→ $f_{w,b}(x) = wx + b$

w, b ⇒ parameters or coefficients or weights



$\hat{y}^{(i)} = f_{w,b}(x^{(i)})$

$f_{w,b}(x^{(i)}) = wx^{(i)} + b$

→ To find correct w, b we need cost function which measures the correctness of a graph.

$\hat{y} - y$ ⇒ error

Squared error cost function

$$J(w,b) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2$$

$\hat{y}^{(i)} = f_{(w,b)}(x^{(i)})$

Cost function intuition

→ Goal : $\min_{(w,b)} J(w,b)$ ⇒ way to write it.

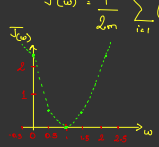
Simplified

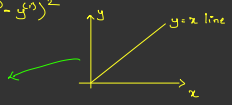
$f_w(x) = wx$ i.e $b = 0$

$$J(w) = \frac{1}{2m} \sum_{i=1}^{m} \left[f_w(x^{(i)}) - y^{(i)}\right]^2$$

$\min_{w} J(w)$

$$J(w) = \frac{1}{2m} \sum_{i=1}^{m} (wx^{(i)} - y^{(i)})^2$$



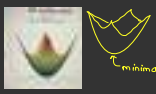


Visualising cost function

→ Back to original model where $b \neq 0$

⇒ 3D plot for 2 variables ie w, b

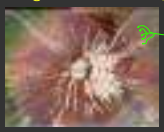
Graph looks somewhat like a soup bowl



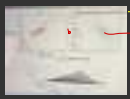
→ This goes on becoming very complex for further increase in features, hence we use something called contour plot, this has ellipses or ovals in it all at same height in 3D plot.

→ All points on a particular ellipse represent diff values of w & b (hence they are diff eq^n) but have same value of J.

→ Imagine contours as top view of mountain, which is sliced.



→ all points at same heights



→ the eq^n represented by crosses on contour plots are given on normal graphs, all have same value of J. (both of them)

Training Linear regression - Gradient descent

→ Gradient descent works for a function with many parameters too Ex $\min_{w_1, w_2, w_3, b} J(w_1, w_2, \ldots, w_n, b)$

We start with some value of w and b

Generally set them to 0,0

→ We keep changing w, b until we settle at or near a minima.

Note : function may have more than 1 minimum

→ Function is always bow or hammock shaped in squared error cost func. but if we train a cost func. based on neural networks the graph will be very complex.

→ Gradient descent chooses the path of steepest descent until we reach at or near a local minima.

Implementing gradient descent

→ $w = w - \alpha \frac{\partial}{\partial w} J(w,b)$ = is assignment operator.

α → small +ve number b/w 0 & 1

↳ controls how much we descend at 1 single step

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

→ w & b are simultaneously updated

Correct way of implementation of simultaneous updation

→ Create temporary variable update both, then store in actual values.

Correct

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

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

$w = temp\_w$

$b = temp\_b$

Incorrect

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

$w = temp\_w$

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

$b = temp\_b$

Learning rate (α)

→ If α is too small, model works too slowly hence it decreases the efficiency of the model

→ " " " large, gradient descent overshoots & never reaches a minimum i.e gradient descent diverges or fails to converge.

→ what happens when we are already at a minima?

the values further don't update as $\frac{\partial J}{\partial (\text{parameter})} = 0$

→ As we approach minima gradient descent automatically takes smaller steps until minima is attained (as derivative value goes on ↓ hence steps becomes smaller)

→ ∴ Gradient descent reaches minimum even with constant α.

Training Linear regression ⇒ Gradient descent for linear regression

→ $f_{w,b}(x) = wx + b$

$$J(w,b) = \frac{1}{2m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)}) - y^{(i)} \right)^2$$

$$w = w - \alpha \frac{\partial}{\partial w} J(w,b) \quad \Rightarrow \frac{\partial}{\partial w} J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)}) - y^{(i)} \right) x^{(i)}$$

$$b = b - \alpha \frac{\partial}{\partial b} J(w,b) \quad \Rightarrow \frac{\partial}{\partial b} J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)}) - y^{(i)} \right)$$

→ The type of function produced in squared error cost function is called convex function ⇒ it has only single global minima & no local minima.

Running Gradient descent

→ Gradient descent on a house price predictor with 47 datapoints,

this is called batch gradient descent as each step of the descent uses all of the training examples.

There are other kinds of regression too which need not take the entire batch for each step.

Linear regression with multiple variables – Multiple features

→ Multiple features ⇒ $x_1, x_2, \ldots, x_n$

x_j = j^{th} feature n = total number of features

$\vec{x}^{(i)}$ = features of i^{th} training example

↳ its a list of 'n' numbers, its a vector which includes all the features of i^{th} training example

$x_j^{(i)}$ ⇒ value of j^{th} feature of i^{th} training example

Example

→ $f_{\vec{w},b}(\vec{x}) = w_1x_1 + w_2x_2 + w_3x_3 + \ldots + w_nx_n + b$

$\vec{w} = [w_1\ w_2 \ldots\ w_n]$ ⟶ new vector of all weights

$\vec{x} = [x_1, x_2 \ldots x_n]$

$$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$$

(· : dot product of vectors)

This is called multiple linear regression.

(not multivariati regression)**

Vectorization, Part – 1

→ This makes code shorter & more efficient, uses modern numerical linear algebra libraries, uses GPU.

$\vec{w} = [w_0\ w_1 \ldots\ w_{n-1}]$; b

$\vec{x} = [x_0\ x_1 \ldots\ x_{n-1}]$

(0 indexing for coding)

w = np.array([elements])
b =
x = np.array([elements])

Using loop (include Numpy)

```
f = 0
for j in range(0, n):        # goes from 0 to n-1
    f = f + w[j] * x[j]
f = f + b
```

Using vectorization

```
f = np.dot(w,x) + b
```

⟶ when dataset is longer this runs much more faster than the loop

Vectorization, part-2

→ Why is vectorization faster?

In Loop each element is accessed separately multiplied, added but in vectorization entire arrays are accessed & multiplied at once, then computer performs very effiient addition.

∴ Vectorization is very efficient & scales well to longer datasets

Gradient descent for multiple linear regression

→ $\vec{w} = [w_1 \ldots w_n]$

$f_{\vec{w},b} = \vec{w} \cdot \vec{x} + b$

$J(\vec{w}, b)$

repeat {

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

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

}

Gradient descent for one feature vs n features

for one feature

$$w = w - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)}) - y^{(i)}\right) x^{(i)}$$

($\frac{\partial}{\partial w} J(w,b)$)

$$b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)}) - y^{(i)}\right)$$

($\frac{\partial}{\partial b} J(w,b)$)

for multiple features

$$w_1 = w_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right) x_1^{(i)}$$ ($\frac{\partial}{\partial w_1} J(\vec{w},b)$)

$$w_n = w_n - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right) x_n^{(i)}$$ ($\frac{\partial}{\partial w_n} J(\vec{w},b)$)

$$b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)$$

→ Alternative way of finding w, b:

Normal eq^n

→ works only for linear regression

→ Solves for w, b without iterations

Disadvantages

→ isn't generalised to other learning algorithms

→ is slow when number of features are larger than 10,000

→ complex ML libraries use normal eq^n in backend (generally).

Feature scaling part -1

→ This makes gradient descent much more faster.

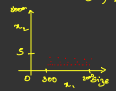
Example

price = $y = w_1 x_1 + w_2 x_2 + b$

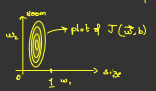
x_1 = size = 300 - 2000 sq.feet ; large range

x_2 = rooms = 0-5 ; small "

the larger the size of feature, lower will be the value of its weight & vice versa.



variation in size is very large

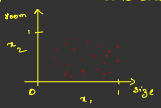


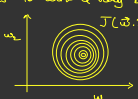
Very small change in w_1 can have large impact hence it varies in a very small range

& w_2 varies in large range as x_2 is small

∵ the ellipses are so small the descent bounces a lot before reaching minima, hence we need some sort of a hack.

→ Make both x_1 & x_2 comparable i.e make some changes to units & vary both x_1 & x_2 say in range of 0 to 1, then the plots begin to look like this:





Here the descent reaches the minima much more easily.

Feature Scaling part -2

→ How to scale features

Ex $300 \le x_1 \le 2000$ $\quad 0 \le x_2 \le 5$

Method 1

$x_{1\,scaled} = \frac{x_1}{2000}$ → max value of range $\quad x_{2\,scaled} = \frac{x_2}{5}$

$0.15 \le x_{1\,scaled} \le 1$ $\quad 0 \le x_{2\,scaled} \le 1$

aim for $-1 \le x_j \le 1$ for each feature x_j

Acceptable ranges ⇒ $-1 \le x_j \le 1$

$-3 \le x_j \le 3$

$-0.3 \le x_j \le 3$

Ex

$0 \le x_1 \le 3$ ⇒ accepted

$-2 \le x_2 \le 0.5$ ⇒ "

$-100 \le x_3 \le 100$ ⇒ too large, ∴ rescale

$-0.001 \le x_4 \le 0.001$ ⇒ " small, ∴ "

$98.6 \le x_5 \le 105$ ⇒ " large, ∴ "

Mean normalisation

$300 \le x_1 \le 2000$ $\quad 0 \le x_2 \le 5$

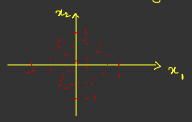
Mean of all obs $x_1 = \mu_1$ $\quad$ Mean of all obs $x_2 = \mu_2$

Method 2

$x_{1\,scaled} = \frac{x_1 - \mu_1}{2000 - 300}$ (max − min) $\quad x_{2\,scaled} = \frac{x_2 - \mu_2}{5 - 0}$

Suppose $-0.18 \le x_1 \le 0.82$ $\quad -0.46 \le x_2 \le 0.54$

the values vary here between -1 to 1



Z-score normalisation

Calculate standard deviation of each feature (σ)

Normal distribution or bell shaped curve (same as σ)

or

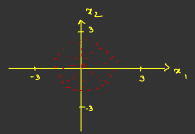
Gaussian distribution

Method-3

$300 \le x_1 \le 2000$ $\quad 0 \le x_2 \le 5$

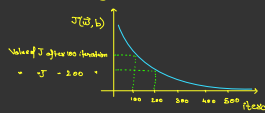
$x_{1\,scaled} = \frac{x_1 - \mu_1}{\sigma_1}$ $\quad x_{2\,scaled} = \frac{x_2 - \mu_2}{\sigma_2}$

$-0.67 \le x_1 \le 3.1$ $\quad -1.6 \le x_2 \le 1.9$



Checking gradient descent for convergence

→ Learning curve



After each iteration J must decrease, if J ↑ even in 1 step it means α is chosen poorly or there is some bug in the code

• 500 • J seems to flatten that means its no longer decreasing hence J has more or less converged

There is no telling after how many iterations gradient descent converges

→ Automatic convergence test

Let ε be very small number like 10^{-3}

If $J(\vec{w}, b)$ decreases by $\leq \varepsilon$ in one iteration then declare convergence

Disadv: finding correct value for ε is very hard hence many prefer Learning curves as we can tell if there is a mistake in code or not too in that method.

Choosing learning rate

→ In learning curve if the graph ↑ then α is not chosen properly. (or bug in code)

→ If Learning curve look like this, then either α is very high or error in code possibly $w = w + \alpha d$

→ Even when α is small graph ↑ then there is bug in code
(this is done to debug)

→ Try with many α, generally 3x of prev α like 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1.

Feature engineering

→ its using intuition to design new features, by transforming or combining original features.

Ex $f_{\vec{w},b}(\vec{x}) = w_1 x_1 + w_2 x_2 + b$

x_1 ⇒ frontage of plot
x_2 ⇒ depth " "

$x_3 = x_1 x_2$ ⇒ area of plot

$f_{\vec{w},b}(\vec{x}) = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$

Area helps predict price more accurately, hence we create a new feature

Polynomial regression

→ we may need a curve which fits dataset properly cz there is no proper st. line, we choose diff polynomial degree eq^n to do this

Quad eqn func ∩ after a point (i.e after maxima)  if we stop a func which only ↑ use cubic or $y = w_1 x + w_2 \sqrt{x} + b$

→ Feature scaling becomes really important here: the powers are varying.

Module - 3

Motivations

→ Classification

Binary classification: outcomes have only 2 possibilities

no	yes
false	true
0	1
↳ negative class	↳ positive class

→ Why linear regression cannot be used to classify?

* Adding a training shifts the decision boundary which is not desirable.

Logistic regression

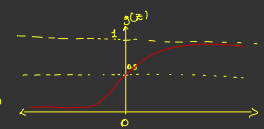
→ Has S-like curve.

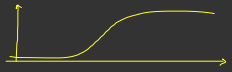
→ The function used here is called sigmoid function or logistic function ⇒ $g(z) = \frac{1}{1+e^{-z}}$

$z \gg 0 \Rightarrow g(z) \approx 1$

$z = 0 \Rightarrow g(z) = 0.5$

$z \ll 0 \Rightarrow g(z) \approx 0$





⇒ our functions (predicted) will look somewhat like this

$f_{\vec{w},b}(\vec{x}) = z = \vec{w}.\vec{x} + b$

$g(z) = \frac{1}{1+e^{-z}} \quad 0 < g(z) < 1$

$f_{\vec{w},b}(\vec{x}) = g(\vec{w}.\vec{x}+b) = g(z) = \frac{1}{1+e^{-(\vec{w}.\vec{x}+b)}}$

Ex function value turns out to be 0.7 ⇒ 70% chance that y is 1.

or

30% . . . 0.

$\underbrace{P(y=0)}_{\text{probability}} + P(y=1) = 1$

Hence func is also written as

Decision boundary

$f_{\vec{w},b}(\vec{x}) = P(y=1 \mid \vec{x}; \vec{w}, b)$

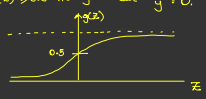
→ In general when $f_{\vec{w},b}(\vec{x}) \geq 0.5$ then $\hat{y} = 1$ else $\hat{y} = 0$.

for $\hat{y}=1$ ↓

$f_{\vec{w},b}(\vec{x}) \geq 0.5$

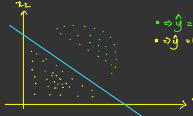
$g(z) \geq 0.5$

$z \geq 0$



$\vec{w}.\vec{x} + b \geq 0 \quad \vec{w}.\vec{x} + b < 0 \Rightarrow \hat{y} = 0$

→ Ex 1



- ⇒ $\hat{y} = 1$
- ⇒ $\hat{y} = 0$

$f_{\vec{w},b}(\vec{x}) = g(z) = g(w_1 x_1 + w_2 x_2 + b)$

curve $z = 0$ becomes the decision boundary ⇒ say for this ex $w_1 = 1, w_2 = 1, b = -3$

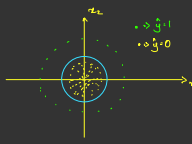
then boundary is

$x_1 + x_2 - 3 = 0$

$\frac{x_1}{3} + \frac{x_2}{3} = 1$ ⇒ st line with $x_{int} = y_{int} = 3$

To right of line $\hat{y} = 1$ & left $\hat{y} = 0$

Ex 2



- ⇒ $\hat{y} = 1$
- ⇒ $\hat{y} = 0$

Let $z = w_1 x_1^2 + w_2 x_2^2 + b$

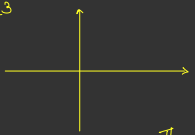
for this ex let $w_1 = w_2 = 1$ & $b = -1$

then for $z = 0 \Rightarrow x_1^2 + x_2^2 = 1$

this is decision boundary, where inside is assigned as 0, outside as 1

this assignment is defined % to need

Ex 3



$z = w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_1 x_2 + w_5 x_2^2$

can be any of 2 degree curve

inside & outside is assigned as 0 or 1 % to need

→ The curve depends on z i.e powers of x_i.

→ The threshold value of decision boundary need not be 0.5 always, it can arbitrarily chosen too

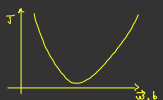
Ex for tumor detection we can choose 0.2 as threshold such that above 0.2 is malignant below is benign; this reduces the chance of false positives.

Cost function for logistic regression

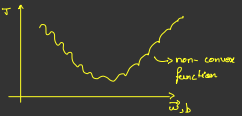
→ Ex training having m training examples with n features with two targets ⇒ 0 & 1 ⇒ (tumor detection)

Squared error cost func. ⇒ $J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m} \frac{1}{2}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2$

for linear regression



for logistic regression



prob ⇒ has too many local minimas hence this is not a good cost function for logistic regression.

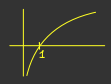
$$J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m} \underbrace{\frac{1}{2}\left(f_{\vec{w},b}(x^{(i)}) - y^{(i)}\right)^2}_{L}$$

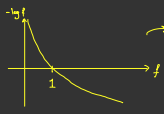
loss $\Rightarrow L\left(f_{\vec{w},b}(x^{(i)}), y^{(i)}\right)$

Loss = $\frac{1}{2}$ of squared difference (here)

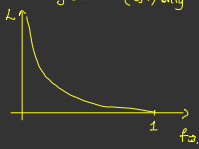
Logistic loss function

$$L\left(f_{\vec{w},b}(x^{(i)}), y^{(i)}\right) = \begin{cases} -\log\left(f_{\vec{w},b}(x^{(i)})\right) & \text{if } y^{(i)}=1 \\ -\log\left(1 - f_{\vec{w},b}(x^{(i)})\right) & \text{if } y^{(i)}=0 \end{cases}$$

$\log f \Rightarrow$ 

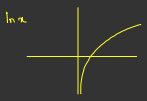
$-\log f \Rightarrow$ 

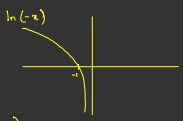
→ significant part is only 0 to 1 as y can be (0,1) only

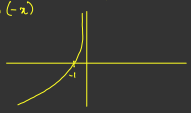


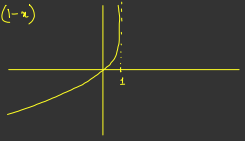
→ if algo predicts f & actually its 1 loss is very close to 0

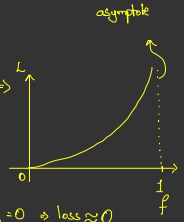
→ " " 0 " " " " " high → loss ≈ ∞









$\therefore -\log(1-f) \Rightarrow$ 

→ when $\hat{y} \approx 0$ & $y = 0$ ⇒ loss ≈ 0

→ when $\hat{y} \approx 1$ & $y = 0$ ⇒ loss ≈ ∞

→ Overall cost function will be convex when this is implemented

$$J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m} L\left(f_{\vec{w},b}(x^{(i)}), y^{(i)}\right)$$

$$L\left(f_{\vec{w},b}(x^{(i)}), y^{(i)}\right) = \begin{cases} -\log\left(f_{\vec{w},b}(x^{(i)})\right) & \text{if } y^{(i)}=1 \\ -\log\left(1 - f_{\vec{w},b}(x^{(i)})\right) & \text{if } y^{(i)}=0 \end{cases}$$

Simplified cost func. for logistic regression

→ The above loss func can be written as follows:

$$L\left(f_{\vec{w},b}(x^{(i)}), y^{(i)}\right) = -y^{(i)}\log\left(f_{\vec{w},b}(x^{(i)})\right) - \left(1-y^{(i)}\right)\log\left(1 - f_{\vec{w},b}(x^{(i)})\right)$$

How did we end up with this complex loss func. & its based on something called maximum likelihood estimation which is part of statistics

$$J(\vec{w},b) = \frac{-1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(f_{\vec{w},b}(x^{(i)})\right) + (1-y^{(i)})\log\left(1 - f_{\vec{w},b}(x^{(i)})\right)\right]$$

This cost func. is now convex & has only one single global minima.

Gradient descent implementation

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

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

$\frac{\partial}{\partial w_j} J(\vec{w}, b) = \frac{1}{m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(x^{(i)}) - y^{(i)}\right) x_j^{(i)}$

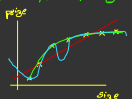
$\frac{\partial}{\partial b} J(\vec{w}, b) = \frac{1}{m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(x^{(i)}) - y^{(i)}\right)$

Derivation?

→ this also can be monitored i.e analysed with learning curves

→ Vectorization, feature scaling is applicable here too.

Problem of overfitting

→ 

⇒ underfits the curve or has high bias

bias ⇒ the program thinks that the curve is a st line & attempts but any st line will not fit this model, hence the model was biased & has high bias

⇒ fits the dataset pretty well, this is called generalisation, it has neither high bias nor high variance

⇒ fits the dataset exactly, cost func. is zero but this will not do good job in prediction with test cases, the model is overfit, it has high variance

eg: even if one datapoint changes slightly the graph will completely change in order to fit that too, hence it has high variance.

Addressing overfitting

→ Collect more training examples

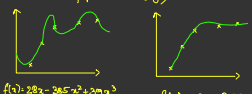
→ Select features to include or exclude ⇒ this is called feature selection

including too many features may also cause overfitting

disadv: useful feature might be lost

→ Regularisation

Reduce the size of parameters (w_j)

Eg: 

$f(x) = 28x - 365x^2 + 23x^3 - 17x^4 + 100$

$f(x) = 13x - 0.23x^2 + 0.00001x^3 - 0.0001x^4 + 10$

Regularisation includes all chosen features but prevents features from having an overly large effect

Note: whether we change b or not makes very less difference practically.

Cost function with regularisation

→ Let's say the $f(x) = w_1x_1 + w_2x_2^2 + w_3x_3^3 + w_4x_4^4 + b$

to prevent this from overfitting we must penalize w_3 & w_4

we can do this by,

$$\min_{\vec{w},b} \frac{1}{2m} \sum_{i=1}^{m} \left[f(x^{(i)}) - y^{(i)}\right]^2 + 1000 w_3^2 + 1000 w_4^2$$

1000 coz its large num, hence w_3 & w_4 would almost be zero.

the basic idea is that that if the weights are small its like having a simple model with fewer features hence is less prone to overfit.

But in a model with many features we do not know which to penalize & which to not hence we penalize all features i.e all w_j parameters, this makes model correctly fit the data.

Eg: n features, house pricing [Linear reg version]

$$J(\vec{w}, b) = \underbrace{\frac{1}{2m} \sum_{i=1}^{m} \left[f(x^{(i)}) - y^{(i)}\right]^2}_{\text{mean squared error}} + \frac{\lambda}{2m} \sum_{j=1}^{n} w_j^2$$

λ ⇒ regularization parameter $\lambda > 0$

λ is scaled by multiplying by $\frac{1}{2m}$ coz this makes it easy to choose λ.

Some include $\frac{\lambda}{2m} b^2$ also, but it makes very little difference if we penalize b or not.

λ's value decides the relative tradeoff between minimizing cost & minimizing weights

when λ is too high model underfits → $f(\vec{w}, b) \approx b$

" " " low " overfits

Derivation of partial deriv. of cost func of logistic regression

$$J(\vec{\omega}, b) = \frac{-1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log\left(f_{\vec{\omega},b}(x^{(i)})\right) + (1-y^{(i)}) \log\left(1 - f_{\vec{\omega},b}(x^{(i)})\right) \right]$$

$$f(x) = \omega x + b$$

$$J = \frac{-1}{m} \left[y \log(f(x)) + (1-y) \log(1-f(x)) \right]$$

$$\frac{\partial J}{\partial \omega} = \frac{-1}{m} \left[\frac{y}{f(x)} \cdot \frac{\partial f(x)}{\partial \omega} - \frac{(1-y)}{1-f(x)} \cdot \frac{\partial}{\partial \omega} f(x) \right]$$

$$f(x) = \frac{1}{1 + e^{-\omega x}}$$

$$\frac{\partial}{\partial \omega} f(x) = \frac{-e^{-\omega x} \cdot x(-1)}{\left(1 + e^{-\omega x}\right)^2} = (+x) \left[\frac{1 + e^{-\omega x}}{\left(1 + e^{-\omega x}\right)^2} - \frac{1}{\left(1 + e^{-\omega x}\right)^2} \right]$$

$$= (+x) \left[f(x) - (f(x))^2 \right]$$

$$= (+x) \, f(x) \left[1 - f(x) \right]$$

$$\frac{\partial J}{\partial \omega} = \frac{-1}{m} \left[\frac{y}{f(x)} \cdot (x) \, f(x)\left(1-f(x)\right) - \frac{(1-y)}{\left(1-f(x)\right)} \cdot (x) \, f(x)\left(1-f(x)\right) \right]$$

$$\frac{\partial J}{\partial \omega} = \frac{-x}{m} \left[y\left(1-f(x)\right) - (1-y)\left(f(x)\right) \right]$$

$$\frac{\partial J}{\partial \omega} = \frac{-x}{m} \left[y - yf(x) - f(x) + y f(x) \right]$$

$$\frac{\partial J}{\partial \omega} = \frac{-1}{m} \left[y - f(x) \right] \cdot x$$

$$\therefore \frac{\partial J}{\partial \omega} = \frac{-1}{m} \sum_{i=1}^{m} \left[\left(y^{(i)} - f_{\vec{\omega},b}(x^{(i)}) \right) x^{(i)} \right]$$

$$J = -\frac{1}{m}\left[y \log(f(x)) + (1-y)\log(1-f(x)) \right]$$

$$\frac{\partial J}{\partial b} = -\frac{1}{m}\left[\frac{y}{f(x)} \frac{\partial f(x)}{\partial b} - \frac{(1-y)}{1-f(x)} \frac{\partial f(x)}{\partial b} \right]$$

$$f(x) = \frac{1}{1+e^{-(wx+b)}}$$

$$\frac{\partial f(x)}{\partial b} = \frac{-(-1)(1)}{\left[1+e^{-(wx+b)}\right]^2} \cdot e^{-(wx+b)} = \frac{1}{1+e^{-(wx+b)}} - \frac{1}{\left[1+e^{-(wx+b)}\right]^2} \Rightarrow f(x)\left[1-f(x)\right]$$

$$\frac{\partial J}{\partial b} = -\frac{1}{m}\left[\frac{y}{f(x)} \cdot f(x) \cdot (1-f(x)) - \frac{(1-y)}{1-f(x)} f(x) \cdot (1-f(x)) \right]$$

$$\frac{\partial J}{\partial b} = -\frac{1}{m}\left[y - yf(x) - f(x) + yf(x) \right]$$

$$\frac{\partial J}{\partial b} = -\frac{1}{m}\left[y - f(x) \right]$$

$$\therefore \frac{\partial J}{\partial b} = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} - f_{\vec{w},b}(x^{(i)}) \right]$$

$$\because \frac{\partial J}{\partial w} = -\frac{1}{m}\sum_{i=1}^{m}\left[\left[y^{(i)} - f_{\vec{w},b}(x^{(i)}) \right] \cdot x^{(i)} \right]$$

Regularised Linear regression

→ $J(\vec{w},b) = \frac{1}{2m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(x^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$

$\frac{\partial}{\partial w_j} J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(x^{(i)}) - y^{(i)}\right) x_j^{(i)} + \frac{\lambda}{m} w_j$

$\frac{\partial}{\partial b} J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(x^{(i)}) - y^{(i)}\right)$

 then w_j shrinks by a little in each iteration.

Let $\alpha = 0.01$ & $\lambda = 1^3$

Regularised logistic regression

→ in general when we train the model with lot of features then there is a higher risk of overfitting

hence to modify cost func, for n features

$J(\vec{w},b) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) + (1-y^{(i)})\left(\log\left(1 - f_{\vec{w},b}(x^{(i)})\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$

$\frac{\partial}{\partial w_j} J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(x^{(i)}) - y^{(i)}\right) x_j^{(i)} + \frac{\lambda}{m} w_j$

$\frac{\partial}{\partial b} J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(x^{(i)}) - y^{(i)}\right)$

Second : neural & deep

Normal eqⁿ from yt : an analytical method to solve for θ

intuition,

in 1D

$J(\theta) = a\theta^2 + b\theta + c$

$\frac{d}{d\theta} J(\theta) = 0$

(to find minima)

when there are multiple features

$\theta \in R^{n+1}$ $J(\theta_0,\theta_1,\theta_2,\ldots,\theta_n) = \frac{1}{2m}\sum_{i=1}^{m}\left[h_\theta(x^{(i)}) - y^{(i)}\right]^2$ (hypothesis func.)

$\frac{\partial}{\partial \theta_j} J(\theta) = 0$ (for every j)

solve for $\theta_0, \theta_1, \theta_2, \ldots, \theta_n$

* Normal eqⁿ method does not require feature scaling method.

Gradient descent	Normal Eqⁿ
* Need to choose α	* No need to choose α
* Needs many iterations	* No need to iterate
* Works well even when n is large	* Needs $O(n^3)$ time to calculate $(X^TX)^{-1}$ (→ order $n \times n$)
	* Slow when $n \geq 10^6$
	Ok-ok for 10^5 faster for 0 to 10^4

For a linear regression having m training set & n features

$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix} \in R^{n+1}$ order : $(n+1)\times 1$

$x_0^{(i)} = 1$ ⇒ taken by us generally

$X = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(m)})^T — \end{bmatrix}$ design matrix, order: $m \times (n+1)$

$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$

$\theta = (X^TX)^{-1}X^Ty$

Stochastic gradient : CS 229 lecture 2 y^t

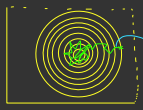
→ if dataset is very large then each iteration to modify values takes $O(n)$ time to process through the whole dataset.

→ Here we need some sort of alternative

→ Stochastic gradient descent :

```
repeat {
    for i=1 to m
    {
        θ_j = θ_j − α ∂/∂θ_j (h_θ(x^(i)) − y^(i)) x_j^(i)
        (for every j 1 to n)
    }
}
```

Contour plot ⇒



→ stochastic gradient never converges as it amends its values for each home, but when we have a very large dataset this is fine as it reaches very near to minimum & this only is applied instead of batch gradient descent.

Rough derivation of normal eqⁿ

$\nabla_\theta J(\theta)$ ⇒ represents derivative of $J(\theta)$ $\theta \in \mathbb{R}^{n+1}$ matrix ⇒

$$\nabla_\theta J(\theta) = \begin{bmatrix} \frac{\partial}{\partial \theta_0} J(\theta) \\ \frac{\partial}{\partial \theta_1} J(\theta) \\ \frac{\partial}{\partial \theta_2} J(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} J(\theta) \end{bmatrix}$$

$\nabla_\theta J(\theta) = \vec{0}$ for reaching global minima

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

$$X\theta = \begin{bmatrix} — & (x^{(1)})^T & — \\ — & (x^{(2)})^T & — \\ & \vdots & \\ — & (x^{(m)})^T & — \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} = \begin{bmatrix} (x^{(1)})^T\theta \\ (x^{(2)})^T\theta \\ \vdots \\ (x^{(m)})^T\theta \end{bmatrix} = \begin{bmatrix} h_\theta(x^{(1)}) \\ h_\theta(x^{(2)}) \\ \vdots \\ h_\theta(x^{(m)}) \end{bmatrix}$$

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

$$\begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \vdots \\ h_\theta(x^{(m)}) - y^{(m)} \end{bmatrix}$$

$$J(\theta) = \frac{1}{2} (X\theta - y)^T (X\theta - y)$$

$$\nabla_\theta J(\theta) = \nabla_\theta \left[\frac{1}{2} (X\theta - y)^T (X\theta - y) \right]$$

$$= \frac{1}{2} \nabla_\theta \left[(\theta^T X^T - y^T)(X\theta - y) \right]$$

$$= \frac{1}{2} \nabla_\theta \left[\theta^T X^T X \theta - \theta^T X^T y - y^T X \theta - y^T y \right]$$

derivative

$$\Rightarrow \frac{1}{2} \left[X^T X \theta + X^T X \theta - X^T y - X^T y \right]$$

$$\Rightarrow X^T X \theta - X^T y = 0$$

$$X^T X \theta = X^T y$$

$$\theta = (X^T X)^{-1} X^T y$$

what if X is non invertible?

it means we have redundant features i.e features which are linearly dependent. This can be solved by: (i) Pseudo inverse (ii) Examining which of the features are linearly dependent.

i.e $x_1 = 3x_2 + c$

Ex $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$

$$\nabla_A f(A) = \begin{bmatrix} \frac{\partial f}{\partial A_{11}} & \frac{\partial f}{\partial A_{12}} \\ \frac{\partial f}{\partial A_{21}} & \frac{\partial f}{\partial A_{22}} \end{bmatrix}$$

($f(A) = A_{11} + A_{12}^2$ anything like this)

$\operatorname{tr} A = \operatorname{tr}(A) = \sum_{i=1}^{n} A_{ii}$

n×n matrix

→ $\operatorname{tr}(A) = \operatorname{tr}(A^T)$

→ $f(A) = \operatorname{tr}(AB)$

$\nabla_A f(A) = B^T$

→ $\operatorname{tr}(AB) = \operatorname{tr}(BA)$

→ $\operatorname{tr}(ABC) = \operatorname{tr}(CAB)$

→ $\nabla_A AA^TC = CA + C^TA$

(analogous to $\frac{d}{da}(a^2c) = 2ac$)

→ $A^TA = \sum_i A^2$ (sum of sq. of ele.)

Advanced Algorithms

Welcome

Week 1 - already applied neural net.

Week 2 - make your own - .

Week 3 - techniques

week 4 - decision trees

Neurons and the brain

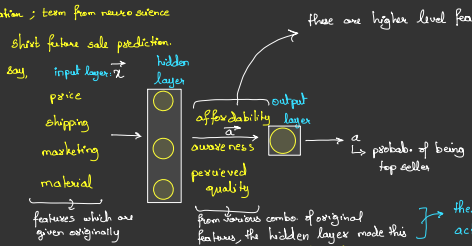
→ just like biological neurons artificial neuron(s) take some input ie some number, computs something then output another number which acts as an input for next neuron(s).

→ There was need for these as classic ML models couldn't handle the bombardment of data in our era whereas neural network models perform highly well even when dataset is very large.

Demand prediction

→ a: activation ; term from neuro science

→ Example: Shirt feature sale prediction.



→ Neuron or neurons forming a group is called a layer.

→ All ele. of layer have access to all of the inputs but different neurons choose only specific inputs and ignores others. This is not done manually but we do it algorithmically.

→ If we hide the first column we are just left out with a logistic regression model which predicts whether the T-shirt would be a top-seller or not. it also does feature engineering automatically.

→ There can be multiple hidden layers too of various sizes comprising of diff no of neurons.

→ The no of hidden layers & the no of neurons in each hidden layer ⇒ this has large impact on efficiency of model.

⇒ deciding this comes under neural net work architecture.

⇒ When there are multiple hidden layers, such models are also called multilayer perceptron

Example: Image recognition

→ m x n pixel img is a m x n grid in which intensity values from 0-255 is stored

→ On flattening it into a 1D vector we get m x n ele as column vector. The model takes this as input & must output the identity of person in img.

→ hidden layer 1 ⇒ looks for straight edges in images. Looks at a highly zoomed in img.

" " 2 ⇒ " parts of eyes, nose, face etc " " slightly zoomed out part of img.

" " 3 ⇒ aggregate diff parts of face & tries to match it. Looks at even more zoomed out part of img.

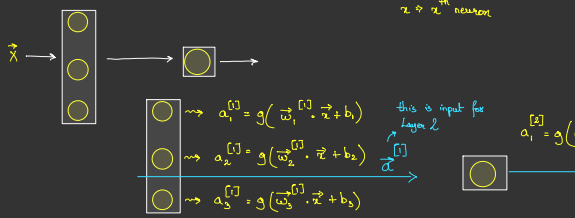
∴ diff neurons corresponds to diff size of regions of img.

→ Nobody explicitly programmed the layers to do so, it learnt this by itself.

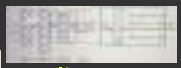
Neural net work layer

$a_2^{[3]}$ ⇒ 3^{th} layer

2^{th} neuron



More complex neural net works



$\vec{x} = \vec{a}^{[0]}$

$\vec{a}^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{bmatrix}$

for an arbitrary layer l & arbitrary unit j (i.e j^{th} neuron)

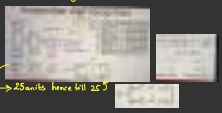
$a_j^{[l]} = g\left(\vec{w}_j^{[l]} \cdot \vec{a}^{[l-1]} + b_j^{[l]}\right)$

activation function (coz it gives activation values)

Inference: making predictions (forward propagation)

→ Classify the digit as Zero or one

→ 8X8 grid ; 255 → white ; 0 → black



25 units hence (8*8 = 25)

$\vec{a}^{[3]} = f(\vec{x})$ = probab of being a handwritten 1.

∴ this algo goes from left to right i.e from $a^{[1]}$ to $a^{[2]}$ to $a^{[3]}$...., this is called forward propagation ; backward propagation is used for learning

We have more hidden layers initially & it goes on decreasing as we get closer to output layer.

Interference in code

→ Coffee bean roasting example:

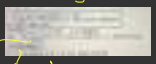
Digit classfn example

Data in tensor flow

→ Rows, columns ; order = rows x columns

→ In regression ckt we just dealt with vectors but here we will deal with matrices mainly as the datasets are immensly large ; solving by matrices is computationally efficient.

→ Technically tensor is more general than matrix, but here consider tensor is like just like matrix.



tensorflow array

this command converts it to numpy array

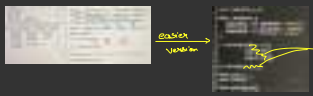
a_1 is initially a 1X1 matrix after converting it becomes 2D Numpy array.

Building a Neural network

In

→ The above code we explicitly pass the data from one layer to another, we can do this automatically by way mentioned below:

→ Digit classification example:



Forward prop in a single line

→ notation

→ Demonstration of single line of func.

General implementation of forward propagation

→

each column is weight vectors

units = W.shape[1]
↳ gives no of columns of W

a_out = np.zeros(units)
creating output arr of units size & initialising it with zero

w = W[:,j] ⇝ gives diff columns of W matrix

→ dense here takes input, process & returns output

→ Sequential is responsible for stringing all layers & finally returning the final ans.

W → matrix (caps)
w → array & vectors

Is there a path to AGI?

Artificial intelligence (AI)

- Artificial narrow intelligence (ANI)
  → Specific applications like self driving car, smart speaker etc.
- Artificial general intelligence (AGI)
  → AI capable robots; can do anything what human can do.

→ The progress is very slow as what a biological neuron does is way more complicated than a simple logistic or linear regression & we do not know how brain functions completely.

→ One learning Algorithm hypothesis

Same part of brain can learn multiple things

We do not know whether the brain has only 1 algo, even if it does we do not know what the algo is hence the progress is still far in future

How neural networks are implemented efficiently?

→ Vectorization

→ by vectorization the program becomes highly efficient

Matrix multiplication

→ Dot product

$$\begin{bmatrix} a \\ b \end{bmatrix} \cdot \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} a \cdot c \\ b \cdot d \end{bmatrix}$$

$$\vec{z} = \vec{a} \cdot \vec{w} = \vec{a}^T \times \vec{w}$$

Matrix multiplication code

→

→ another way of writing matrix multiplication

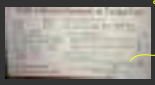
↳ A.T returns transpose of vector

→ implementation.

Module - 2

Tensorflow implementation

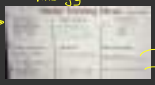
→ Handwritten digit classification example.



→ type of loss function → explained later.

Training details

Analogy



→ taking an avg. of this loss func gives cost func.
→ epoch: till where to run the minimization; basically similar to no of iterations

→ initially all these came under Keras library but later got merged into tensorflow.

→ to train data & minimize loss ⇒ Gradient descent

→ the keystep here is to calculate partial deriv. term, backward propagation algo does this very efficiently, call model.fit() for this.

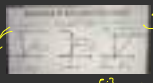
→ there are even better algos than gradient descents which are faster.

→ " " " " activation func than sigmoid which can make the pgrm run much faster.

Alternatives to sigmoid activation function

Awareness is not a 0 or 1 classif^n it may have various degrees hence we use another activation function called Rectified linear unit function, its given by

(ReLU) $g(z) = \max(0, z)$



→ Commonly used activation functions
→ taught later

Can be called both ways eg $a^{[3]} \cdot g(z)$ if $g(z) = z$ then,

$a^{[3]} = z$ which is like having no activation function.

Choosing activation functions

→ Different layers can have different activation functions.

→ Activation func. of output layer:

Depending on y characteristics of model the activation func. of output layer is decided:

→ if output layer only has 2 class of output: Binary classification ⇒ sigmoid func.

→ output can have both +ve & -ve values: Ex-stock price: Regression ⇒ Linear activation function

→ if " " only +ve " : Ex- house price: " ⇒ ReLU

Search about tan h and Leaky relu most of the places we prefer leaky rely over rely....for reason refer what are the things a function must qualify to become an activation function....(1) it must be non-Linear...(2) It must be diffable and deriv must not be zero else it will suffer with vanishing gradient problem, which is the case with tanh, sigmoid and step function (which is also an activation function but not at all preferred).....Leaky relu does not have this prob, basically leaky rely is max(0.001z, z) ....0.001 can be replaced with any small number....

→ Activation func. of hidden layers:

→ most common choice is ReLU. reasons:

(i) ReLU is faster

(ii) Relu becomes flat only in one part whereas sigmoid flattens at both ends, when the graph flattens gradient descent becomes slower, when it becomes too slow sigmoid func. also runs separately resulting in slowing down the overall pgrm.

tan h activation func. Swish activation func.

Leaky relu " "

Why do we need activation functions?

if we use linear activation function everywhere it just goes on to become a simple linear regression model, this defeats the purpose of making a neural network coz the model won't be able to learn anything more complex than a linear regression model.

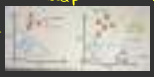
↳ just a linear regression

using all linear but sigmoid at end makes it a simple logistic regression func.

Multiclass classification

→ Here we classify things into multiple classes. Eg MNIST handwritten digit classification

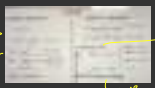
→ Soft max algo does this.



Softmax

→ It a generalisation of logistic regression which is binary class classification to multiclass classification context.

if n=2 it just becomes logistic regression



→ Sparse Categorical Cross entropy loss function

↳ if value of a_j is very close to one & it is correct then penalty is very low if it was wrong then it is heavily penalised

$a_1 = g(\vec{z}) = \frac{1}{1+e^{-\vec{z}}} = P(y=1|\vec{x})$

$a_2 = 1 - a_1 = P(y=0|\vec{x})$

loss = $-y \ln a_1 - (1-y) \ln (1-a_1)$

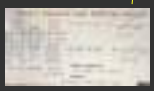
$\Downarrow$

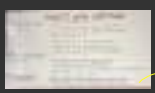
$\underbrace{-y \ln a_1}_{\text{if } y=1} \underbrace{- (1-y) \ln a_2}_{\text{if } y=0}$

Neural network with softmax output

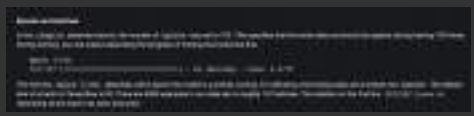
→ The layer having softmax activation is called softmax layer or softmax activation function.

Unlike others here $f(x)$ is func. of $\vec{z}_1$ to $\vec{z}_{10}$ (in model below)

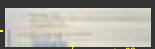




better version shown later

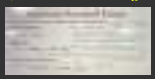


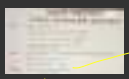
Improved implementation of softmax



↳ roundoff errors here

to reduce these roundoff errors we take some measures



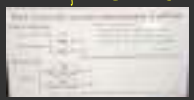


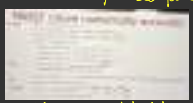
∵ it doesn't give probabs as to as we make the last line

if we don't take a seperate variable 'a' the roundoff errors'd

to do this we change activation func. of output layer to linear & modify the loss func.

→ for logistic regression either works fine but for softmax the errors gets pronounced, to avoid this we implement the correction in the following way:





∵ it doesn't give probabs as to as we make the last line

Multilabel classification - classification with multiple outputs

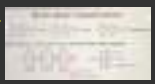
John Macoun
Scientist who popularised use of convolutional layer.

→ Here associated with each img there could be multiple labels

→ E.g. A self-driving car gets an img & asks (i) whether there is a car in front of me?
(ii) " " " bus " " " ?
(iii) " " " pedestrian " " " ?

→ There are 2 approaches to solve this:

(i) Treat it as 3 separate prob.; 3 models which investigate one of the three questions & these 3 combine to form the req. model (ML models)

(ii) Alternatively train one neural network with three outputs 

Advanced optimization

→ $w_j = w_j - \alpha \frac{\partial}{\partial w_j}(\vec{w}, b)$

→ if somehow we can control alpha's value to be large initially then decrease the algo will run even faster, this can be done by "Adam Algorithm"

Adam Algorithm

→ If algo notices the slopes are becoming smaller in same dirⁿ then it ↑ α's value

→ If " " " function has begun to overshoot then it automatically ↓ α so that the func can reach the minima efficiently.

Adam: Adaptive moment estimation

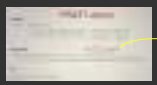
$$w_1 = w_1 - \alpha_1 \frac{\partial}{\partial w_1} J(\vec{w}, b)$$

$$w_2 = w_2 - \alpha_2 \frac{\partial}{\partial w_2} J(\vec{w}, b)$$
$$\vdots$$
$$w_{10} = w_{10} - \alpha_{10} \frac{\partial}{\partial w_{10}} J(\vec{w}, b)$$

$$b = b - \alpha_{11} \frac{\partial}{\partial b} J(\vec{w}, b)$$

If w_j or b keeps on moving in same direc: increase α_j
" " " " oscillating : decrease α_j

This is done through complex steps & is beyond the scope of this course

Implementation: 
→ it needs some initial value of α.
Try multiple α larger & smaller & pick the one which gives the best performance wrt. J

Adam algo. is more robust ∵ α is tuned in & works much faster than gradient descent.

Additional layer types

→ Dense layer: Each neuron output is a func of all the activation outputs of the previous layer.

$$\vec{a}_1^{[2]} = g\left(\vec{w}_1^{[2]} \cdot \vec{a}^{[1]} + b_1^{[2]}\right)$$

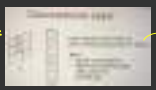
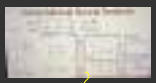
→ Convolutional layer: 
→ here it only looks at pixels of a designated area.

Illustration: ECG example: it has heights of peaks as input & outputs if the patient has any diagnosible heart condition.


the first neuron here looks at input 1-20 next one looks at 11-30 & so on....

1st & 2nd hidden layers: convolutional
output " : sigmoid

→ With convolutional layers we have many architecture choices such as how big window of inputs that a single neuron should look at & how many neurons should a layer have.

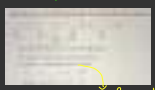
→ By choosing these effectively we can build layers more powerful than dense layers.

What is a derivative?

→ python package SymPy ⇒ used for calculus

Computation graph

→ like flowchart but of math



to calculate derivative backward prop is used coz if we were to calculate it from left to right we would have to calculate everything again & again

in forward prop.

Large neural network

→ for calculation of z_2, w^2, a^2, a', w' etc we start from left & move towards right ⇒ forward prop.

→ to find derivative we start from right & move to left i.e we first start from $\frac{\partial J}{\partial a^2}$ ⇒ backward.

→ if instead we use forward prop to calculate derivative for a model having n nodes & p parameters it would end up taking $(n \cdot p)$ steps ⇒ inefficient

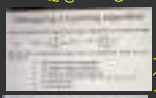
whereas forward prop will do it in $(n+p)$ steps ⇒ efficient

→ this is done in tensorflow & called auto diff or automatic differentiation.

Module-3

Deciding what to try next?

→ Debugging learning algo.



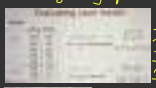
→ possible solutions

→ in this module we mainly learn about diagnostic i.e what to do next when our model is not working the way we want it to.

Evaluating a model

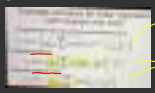
→ when we have just 2 features then we can just look at graph & have an idea about its bias & variance but this is not the case with models with many features. How do you tackle this problem?

→ We generally split data into 2 parts, training & test. We train model on training data & test its performance or how well it predicts with remaining 30% test data.



→ notations which will be used

→ for a model with high variance....



→ cost func of model

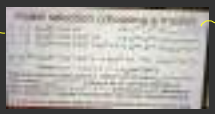
→ it doesn't include regularisation term.

J_{test} ⇒ fraction of test set that has been misclassified

J_{train} ⇒ " " train " " " "

→ for logistic loss.

Model selection and training/cross validation/test sets



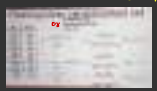
degree ←

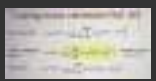
(notation used here) represents degree.

Training error is an over optimistic estimate of generalization error coz the model was trained on that dataset.

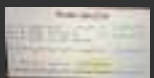
When we choose a model based on degree of curve looking J_{test} as reference then J_{test} becomes overly optimistic estimate of generalization error too. How to tackle this?

→ We modify the procedure of data splitting





regularisation is only done when minimizing.

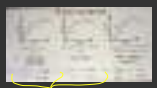


→ we hold CV set as reference while choosing degree of curve & use test data to predict the accuracy of model since the machine has never seen this before it will be a fair estimate of generalization error.

→ just to check first model

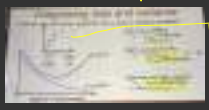
→ Do not touch the test set until the machine is learning, once it has learned fully then we can test its efficiency based on test set performance.

Diagnosing bias and variance



purple cross → CV set
red " → train "

this is for this example



Example of how an algo can have both high variance & high bias.
→ it initially overfit than for some reason it started to vary & then underfit
∴ high variance ∴ high bias

→ High bias & variance doesn't happen to linear regressions but can happen when we train neural network. Indicator → $J_{cv} \gg J_{train}$

Regularization and bias/variance

Establishing a baseline level of performance

→ Speech recognition example

Let's say training error = 10.8%

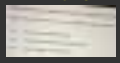
J_{cv} = 14.8%

We may think this is high but human level performance is 10.6%

So it's ok for J_{train} but J_{cv} must be reduced ∴ this model has variance prob.

∴ To evaluate a model we need some baseline

tells whether we have high variance prob. or not

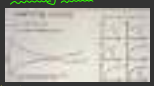


the baseline can be zero too for some models but for any model the first thing to do is to establish baseline for error.

tells whether we have high variance prob. or not

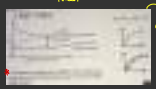
if both are high then its high bias & high variance problem.

Learning curve



as datapoints ↑
error ↑ as it is hard to fit them all

J_{train} will always be lower than J_{cv} coz the model was trained on that.



Case: fitting a st. line to a dataset

Curve flattens out coz no matter how much more data points we collect it will have almost same avg. error

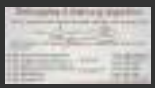
Example
as we ↑ datapoints we can find a better 2nd degree curve to fit them well.

→ model may do so well on training data that it may surpass humans but J_{cv} will be higher but as we get more data this is resolved initially

→ It might be computationally expensive doing this process again n again but it does help get an idea of how model is working.

Deciding what to try next revisited

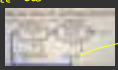
→ If features are more than it gives the model too much flexibility to fit very complicated models.



Bias/variance & neural networks

→ We always have to trade off between bias and variance but in neural network this is not necessarily the case.

Large neural networks are low bias machines

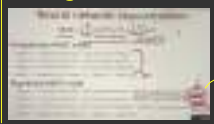


→ coz it has high variance.

Will large neural network create a high variance?

No, a larger model with well chosen regularisation will do better or atleast as well as smaller model.

But it becomes more computationally heavy.



→ we can choose diff lambda for diff layers.

As long as training data isn't too big the neural network model will have low bias ; often we face high variance probs.

Here there is a unit missing continued on next page

Error in metrics for skewed datasets

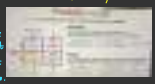
→ Example,

Let's say a model has to predict a rare disease from input ; it has 1% error & if only 0.5% of population has disease then this error is very very high. Instead a simple command print("y=0") has 0.5% error.

Hence we cannot look at error % & all about model when there is a rare class or "skewed class" involved.

(synonym)

We use common pairs of error metrics Precision & recall to tackle these situations.

→ 

* Model will never predict 1 ∴ first two boxes in first row become zero.

If we take previous print("y=0") example:

recall would be zero

→ In general algo with zero recall or zero precision is useless

→ If an algo predicts zero always then precision is undefined *

Precision tells how accurately model is predicting.

&

Recall tells of all having disease how many were diagnosed.

Both must be as high as possible for a good model.

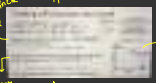
Trading off precision and recall

→ Practically there is a tradeoff btw them. ↔ converse is also true.

→ On ↑ threshold from 0.5 to 0.7 precision ↑ but recall ↓ (cz out of ppl who have disease will be predicted comparatively)

↓ (cz now we will be more confident that he has the disease)

increase the threshold if this is the case.



→ tradeoff btw precision & recall

lower the threshold if this is the case

→ Generally we manually pick the threshold

→ If we wanna do it automatically then we use another metric called F1 score

F1 score

Challenge: we aren't evaluating algo using 2 metrics

We need just a single score which can directly tell us which is the best algo.

Method - 1

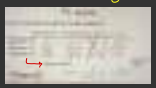
take avg of P & R ie $\frac{P+R}{2}$ ⟿ not that good

Method - 2 ⇒ F1-Score

It combines P & R & gives more emphasis to whichever is lower.

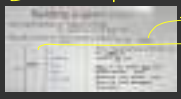
$$F1 = \frac{1}{\frac{1}{2}\left(\frac{1}{P} + \frac{1}{R}\right)} = \frac{2\,PR}{P+R}$$ [Harmonic mean]

higher the score better the model

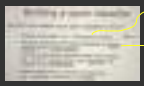


Iterative loop of ML development

Ex - email spam classifier



→ say
→ present or absent or no of occurrences



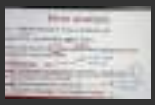
→ creating fake emails to catch hold of spammers
→ backtrack email from where it came

Choosing right ideas are very important

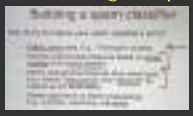
Error analysis

→ First check for bias & variance, then this one

→ say we have 1000 misclassified among 5000 then pick a random subset of 100 and look at them and classify

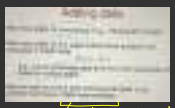


→ Then start addressing major probs like pharmaceutical spam rather than misspelled words first.

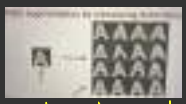


→

Adding data

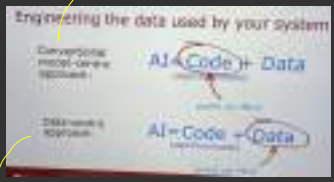


→ take existing training example to create a new one by rotating or taking mirror ing etc.



A grid is placed & warped

Data augmentation can also be done to audio datasets by introducing noise.



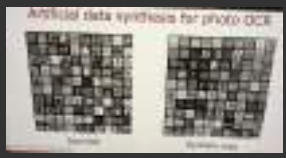
→ classic method followed from long back

→ thanks to new tech we can spend more time collecting data rather than on code.

Data synthesis

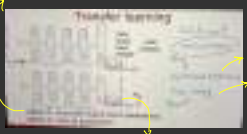
Ex data synthesis for photo OCR (text recognition)

we can just type in random fonts take screenshots of it with diff colors and contrasts



Transfer learning : using data from a different task

If training set is very small option 1 works better else option 2.



→ We can take a model pre-trained on a large dataset, change its output layer & fine tune it to work as we need it.
→ Refers to pre-training part of model
→ Refers to modification of out layer part.
→ We can download someone else's model & then fine-tune it.

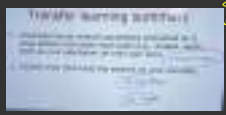
new weights only in output layer, all others can be kept same too.

→ The input type for pre-trained & our model must be same. Ex same pixel images, audio input etc.

Why does this work?

Coz the model recognizes generic features which would be applicable on our inputs too.

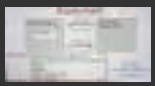
→ generic features



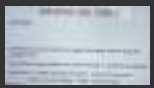
If the pre-trained model is highly efficient then even so-many images while fine tuning would be enough to build a good model.

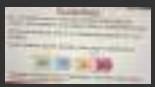
Full cycle of a machine learning project

After training model or deployment we may again collect data



Fairness, bias and ethics

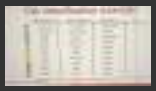




Module-4

Decision tree model

Cat classification example



In decision trees features take on discrete values

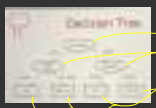
like face is round or not round

ear - floppy or pointy

whiskers are present or absent

[Later we will see features taking more than two discrete values or continuous value features later]

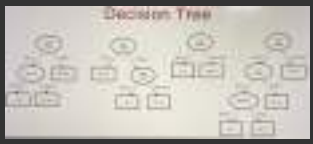
Oval/rectangles are called nodes



root dec leaf

Amongst all these decision trees some will do good & some bad on training set on CV or test set

∴ Job of algo is out of all pssbl. dec. trees to pick one that does well on training set & then also generalizes well to new data



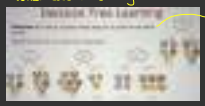
Learning process

Process of building a decision tree

→ What feature must be used at root node?

→ What " at subsequent nodes.

What are some key decisions we made.....

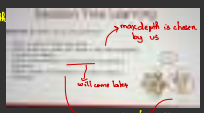


→ this is decided based on purity

Why we prefer trees with lower depths?

→ So that tree doesn't get too big and unwieldy

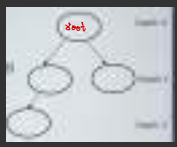
→ Big trees are more prone to overfitting.



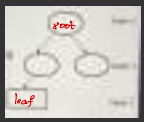
Ex how is cat

∴ just classify it as not cat

ie basically if we feel that splitting further down is unworthy.

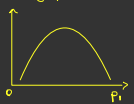


if max depth is 2



Measuring purity

→ Entropy function is denoted by $H(p_1)$

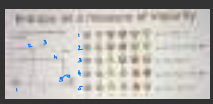


p_1 = fraction of examples which are cat

$$\text{purity} \propto \frac{1}{H(p_1)}$$

$$\text{Impurity} \propto H(p_1)$$

at 50% correct & 50% wrong we have highest entropy



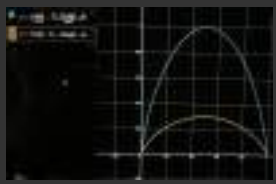
Now,

p_1 = fraction which is right

p_0 = " " " wrong

$p_1 + p_0 = 1$

$$H(p_1) = -p_1 \log_2(p_1) - p_0 \log_2(p_0) = -p_1 \log_2(p_1) - (1-p_1) \log_2(1-p_1)$$



→ We take log base 2 coz result will be at height 1 which makes it easier to predict probabilities

→ On taking base e graph shifts below just that setting a threshold value becomes comparatively difficult

Note: for computational purposes $(0)(\log_2 0) = 0$

To summarize,

Entropy function is a measure of impurity of a set of data

Starts from 0 goes to 1 & comes back to 0 as func. of fraction of true examples in sample

Gini criteria

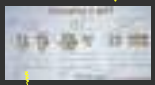
There are many other func. which look like entropy function like gini criteria which will be present in open source package. This can also be used to make decision trees but for the sake of simplicity, in this course we will just look at entropy function, which will work well for most of the applications.

Choosing a split : information gain

→ The way we decide what feature to split on at a node is to check which reduces entropy the most.

→ reduction of entropy is called information gain

How to compute information gain ?



↓
weighted avg

Info gain & purity

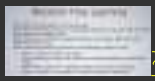
∴ which feature has higher info gain that should be chosen.

↓
weighted avg.
→ fraction which is correct.

$$\therefore \text{Information gain} = H(p_1^{root}) - \left(w^{left} H(p_1^{left}) + w^{right} H(p_1^{right})\right)$$

Putting it together

pieces of building a decision tree



→ stopping criteria

→ first check at root node for which feature has highest info gain.

→ Repeat process at all nodes

→ We build this using recursions if we are doing it by scratch hence its called recursive splitting.

∴ we build an overall decision tree by putting together various small sub-decision trees.

→ Some libraries have good default choices for choosing maximum depth

→ larger the tree will be for higher max depths & this is like fitting higher degree polynomials while training large neural networks, it will usually allow dec tree to learn more complex model but will also risk overfitting.

→ In theory, we can use cross-validation set to pick max depth parameters, i.e by picking many values & finally choosing one which does best on CV set. In practice open-source libraries have even better ways to choose this parameter for us.

→ Another way is to stop when info gain by splitting further is below a threshold value

→ " " " " " " no of example in the node " " " " .

Using one-hot encoding of categorical features

→ How to tackle features which can take on more than 2 values

Let's say there are 3 ear shape oval, pointy and round

Let's split ear shape into 3 features

⇒ Ear shape split into 3 features

→ If a categorical feature can take on k values, create k binary features (0 or 1 valued)

→ The value which is 1 is the hot feature

→ When features have things like present, absent or yes, no, not known etc are called categorical features if we convert them to 1,0 then it becomes numerical & then that can be fed to neural networks too.

Continuous valued features

→ What if a feature can take on continuous values?

How do you tackle it?

→ Let weight also be a feature in the example, then,

→ We pick diff thresholds & compute info gain, the one which gives max. info gain is chosen.

→ One convention would be to sort all examples acc to weight or acc value of this continuous feature & take all the values that are midpoints between the sorted list of training examples as the values for consideration for this threshold over here.

Regression tree

→ Let's say you have to predict weight of the animal.

Note: There is nothing wrong with a decision tree which chooses to use same features to split along left & right

→ Once the algo reaches to a leaf node it predicts the value as the avg of all training example which reached that leaf node

→ If you were constructing a dec tree from scratch using this dataset in order to predict the weight, key decision is to choose which feature to split on...

→ Suppose there are 3 possibilities as follows, how do you choose which is best?

→ In classification we tried to reduce entropy, here we try to reduce variance of weight i.e y variable.

→ Total weighted avg of variance of a node is subtracted from variance at root node ⇒ this quantity is called reduction in variance.

value of reduction in variance ∝ good split

→ we go on splitting this way and calculating until stopping criteria is not reached

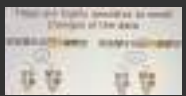
$$\text{Reduc. invariance} = V_{root} - \left(w_{left} V_{left} + w_{right} V_{right}\right)$$

V = variance

Using multiple decision trees

→ Disadv of using single tree: its highly sensitive to new data

→ To tackle this we build a bata decision trees, its called a tree ensemble.



→ Ex: If we change just 1 training example we get an entirely diff tree which makes this model not so robust.

→ Now, consider we have this ensemble of trees. If a new data comes it is run in all of em & they are made to vote. Majority vote will become the final prediction

Sampling with replacement

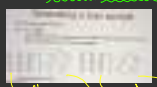
→ This technique is used to build an ensemble of trees

→ Pick, put it back, pick again .....

→ we take all our data, put it in a bag & perform sampling with replacement (same examples can repeat too)

→ By doing this we end up with data similar to training data & then we make dec tree for each of these

Random forest algorithm



so on upto B times → Bagged decision tree ensemble

1st sample → its tree; 2nd sample → its tree

→ One disadv of this method is we end up with more or less same split at root node. To solve this: Enter Random forest.

→ This makes algo explore how small change in data affects the model, so when new data comes prediction will be more or less accurate.

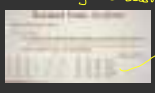
→ We generally build 64-128 trees after a certain no of trees it just becomes computationally heavy & not much better at performance

XGBoost

→ One of major algos used to build ensemble of trees.

→ For successive trees it picks those examples (putting a ?) which had been misclassified earlier

→ This way we boost the algo which ends up making the model work very nicely

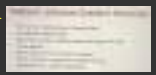


→ these had been misclassified and now picking them is higher probab.

→ Any tree k will have higher chances of picking these examples which had been mis-classified by prev trees i.e from 1 to k-1. How it is done is complex math which can be explored later,...

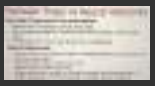
XGBoost stands for extreme gradient boosting

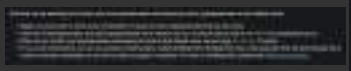
→ It assigns diff weights to diff training examples hence it need not make too many random sets

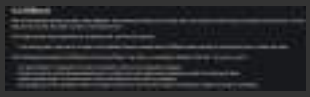


Implementing ⇒

When to use decision trees?







4.1 Decision Tree

In this section, we'll work with the Decision Tree we previously learned, but now using the

There are several hyperparameters in this function. See object-hyro-fields.match. We will use only several of them and also we will not perform brute-force search for hyperparameter tuning in this lab but you are encouraged to do so and compare the results 😉

The hyperparameters we will use and investigate here are:

- min_samples_split: The minimum number of samples required to split an internal node.
  - Choosing a higher min_samples_split can reduce the number of splits and may help to reduce overfitting.
- max_depth: The maximum depth of the tree.
  - Choosing a lower max_depth can reduce the number of splits and may help to reduce overfitting.

4.2 Random Forest

Now, let's try the Random Forest algorithm also, using the Scikit-learn implementation.

- All of the hyperparameters found in the decision tree model will also exist in the algorithm, obvious as Random Forest is an ensemble of many Decision Trees.
- One additional hyperparameter for Random Forest, in addition to n_estimators, is n_jobs that tells it the number of decision trees that make up the Random Forest.

Remember that for a Random Forest, we randomly choose a subset of the features AND randomly choose a subset of the training examples to train each individual tree.

- Following the lectures, if is the number of features, we will randomly select $\sqrt{n}$ of those features to train each individual tree.
- Note that you can modify this by setting the `max_features` parameter.

You can also speed up your training jobs with another parameter, `n_jobs`.

- Since the training of each tree is independent of each other, it is possible to train each tree on a parallel.
- So setting `n_jobs` higher will increase how many CPU cores it will use. Note that the numbers may relate to the maximum available CPU may impact on the overall performance of your PC and even lead to freezes.
- Changing this parameter does not impact on how the machine can produce the training time.

We will run the same script again, but with another parameter, `n_estimators`, where we will choose between 10, 50, and 100. The default is 10.

Unsupervised Learning

Module - 1

Welcome

→ Clustering, anomaly detection ⇝ Module - 1

→ Recommender sys. ⇝ " - 2

→ Reinforcement learning ⇝ " - 3

What is clustering?

→ It sees unlabelled data points & groups them & tries to find pattern in them.

→ Ex Grouping similar news, grouping DNA data, astronomical data analysis

K-means intuition

→ Randomly picks points equal to no of clusters we need (say we need 2 here)

→ It first assigns them it starts moving the cluster centroid.

→ First it assigns centroids at random then it goes through all training examples & assign each point to its closest centroid.

→ Then it moves the centroid to avg location of all points assigned to that centroid.

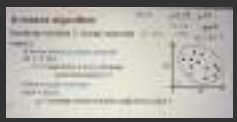
→ Now again it checks all training examples & reassigns them then again

→ when after an iteration nothing changes then it means that the algorithm has converged.

K-Means algorithm

→ All μ_i are vectors having same dimensions as training examples.

→ L2 norm way of writing distance ⇒ $\|x^{(i)} - \mu_k\|$; in algo we minize squared dist $\min_k \|x^{(i)} - \mu_k\|^2$



→ What if a cluster has zero training examples assigned to it?

→ M-1 ⇒ eliminate that cluster ⇝ more common method

M-2 ⇒ reinitialize that cluster

Optimization objective

$c^{(i)}$ = index of cluster $(1,2,\dots,K)$ to which which example $x^{(i)}$ is currently assigned

μ_k = cluster centroid k

$\mu_c^{(i)}$ = cluster centroid of cluster to which example $x^{(i)}$ has been assigned

Cost function : Distortion function

$$J(c^{(1)},\dots,c^{(m)}, \mu_1,\dots,\mu_k) = \frac{1}{m}\sum_{i=1}^{m} \|x^{(i)} - \mu_{c^{(i)}}\|^2$$

sq. dist btw all points & their cluster centroids

∴ to minimize this machine automatically starts to assign points to closest centroids

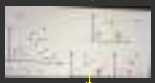
And to move cluster centroids.....

it assigns centroid to mean value of coordinates of examples assigned to that centroid.

Initialising K-means

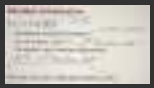
→ no of clusters (K) must be less than training examples(m) i.e $K < m$; else there would be less than 1 training example per cluster

→ Randomly pick K training examples & assign $\mu_1 \dots \mu_k$ to them.



this is a local optima

→ so we randomly initialize & build multiple models & then compute cost func for them the one with lowest one will be chosen.



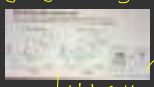
→ Generally we randomly initialize it 50-100 times ; doing it too many times just make it computationally heavy.

Choosing the number of clusters

→ Elbow method

→ J vs K is plotted. K is plotted ..... the no of clusters till which J decreases rapidly & after which it slows down is chosen as our preffered value of K ; that particular point is called elbow cz the curve looks like it

→ K is often ambiguous coz many-a-times J vs K curves are many-a-times smooth & elbow cannot be spotted, choosing no of clusters just to reduce J is not good enough then we will always end up with largest value of K.



→ trade off btw how good img looks vs how much you can compress

↳ trade off btw no of t-shirt size & cost of manufacturing.

Hence most of the times its us who decide how many clusters are required according to our need.

Anomally detection

Finding unusual events

Anomally detection example : Aircraft engine features

x_1 = heat generated

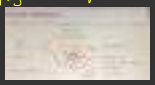
x_2 = vibration intensity

Dataset : $\{ x^{(1)}, x^{(2)}, \ldots, x^{(m)} \}$ New engine : x_{test}

Density estimation : first thing we do with dataset is build a model which calculates probability of points being seen in dataset

if $p(x_{test}) < \varepsilon$ ⇒ flag that example as anomally

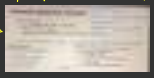
(we choose this)



Usage :

fraud detection o pssl features: how often log in ? no of webpage visited ? transactions ? typing speed ? If flagged security team looks at it closely may give captcha or trigger alarms etc....

manufacturing ⇒



Gaussian (Normal) Distribution [check out dalton board]

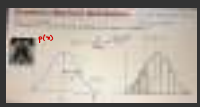
or bell-shaped distribution

x → random variable

probab. determined by μ & σ^2

Say we get a 100 no & plot histogram it will look like this

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$



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

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

Area under curve = 1 coz max probab can be equal to 1.

these are technically called maximum likelihood for μ & σ some use $\frac{1}{m-1}$ instead of $\frac{1}{m}$ it makes very little diff here but it helps in other calculations.

→ if x is a single no this works, when x has multiple variable .....

Anomally detection algorithm

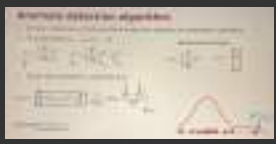
$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

$$p(\vec{x}) = p(x_1; \mu_1, \sigma_1^2) * p(x_2; \mu_2, \sigma_2^2) * p(x_3; \mu_3, \sigma_3^2) * \ldots * p(x_n; \mu_n, \sigma_n^2)$$

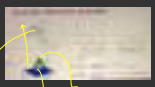
$$= \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2)$$

this formula works well even if features are not statistically independent.

x_i is a list of nos having m nos i.e. numbers recorded in x_i feature of all training examples...



→ even if one feature is less than normal then it will flag as an anomally coz $p(x_j)$ say becomes very small hence reducing $p(\vec{x})$

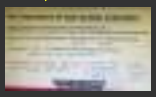


this comes here

→ the graph's height represents $p(x_1) * p(x_2)$

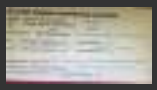
Developing & evaluating an anomaly detection system

In practical situation its ok even if there are some anamolous example in training set its ok



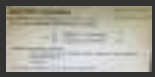
In general its better if CV & test set have some anamolies in them.

When we do collect data often we know that a small number of examples are anamolous hence we use them with labels to train, check & evaluate. For example,



if the number of flawed engines say are very small then we can omit test set & just make CV set.

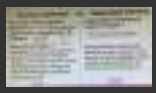
Evaluating a model



∴ having some anomalies helps the model, why not run a supervised algo. on this data.

Anomaly detection vs supervised learning

→ ∵ future anomalies maybe entirely diff from prior ones its safer to use anomaly detec. (if our data is of this type) coz anything deviating away too much will be flagged.



hackers find new ways to hack into systems hence we use anomaly detection for security also.

Choosing what features to use

→ In unsupervised learning choosing what features to use & what not to becomes very important.

→ Try to make features gaussian.

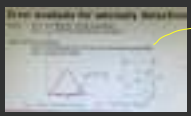
↷ smaller values of c is more effective

We transform the features to make it gaussian

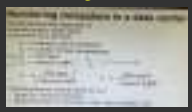
Directly log x might give errors if x was zero ∴ we write log(x + 0.001) to avoid these errors.

→ Same transformation must be applied to CV & test sets too.

Error analysis



→ then we have to look at that example & try to figure out why was it an anomaly & try to come up with a new feature which can recognise those kinds of anomalies



Module-2

Recommender systems

Making recommendations

→ We have users as well as some no of items

n_u = no of users

n_m = " " items (have movies)

$r(i,j)$ = 1 if user j has rated the movie

$r(1,1) = 1$

$r(2,1) = 0$

$r(\text{item no}, \text{user no})$ $y(\text{item no}, \text{user no})$

$y(i,j)$ = rating given by user j to movie i (defined only if $r(i,j) = 1$

→ Look at movies user hasn't rated yet, predict their rating for it, if that's high then recommend it to them.

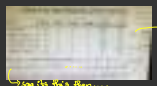
$x^{(i)}$ is defined for a item & has n nos in it.

$w^{(j)}$ " " " user " " " " "

Using per-item features

→ Let's assume we have features of each movies.

n : no of features

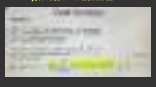


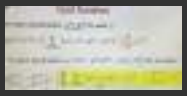
→ features of a movie

for user j's rating prediction of movie i = $w^{(j)} \cdot x^{(i)} + b^{(j)}$ → this looks like linear regression

Except there are n diff linear regression model i.e one for each user.

Cost function





We do not divide by $m^{(j)}$ for recommender sys

$$J\left(\begin{matrix} w^{(1)}, w^{(2)}, \dots, w^{(n_u)} \\ b^{(1)}, b^{(2)}, \dots, b^{(n_u)} \end{matrix}\right) = \frac{1}{2} \sum_{j=1}^{n_u} \sum_{\substack{i=1: \\ r(i,j)=1}}^{n} \left[w^{(j)} \cdot x^{(i)} + b^{(j)} - y^{(i,j)} \right]^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} \left(w_k^{(j)} \right)^2$$

all nos in $w^{(j)}$ for all j will be penalized to avoid overfitting

w is a weight vector having 'n' ele.

What if we do not have any prior features.....

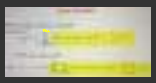
Collaborative filtering algorithm

Let's assume we already know $w^{(j)}, b^{(j)}$ of all users

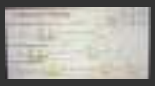
→ our prediction of x_i. The more users we have more accurately we can determine $x^{(i)}$ (who have rated more movies)

Cost function

Given: $w^{(1)}, b^{(1)}, w^{(2)}, b^{(2)}, \dots, w^{(n_u)}, b^{(n_u)}$



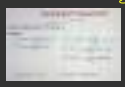
Putting together both of above cost functions we get Collaborative filter algorithm



∴ Overall cost func. becomes....

$$\min_{\substack{w^{(1)}, w^{(2)}, \dots, w^{(n_u)} \\ b^{(1)}, b^{(2)}, \dots, b^{(n_u)} \\ x^{(1)}, x^{(2)}, \dots, x^{(n_m)}}} J(w, b, x) = \frac{1}{2} \sum_{(i,j):\, r(i,j)=1} \left(w^{(j)} \cdot x^{(i)} + b^{(j)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} \left(w_k^{(j)} \right)^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} \left(x_k^{(i)} \right)^2$$

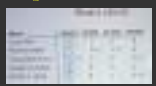
How to minimize : Gradient descent

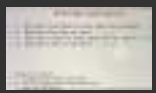


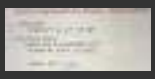
→ Name stems to sense that multiple users have rated the same movie collaboratively, giving us sense of what this movie may be like, that allows to guess what are appropriate features for that movie, this in turn allows us to predict how other users that haven't yet rated that same movie may decide to rate it in future.

Binary labels : favs, likes and clicks

1 : liked 0 : disliked ? : unrated

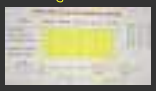






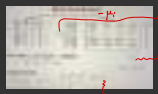
Mean normalisation

→ Algo becomes more efficient if we perform mean normalisation.



Let's say a new user Eve comes & has not rated any movies yet, now if we run the model it will predict that all ratings by Eve is zero which is not reasonable

To counter this we use mean normalization.

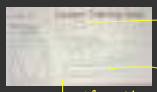


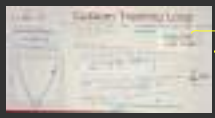
This is a more reasonable prediction.

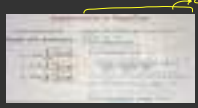
→ Due to mean normalisation algo becomes a little faster too.

→ We can also normalise columns instead of rows : can be done when new movie arrives but generally its not done in this way coz we don't know how much the user may like it.

Tensorflows implementation of collaborative filtering



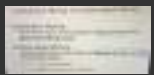


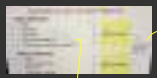


Finding related items

5 or 10 similar ones are listed & recommended.

Collaborative filtering vs Content based filtering





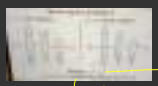
from $x_u^{(j)}$ we compute $v_u^{(j)}$ . . . must be of same size

$x_m^{(i)}$ . . . $v_m^{(i)}$

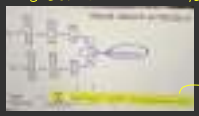
$v_u^{(j)} \cdot v_i^{(m)}$ gives prediction of rating

Deep learning for content based filtering

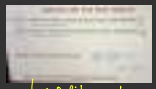
→ User n/w & movie n/w can have diff no of hidden layer & diff no of units per hidden layer but output layer must have same no of units.



→ We can combine both n/w to one model also



Using this to find similar items



→ This can be computationally very heavy if no of movies are very large.

How to tackle this ?.....

PTO....

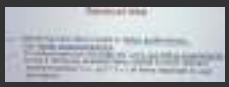
Recommending from a large catalogue

→ Its done in 2 steps retrieval & ranking

this can be pre computed

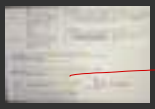
hence we only need to compute V_u and we are done

The no of movies we wanna retrieve is our choice



Ethical issues of recommender systems

Tensorflow implementation of content based filtering



L2 normalization

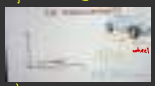
& reducing dimension to one.

Principal Component Analysis

Reducing the number of features

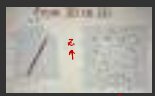
→ PCA helps in visualisation of data, say we have 50 or 100 or 1000 features, with PCA we reduce this to 2 or 3 features so that we can plot it.

→



In these we can just take feature that varies & drop the other one

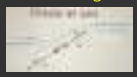
z-axis is combination of features x_1 & x_2



PCA algorithm

→ Features should be normalized first to have zero mean, if needed after this feature scaling must also be applied.

→ PCA reviews all axis & starts to make its own axes, on a good axis the data must be spread widely ie it must capture all or maximum variation of data.



If we are reducing to 1D data then algo will choose this principal component or first principal component.

→ How is point projected on axis

(position vector of point) • (unit vector in dir^n of z-axis) = dist of point from origin of new axis

2nd principal component will be 1st to 1st principal component.

3rd " " " " " " 1st & 2nd " "

→ here x_1 & x_2 both are equally treated & algo tries to capture maximum variance.

here y is a label & has special importance

Algo minimizes dist. parallel to y axis

reconstruction of original data (approximately same data will be constructed)

dist of point from origin of new axis • unit vec along z = approximate original data points.

PCA in code

steps

higher the value more the data is retained

$z_1 + z_2 = 1$ ie 100% of data is retained

we don't do this now coz we have machines which upload data very fastly

PCA also has cost its easier now to upload xcalibur than perform PCA

Module - 3

What is reinforcement learning?

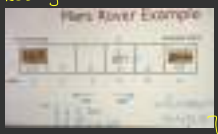
→ Reward function : when model does something desirable its rewarded & when it does something undesirable its penalised.

→ Applications of reinforcement learning : controlling robots, factory optimization, financial (stock) trading, playing games (including video games)

Mars rover example

→ The rover can be in any of the six boxes or 'states'

→ Let's say its at 4



associate with state

state, action, reward, new state

(s, a, R(s), s')

1 & 6 are terminal states

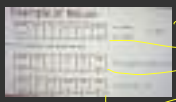
→ at state 4, go left, reward = 0, s' = 3

The return in reinforcement learning

R_i ⇒ reward of i^{th} state

γ ⇒ discount factor (generally a number close to 1)

$$\text{Return} = R_1 + \gamma R_2 + \gamma^2 R_3 + \dots.$$



→ for this example $\gamma = 0.5$

→ from diff states if we go left what will be the reward is written

→ " " " " " right " " " " " "

→ A hybrid to get max reward is written.

→ Discount factor also pushes away the -ve rewards ie further it pushes away lesser the -ve value will be added.

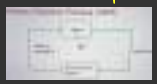
Making decisions : policies in reinforcement learning

→ Our job in reinforcement learning is to come up with a policy (π) which takes state (s) as input & map it to an action (a) which is to be performed.

state (s) —policy (π)→ action (a)

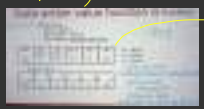
Review of key concepts

→ Formalism of reinforcement learning application is called Markov Decision process (MDP), it says that the future only depends on current state & not on anything that might have occurred prior to getting to the current state.



State - action value function definition (or Q function)

→ Generally denoted by Q(s, a)



→ Let's say these are the optimal behaviours at different states

Q(2, ←) ⇒ return it gets once it goes left from 2 then behaves optimally

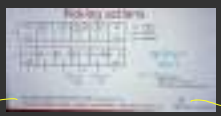
Q(2, →) ⇒ " " " " " right " " " " "

Q(s, a) = 'Return' if you

→ start in state s

→ take action a (once)

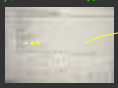
→ behave optimally after that



→ Q is written this way in some books

→ So if we had a way to compute Q(s, a) for all states then we can just go on picking $\max_a Q(s, a)$ for optimal behaviour.

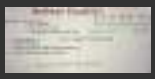
State -action value function example



→ ∵ $\gamma = 0.9$ here the rover waits and goes towards 100

Bellman Equation

→ This helps us compute $Q(s,a)$



Example

intuition of bellman eqn

$$Q(s,a) = R(s) + \gamma \max_{a'} Q(s',a')$$

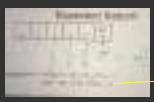
$$= R(s) \quad \text{(if } s \text{ is terminal state)}$$

$R(s)$ is also called as the immediate reward.

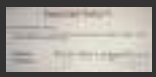
Random (stochastic) environment

→ Due to some randomness in environment the system may not perform as commanded. Ex on commanding the rover to perform a particular action, it does so 90% of times but 10% of times it does not.

→ Here in stochastic we have multiple returns, maximizing return doesn't make sense eg return is a random number instead we maximize the avg. value of sum of discounted rewards



→ Mathematically it is written this way



Ex: its in state 3 it could be 2 or 4

Modified Bellman eqn

$$Q(s,a) = R(s) + \gamma E\left[\max_{a'} Q(s',a')\right]$$

Example of continuous state space applications

→ Up until now the rover could be in one of six positions, these are discrete states, say now that a vehicle (truck) can be any where on a line, this is continuous state space.

State of truck is given by its position in x,y dirn, its orientation, its vel. in x & y dirn & its angular vel. $\dot{\theta}$ ie how fast it can turn.

State of helicopter includes its position in x, y, z dirn, its roll, pitch, yaw & their ϕ θ ω rate of changes

∴ in a continuous state MDP a state is a vector comprising of many numbers which could take on any range of values.

Lunar lander

→

1 if left leg is touching ground else 0

1 if right

Goal: to learn a policy π that, given s picks $a = \pi(s)$ so as to maximize return $\gamma = 0.985$ for this. [this is given, generally for these high value of γ is chosen]

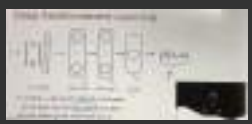
Learning the state -value function

→ Input $\vec{x} = \begin{bmatrix} s \\ a \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \\ \dot{x} \\ \dot{y} \\ \dot{\theta} \\ l \\ r \\ a \end{bmatrix}$

actions = nothing, left, main, right

0 or 1

Output = $Q(s,a)$



If the neural network is trained properly then for any state s All four Q values can be calculated & the max one among them can be performed.

Approach: We will use Bellman's eqn to create a training set with lots of examples of x & y, & use supervised learning then to learn a map from x to y i.e from state -action pair to target value of $Q(s,a)$

$$Q(\underbrace{s,a}_{x}) = \underbrace{R(s) + \gamma \max_{a'} Q(s',a')}_{y} \quad \text{we need } f_{w,b}(x) \approx y$$

We start performing random actions

$f_{w,b}(x) \approx y$

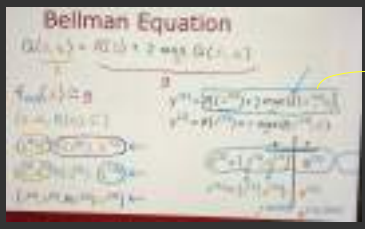
$(s, a, R(s), s')$

$(s^{(1)}, a^{(1)}, R(s^{(1)}), s'^{(1)})$

$(s^{(2)}, a^{(2)}, R(s^{(2)}), s'^{(2)})$

⋮

→ each of these tuples is one example in training set



→ initially Q func is a random guess

→ after forming these we train our model based on this data & try to come up with a func. which takes in x & outputs 'y'

Mnih et al

→ the algo stores the most recent 10000 examples ..... this is called replay buffer

→ Q_{new} is a better estimate of Q func. as we run this repeatedly long enough we get a pretty good estimate of Q func.

This algo is called DQN or deep-Q-network

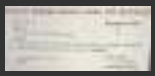
Algorithm refinement: improved neural network architecture

→ instead of having one output for $Q(s,a)$ & running it four times for four different actions we have 4 outputs to predict all four $Q(s,a)$ at same time, this also helps calculating $\max_{a'} Q(s',a')$ faster

How is action input here then?

Algorithm refinement: ε-greedy policy

→ Even while learning we need to pick some actions, how do we do it? For this we use ε-greedy policy



Now for picking subsequent actions we can either pick

option 1 or option 2

✗ ✓ coz say while initialising Q, $Q(s,\text{main})$ was low then algo never does it if option 1 is picked so we introduce a little exploration i.e. ε.

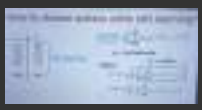
We can initially have high ε & then reduce it to a lower value like 0.01

→ In supervised learning choice of hyper parameters may make algo. 3 to 4 times slower but in unsupervised it may make 10-100 times slower .... so this is very important.

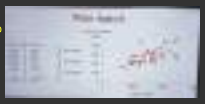
wrong

Algorithm refinement: mini-batch & soft update

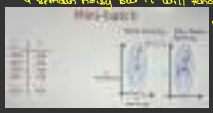
Mini-batch (applicable to both supervised & unsupervised learning)

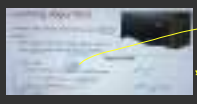


say if we have very large value of m algo becomes too slow ∴ we use only m' of these per batch.

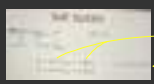


The contour of mini-batch may not converge & remain noisy but it will tend toward global minima. This is okay coz each iteration is practically inexpensive.





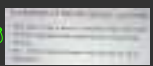
mini-batch. Say an iteration was bad & Q_{new} is worse than Q if we overwrite Q as Q_{new} then it would be bad. To prevent this we use soft update.



→ hyper parameters we choose. These are re-picked to add upto 1. The higher the coef of w_{new} is more abruptly w changes.

→ By soft update its observed that algo converges more reliably avoiding oscillation or divergence or other undesirable prop.

State of reinforcement learning



Refer last Moon rover lab for more info on reinforcement learning, several papers have been mentioned in labs to study from, refer those at times….do download all lab files

Activation func.

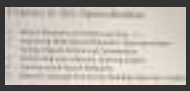
Leaky relu $\max(\alpha z, z)$

ELU $f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \le 0 \end{cases}$

SELU $g(x) = \lambda f(x)$ → will self normalize mean 0 std dev 1 but model should be deep, look Gaussian initialized Gaussian more.

SELU > ELU > Leaky Relu > Relu > Tanh > Sig

Deep Learning Specialisation



Neural Networks & deep learning

Module - 1

What is a neural network?

Supervised Learning with neural networks

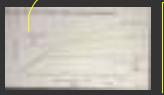
→ Convolutional NN
→ Recurring NN

→ for 1-D data

Structured data : database with columns, rows etc.

Unstructured data : - - img, audio or text to translate.

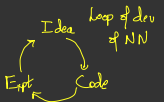
Why is deep learning taking off?

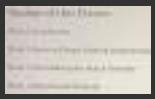


hierarchy is not perfectly defined here, depends on how well features are engineered

Why is sigmoid bad?

Gradient becomes very low
due to which parameters get updated very slowly.





Module - 2

Binary classification

if img is 64X64 then its gonna have 64X64X3 → R,G,B = 12,288 entries as input

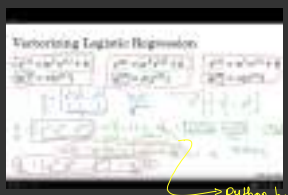
Logistic Regression

$$f(w,b,x) = \frac{1}{1+e^{-(wx+b)}}$$

Logistic regression cost function

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

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



→ Python broadcasting

Derivation

$$L = -(y \ln a + (1-y) \ln(1-a))$$

$$\frac{dL}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$\frac{dL}{da} = \frac{a-y}{a(1-a)}$$

$$\sigma(z) = a \quad \therefore \frac{da}{dz} = \frac{d}{dz}(\sigma(z))$$

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

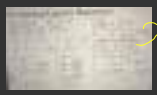
$$\sigma'(z) = \frac{+e^{-z}}{(1+e^{-z})^2} = \left(\frac{1}{1+e^{-z}}\right)\left(\frac{e^{-z}}{1+e^{-z}}\right)$$

$$= \sigma(z)\left(1-\sigma(z)\right)$$

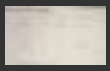
$$\therefore \frac{da}{dz} = \sigma(z)\left(1-\sigma(z)\right) = a(1-a)$$

$$\frac{dL}{dz} = a - y$$

Vectorizing Logistic Regression



Broadcasting in python



Note on numpy/python vectors

$a = np.random.randn(5)$ ⇒ 5 num of gaussian $[-,-,-,-,-]$ ⇝ here $a = a^T$ & $a \cdot a^T$ is just a no

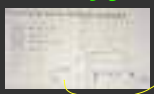
$a =$ ——— $(5,1)$ ⇒ $[[-,-,-,-,-]]$ ⇒ here $a \neq a^T$ here $a \cdot a^T$ is an numpy array

Tip: use assert func. of python to get notified when something is indeed wrong.

Module - 3

Neural network

Vectorizing across multiple examples



Activation functions

$$\tanh = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

tanh is almost always superior to sigmoid function coz its centered at 0 & mean comes near to 0 which makes learning faster

tanh cannot be used in output layer of binary classf^n coz we need output btw 0 to 1 there

Problem with sigmoid & tanh ⇒ when z is very small or large then gradient is close to zero which slows down the learning a lott

Relu $a = \max(0, z)$ Rectified linear unit

Leaky Relu ⇒ ..... $a = \max(\alpha z, z)$ $\alpha = 0.01$ or some small value.

Why do we need non-linear activation functions

→ If we activate all neurons with linear function then the model cannot learn anything more complex than a simple linear regression, hence we prefer non-linear functions as activation func.

→ Linear func. is used very rarely when we do compression (in hidden layers) more about this later.

→ If the model we build is of regression then we can use it only in the output layer to get the desired output.

Derivatives of activation function

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

$a = \tanh = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

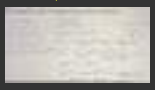
$$a' = \frac{(e^z + e^{-z})(e^z + e^{-z}) - (e^z - e^{-z})(e^z - e^{-z})}{(e^z + e^{-z})^2} \Rightarrow a' = 1 - a^2$$

Leak relu $= h(z)$

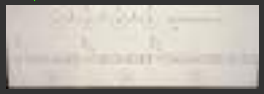
$$h'(z) = \begin{cases} \alpha & \text{if } z < 0 \\ 1 & \text{if } z > 0 \end{cases}$$

Gradient descent for neural networks

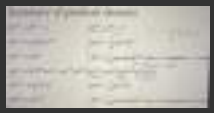
→ keepdims = True ≡ prevents python from giving output of shape (n,)



Derivation of above eqn



→ all these have to be computed for updation for this we need all below derivatives.



$$C = J(\hat{y}, y) = -\left(y \ln \hat{y} + (1-y)\ln(1-\hat{y})\right)$$

$$\frac{\partial C}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$$

$$\frac{\partial C}{\partial \hat{y}} = \frac{\hat{y} - y}{\hat{y}(1-\hat{y})}$$

$$\frac{\partial \hat{y}}{\partial z^3} = \sigma(z^3)(1-\sigma(z^3))$$

$$\frac{\partial z^3}{\partial a^2} = \theta_3 \quad \frac{\partial z^3}{\partial b_3} = 1 \quad \frac{\partial a^2}{\partial z^2} = g'(z^2)$$

$$\frac{\partial z^2}{\partial \theta_2} = a_1 \quad \frac{\partial z^2}{\partial b_2} = 1$$

$$\frac{\partial z^2}{\partial a^1} = \theta_2^T$$

$$\frac{\partial a_1}{\partial z_1} = g'(z_1)$$

$$\frac{\partial z_1}{\partial \theta_1} = x \quad \frac{\partial z_1}{\partial b_1} = 1$$

Random initialization

→ In logistic / linear regression we can initialize all weights to zero but we cannot do the same in deep coz gradient descent won't work. Why?

→ Reason:

Lets say there are 1 hidden layer with two neurons.

$x \rightarrow$ (O, O) → O → output

$$w^1 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad b^1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad w^2 = \begin{bmatrix} 0 & 0 \end{bmatrix} \quad b = [0]$$

Now since $w_{1,1}$ & $w_{2,2}$ of w^1 is same their updations are always gonna be same hence they won't be able to calculate anything complex, it just same as having only one neuron in hidden layer.

This is known as **Symmetry problem**

∴ we initialize it as $W^{[1]} = np.random.randn((2,2)) * \alpha$

α: generally very small coz if we are using sigmoid or tanh taking large values makes gradient very small hence makin learnin very small if output or hidden layers don't involve tanh or sigmoid than this is not much of an issue.

Module - 4

Deep L-Layer neural network

L - layer network

no of layer NN has $n^{[l]}$ → l^{th} hidden layer

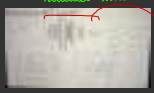
$a^{[l]} = g^{[l]}(z^{[l]})$; $n^{[l]} = k$ → no of neurons in l^{th} layer

↓ activation func. of l^{th} layer

$w^{[l]}$ = weights of l^{th} layer.

$b^{[l]}$ = bias " "

Figuring out dimensions of matrices

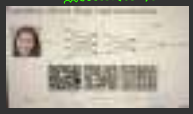


$z^1 = w^1 x + b^1$: $(3,1) = (3,2)(2,1) + (3,1)$

$z^2 = w^2 a^1 + b^2$: $(5,1) = (5,3)(3,1) + (5,1)$

Generally	Vectorised
$W^{[l]} : (n^{[l]}, n^{[l-1]})$	$Z^{[l]}, A^{[l]} : (n^{[l]}, m)$
$b^{[l]} : (n^{[l]}, 1)$	
$dW^{[l]} : (n^{[l]}, n^{[l-1]})$	
$db^{[l]} : (n^{[l]}, 1)$	
$X^{[l]} : (n^{[l-1]}, 1)$	

Why deep representations



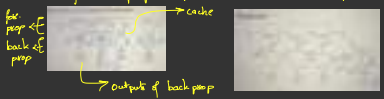
The further layers automatically compute & analyse complex stuff.

Ex  XOR of n nos using $\log_2 n$ neurons & fast & easy

" " " " " single layer ⇒ slow & complex & inefficient

Forward & backward prop

→ in forward prop itself we cache i.e store a few values so that they can be used in back prop, we cache z, w, b



Parameters vs Hyperparameters

w, b are parameters

Hyperparameters are decided by us, which ultimately control the parameters.

Ex α, # iteration, # hidden layers, # units, choice of activation func., regularisation, mini-batch size, momentum etc.

Course - 2 : Hyperparameter tuning

Module - 1

Train / Dev / Test sets

→ Previously the split would be 70/30 or 80/20 or 60/20/20, but now where there is abundance of data we can have fewer examples in dev/test like maybe 10,000 if there are say million training examples. Many times split is like 98/1/1 or 99.5/0.4/0.1 or 99.5/0.25/0.25 etc.

→ Dev & test set must come from same dataset not different ones for ex: all are pics clicked from smart phone.

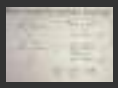
→ Normally when ppl say they just have train/test set then it means they actually have train/dev set but they call it train/test.

Bias / variance

→ This can become the perfect fit if the optimal error was 15%. So the validity of the model depends on Bayes error.

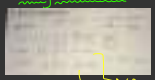
(ex Bayes Error)

Basic principle for machine learning



In NN we don't have to worry abt tradeoff i.e getting more data ↓ variance without affectin bias much & choosing bigger NN with proper regularisation ↓ bias almost without affectin variance.

Regularisation



L2 norm
L1

In neural net work

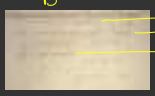
$$J(w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \underbrace{\frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|_F^2}_{\text{Frobenium norm}} \qquad \|w^{[l]}\|^2 = \sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}} (w_{i,j}^{[l]})^2$$

Why regularisation reduces overfitting?

→ As $w \downarrow$ $wz+b \downarrow$ then gradient value↑ in sigmoid or tanh ∴ almost linear ∴ reduces complexity hence reduces overfittin

Dropout regularisation

We copy NN then randomly remove few neurons & train it to test if overfitting is reduced.



→ probability that a neuron will be kept
→ matrix with random values
→ Lets say there are 50 units & 20% chance of dropping neurons so 10 neurons are dropped but weights of later layers are trained based on all neurons ∴ we enable us to not distend weights ∴ a3/= keep-prob
Train time becomes less & computation becomes less complex due to this
→ at each pass of gradient descent we zero out different set of neurons not the same one.
→ at test time we don't use dropout & keep all neurons & predict the out put.

Understanding dropout

→ The inputs get eliminated randomly hence the neuron cannot rely more on a particular feature ∴ weights↓ ∴ regularisation

→ keep prob is different for different layers, the layer having more neurons have lower keepprob & vice versa & in layers where we do not worry about overfitting keep-prob as 1. Input layer is generally not parameterised as if we'd it's done with high keep-prob ≥ 0.9

→ The downside of this is we have even more hyper parameters now.

→ One down side of dropout is cost func is not well defined anymore, so one of the probl solⁿ is to turn off dropout & ensure that learning curve is normal & then implement drop-out.

→ It's generally used in computer vision

→ The computation of cost func becomes harder as inputs are dropped at random.

Other regularisation methods

Data augmentation [refer prev course pg no 24]

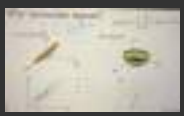
Early stopping

initially graph will underfit & on long training its gonna overfit hence its stopped somewhere at middle.

Here we follow tradeoff between bias & variance.

Normalizing inputs

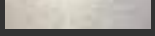
$x = \frac{x - \mu}{\sigma^2}$ because to normalize test & dev sets with μ_{train} & σ^2_{train}.



Why normalize inputs?

cost func
If Un-normalized inputs take a lot of iterations to converge & bounce around a lot too

Vanishing/exploding gradients



Lets take an example of NN with lotta neurons with linear activation func then we can show that

$z_1 = w^{[1]}x \quad z_2 = w^{[2]}a_1$

$a_1 = g(z_1) \quad a_2 = g(z_2) = w^{[2]}w^{[1]}x \quad \ldots\ldots$

$\therefore \hat{y} = w^{[L]}w^{[L-1]}w^{[L-2]}\ldots\ldots w^{[3]}w^{[2]}w^{[1]}x$

$\hat{y}$ → output layer

Let all else except output layer $\approx w = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$ → slightly more than 1.

then $\hat{y} = w^{[L]}\begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} x$

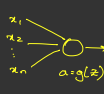
→ if L is large this is very huge & will explode

if its was $\begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$ now if L is large the gradient will vanish.

So its hard to train large NN due to this problem,
A partial solⁿ is how we initialize weights

Initialization of weights

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



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

- this sets the variance to $\frac{1}{n}$
- for tanh
- this is called Xavier initialization

We can also add a parameter that multiplies into the formula & it can be tuned too

for Relu we use $\sqrt{\frac{2}{n^{[l-1]}}}$

Sometimes we use $\sqrt{\frac{2}{n^{[l-1]} \cdot n^{[l]}}}$

Numerical approximation of gradients

if ε is very small values of ε

$$f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \quad O(\varepsilon^2)$$

for non-zero value of ε, error is order of

but

$$f'(\theta) = \frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}$$ error is of order $O(\varepsilon)$

Gradient checking

Take $W^{[1]}, b^{[1]}, \dots, W^{[l]}, b^{[l]}$ & reshape into big vector θ (concatenate)

Take $dW^{[1]}, db^{[1]}, \dots, dW^{[l]}, db^{[l]}$ & reshape into a big vector $d\theta$

Grad check

for each i:

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

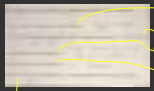
check whether $d\theta^{[i]} \approx d\theta^{[i]}_{approx}$

to check it ...

$$\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2} \approx \begin{cases} 10^{-7} \rightsquigarrow \text{great} \\ 10^{-5} \rightsquigarrow \text{ok} \rightsquigarrow \text{but look out} \\ 10^{-3} \rightsquigarrow \text{debug} \end{cases}$$

$\|v\|_2 \Rightarrow \sqrt{\|v\|^2}$ ← determinant

Implementation of grad check



- eg its very slow to calc grad check.
- look for which layer & when $d\theta$ is differing & try to debug
- i.e $d\theta$ includes regularization term too
- eg computing cost becomes difficult when randomly subset of neurons are dropped

so check and then implement dropout

Very rarely it so happens

that implementation of gradient descent is correct when w&b are close to zero, but as we run gradient descent w&b become bigger & it decreases inaccurately after that.

To check this is not the case run grad check at random initialization & then check after w&b wanders away from zero.

Mini-batch gradient descent

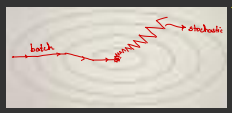
Lets say there are 5×10^6 training

make 5000 batches with 10^3 examples each....

$$X = \underbrace{\left[x^{(1)} x^{(2)} \ldots x^{(1000)} \right]}_{X^{\{1\}}_{(n_x,1000)}} \underbrace{\left| x^{(1001)} \ldots x^{(2000)} \right|}_{X^{\{2\}}_{(n_x,1000)}} \ldots \underbrace{\left| \ldots x^{(5\times10^6)} \right]}_{X^{\{5000\}}_{(n_x,1000)}}$$

$$Y = \underbrace{\left[y^{(1)} y^{(2)} \ldots y^{(1000)} \right]}_{Y^{\{1\}}_{(1,1000)}} \underbrace{\left| y^{(1001)} \ldots y^{(2000)} \right|}_{Y^{\{2\}}_{(1,1000)}} \ldots \underbrace{\left| \ldots y^{(m)} \right]}_{Y^{\{5000\}}_{(1,1000)}}$$

Mini batch t : $X^{\{t\}}, Y^{\{t\}}$



→ ∵ mini-batch updates after every mini batch there will always be some noise while converging unlike batch gradient descent which converges without any noise

→ A lot of time is wasted due to noise hence we need some technique to reduce this we need optimizers.

→ Explore more about Ada grad & ada delta

Mini-batch gradient descent

size = 5000

```
for t in range(t size):
    # forward prop on X^{t}
    Z^[1] = W^[1] X^{t} + b^[1]
    A^[1] = g^[1](Z^[1])
    Z^[2] = ....
    ⋮
    Z^[L] = ...
    A^[L] = g^[L](Z^[L])
    #cost
    J = 1/1000 Σ_{i=1}^{1000} L(ŷ^(i), y^(i)) + λ/1000 Σ_{l=1}^{L} ||w^[l]||_F^2
    # backprop
    w^[l] = w^[l] - α dw^[l]
    b^[l] = b^[l] - α db^[l]
```

Vectorized implementation

→ one epoch of training set

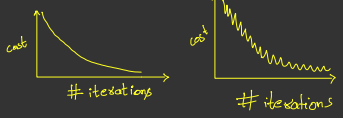
In batch gradient descent in a single epoch it goes through all training examples & per epoch gradient descent takes one step

but in mini batch " " " " " " " " examples of one mini batch & " "

" " " as many steps as size.

Understanding mini batch gradient descent

Batch gradient descent



Choosing mini batch size

→ Size of mini-batch = m : Batch gradient descent : $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$

→ " " " " = 1 : Stochastic " " : $(X^{\{1\}}, Y^{\{1\}}) = (x^{(1)}, y^{(1)})$ ....

In practise mini-batch size is somewhere in between 1 & m.

Stochastic

Looses benifit of vectorisation as we are iterating & computing & updation at each example

Batch

Too much time per iteration

Mini-batch

Vectorization ✓

fast ✓

Fastest learning in general ✓

Choosing mini batch size

Take account of available CPU & GPU

If data set is too small then ($m \leq 2000$) use batch.

Typical sizes are in powers of 2 => 64, 128, 256, 512

Of course we can create a hyper-parameter & pick out the best mini-batch size too.

Exponentially weighted moving average

$v_0 = k$ $\quad 0 < \beta < 1$

$v_1 = \beta v_0 + (1-\beta)\theta_1$

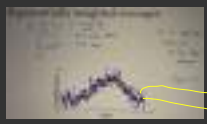
$v_2 = \beta v_1 + (1-\beta)\theta_2$

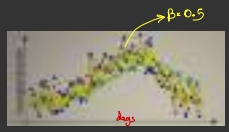
$\vdots$

$v_n = \beta v_{n-1} + (1-\beta)\theta_n$

this is almost same as avg of recent $\frac{1}{(1-\beta)}$ days

$\approx$ avg of recent $\frac{1}{\beta - 1}$ days





if β is small then more weightage is given to current value hence the graph will be noisy (yellow)

if β is large " graph or func gets updated slowly (green graph)

Using this we are gonna build hyperparameter which chooses best β.

Understanding exponentially weighted averages

$v_0 = \theta_0$ $\quad v_1 = \beta v_0 + (1-\beta)\theta_1$

$$v_2 = \beta\left[(1-\beta)\theta_1 + \beta\theta_0\right] + (1-\beta)\theta_2 = (1-\beta)\left[\theta_2 + \beta\theta_1 + \beta^2\theta_0\right]$$

$$v_3 = \beta v_2 + (1-\beta)\theta_3 = (1-\beta)\left[\theta_3 + \beta\theta_2 + \beta^2\theta_1 + \beta^3\theta_0\right]$$

$\vdots$

$$v_n = (1-\beta)\left[\theta_n + \beta\theta_{n-1} + \beta^2\theta_{n-2} + \beta^3\theta_{n-3} + \cdots + \beta^{n-1}\theta_1 + \beta^n\theta_0\right] \quad \text{—} \quad (1)$$

$\therefore$ As powers $\uparrow$ weightage to older datapoints is reduced

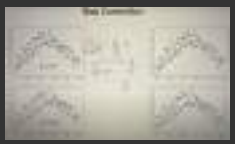
Generally $\beta = 0.9$ for NN

No of recent datapoints taken into consideration is given by $\frac{1}{1-\beta}$

As value of β increases graph changes more gradually hence less noisy graph.

" " " $\downarrow$ " " " rapidly " more " ".

Bias - correction



Look at graphs at left, there $v_0 = 0$ & hence at the start the average is not calculated properly, hence we need some correction

so instead of v_t we use equate RHS in (1) to $\frac{v_t}{1-\beta^t}$

$\rightarrow$ if t is less $1-\beta^t$ is small hence $\frac{v_t}{1-\beta^t}$ will be large

$\rightarrow$ " " " $\uparrow$ —"— " $\uparrow$ " " " " $\downarrow$

$\rightarrow$ Many a times people ignore initial phase & just check later phase, if you wanna correct we can implement this.

Eventually both the graphs combine (as shown in graphs on right) doing this just provides bias correction.

Gradient descent with momentum

$v_{dw} = \beta_1 v_{dw_{prev}} + (1-\beta_1)dw$

most common value for β_1 is 0.9

in general we don't apply bias correction coz after just 10 iterations our algo will be warmed up.

$10 \approx \frac{1}{1-\beta_1}$ for $\beta = 0.9$

Updation

$w = w - \alpha v_{dw}$

$b = b - \alpha v_{db}$

Initial $v_{dw} = 0$ & $v_{db} = 0$

matrix with same dimensions as dw

" " " " " db

many authors use just use

$v_{dw} = \beta_1 v_{dw} + dw$ $\rightarrow$ here v_{dw} ends up being scaled & on tuning β we need to tune α again.

$\rightarrow$ both expressions can be used just α will differ.

RMSprop

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

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

updation:

∴ dw is small ∴ we wanna ↑ it & db is large & we wanna ↓ it

S_{dw} is small & S_{db} is big ∴ on dividing with $\sqrt{S}$ w updation speed ↑ & b " " ↓

most common value for $\beta_2 = 0.999$

$$w = w - \alpha \frac{dw}{\sqrt{S_{dw}} + \varepsilon}$$

$\varepsilon = 10^{-8}$

$$b = b - \alpha \frac{db}{\sqrt{S_{db}} + \varepsilon}$$

→ ε is added coz in case $\sqrt{S} = 0$ then algo should not fail

Adam optimization algorithm : adaptive moment estimation algorithm

initialization: $v_{dw} = 0$, $v_{db} = 0$, $S_{dw} = 0$, $S_{db} = 0$

→ normally default values of $\beta_1, \beta_2, \varepsilon$ is used & α is tuned

→ Its a combination of momentum & RMS prop

→ Bias correction is performed on momentum & RMS prop. i.e $v = \frac{v}{1-\beta^t}$ $\quad S = \frac{S}{1-\beta^t}$

Updation

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

Learning rate decay

→ If we slowly reduce the learning rate this helps the algo., this is called learning rate decay

→ Intuition, initially we can take bigg ex steps but as we start to go towards minima lower learning rate is better as we can oscillate in a tighter region

high α

low α

How do we decrease α

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

(ii) $\alpha = (k)^{\text{epoch num}}\,\alpha_0$ ⇒ exponential decay

k is a number less than one

(iii) $\alpha = \frac{k}{\sqrt{\text{epoch num}}}\,\alpha_0$

(iv) $\alpha = \frac{k}{\sqrt{t}}\,\alpha_0$ where t is a hyperparameter

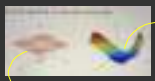
(v) 

i.e after some particular steps we change α

⇒ discrete stair case

(vi) Manual decay ⇒ if training time is too large then learning rate is observed at regular intervals and tweaked manually as and when required.

The problem of local optima



→ also a local optima, this is more common kind & is called a saddle point.

the gradient is zero here too

→ all dots here are local optima

→ There are very intuitions about lower dimensional space b/w as we go to higher dimensions the intuition decreases. More often in high dimensional space we stop at saddle point rather than global optima

→ Problem of plateaus

Plateau is a region where derivative is close to zero for a long time, then it stays on plateau for a very long time and then descends off of it.

Module - 3

Tuning process

Some of hyper parameters: α, β (of exponential avg), #hidden units, mini-batch size, #layers, learning rate decay, $\beta_1, \beta_2, \varepsilon$ (in adam)

α — most imp. generally this is returned

#hidden units, mini-batch size — 2nd priority

#layers, learning rate decay — 3rd priority

$\beta_1, \beta_2, \varepsilon$ — last priority

How to choose the values of hyper-parameters?

→ We can choose some no of random points in a n-dimensional space where n is the no of hyperparameters.

→ Coarse to fine search.

In the space say a particular point did well then we zoom into that region and take more samples from there and go on repeating this.

Using an appropriate scale to pick hyperparameters

→ Sampling uniformly doesn't make much sense, we have to sample across an appropriate scale.

→ Ex 1

Lets say we have to choose $n^{[l]}$ between 50 to 100. Here uniformly searching is fine.

Ex 2 - - - - - - α it can be from 0.0001 to 1. If we search like 0.1, 0.2, ... 1 its not right coz we never explored between 0.0001 & 0.1

To search this way in log scale we implement it this way:

Lets say search between a & b

then start = $\log_{10} a$ & end = $\log_{10} b$ $\quad r \in [start, end] \Rightarrow 10^r$

∴ we choose 0.0001 xxxxx 0.001 xxxxx 0.01 xxxxx 0.1 xxxxx 1

Some amount of random values from these ranges are taken

Ex 3 Lets say we have to choose β

β between 0.9 to 0.999

β = 0.9 xxx xxx 0.99 xx x x x x x 0.999

As β gets closer to 1 its sensitivity increases very highly say β = 0.9000 to 0.9005 → both roughly take avg over 10 examples

but

β = 0.999 to 0.9995 ∴ We need to sample very densely meaning to 1

↓ avg over 1000 examples ⋮ avg over 2000 examples

Hyperparameters tuning in practice: Pandas vs Caviar

→ Some times hyperparameters can get stale, at these times they need to be retested or re-evaluated after certain period of time

→ How people go about searching for hyperparameters

Babysit one model : Panda strategy

→ done when dataset is very huge or there are limited CPU or GPU resources, its monitored individually & hyperparameters are updated.

Training many models at a time : Caviar strategy

→ Many models with different sets of hyperparameters are chosen & trained parallelly & the best one is picked.

The type of approach we choose depends on computational resources & the amount of data we have.

Normalizing activations in a network

→ Batch normalization (created by two researchers Sergey Ioffe & Christian Szegedy), makes hyperparameter search problem very easier, makes NN more robust, helps to choose from large range & will also help to train deep networks more easily.

→ We generally normalize the input so that computation becomes easier, can we similarly normalize outputs of ten each layer?

→ Yes, now there is a choice to whether modify the finally output $a^{[l]}$ or $z^{[l]}$ which goes into activation func. i.e $g^{[l]}(z^{[l]}) = a^{[l]}$, there are many arguments on this but we generally normalize $z^{[l]}$

Implementation

Let's say we have z of a output layer $z^{(1)} \dots z^{(n)}$ of some layer l

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

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

$$z^{(i)}_{norm} = \frac{z^{(i)}-\mu}{\sqrt{\sigma^2+\varepsilon}} \qquad \varepsilon \approx 10^{-8}$$

When we do not want the mean as zero & variance as 1 then.....

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$$ γ & β are learnable parameters which are updated using some algo.

if $\gamma = \sqrt{\sigma^2+\varepsilon}$ & $\beta = \mu$ then $\tilde{z}^{(i)} = z^{(i)}_{norm}$

for other values of γ & β the mean & variance of normalised data changes.

Ex in sigmoid variance longer than 1 might be preferred as non-zero mean to take advantage of its non-linear nature

Note: β here & β in exponential avg. is different.

Fitting batch normalization into a neural network

$$X \xrightarrow{W^{[1]},b^{[1]}} z^{[1]} \xrightarrow{\beta^{[1]},\gamma^{[1]}} \tilde{z}^{[1]} \longrightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{W^{[2]},b^{[2]}} z^{[2]} \xrightarrow{\beta^{[2]},\gamma^{[2]}} \tilde{z}^{[2]} \longrightarrow a^{[2]} \longrightarrow \dots\dots$$

Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots, W^{[L]}, b^{[L]}$

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

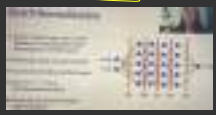
$\beta^{[l]} = \beta^{[l]} - \alpha\, d\beta^{[l]}$

or

update using weighted avg. or RMS prop or moment or Adam.

The above shown implementation is for batch gradient descent.

Benifits of Batch Norm



For mini batch gradient descent

$$X^{\{3\}} \xrightarrow{W^{[1]},b^{[1]}} z^{[1]} \xrightarrow[BN]{\beta^{[1]},\gamma^{[1]}} \tilde{z}^{[1]} \rightarrow g^{[1]}(\tilde{z}^{[1]}) = a^{[1]} \rightarrow \dots\dots$$

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

only looks at data of second mini-batch & normalizes, it doesn't look at z values of first mini batch again

Parameters: $W^{[1]}, b^{[1]}, \beta^{[1]}, \gamma^{[1]}$

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

Now we scale $z^{[l]}$ as $\tilde{z}^{[l]} := \frac{z^{[l]}-\mu}{\sqrt{\sigma^2+\varepsilon}}$ ∵ μ is subtracted if we add b while calculating z or not does not matter at all

∴ We can compute this as

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

$z_{norm} = \frac{z^{(i)}-\mu}{\sqrt{\sigma^2+\varepsilon}}$ ∴ we can just remove $b^{[l]}$

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

∴ Actual parameters are only $W^{[l]}, \gamma^{[l]}, \beta^{[l]}$

Dimension of $z^{[l]}, \beta^{[l]}, \gamma^{[l]} = (n^{[l]}, 1)$

Implementation

for t in range (1, size-mini-batch+1):

Compute forward prop on $X^{\{3\}}$

In each hidden layer use BN to make $z^{[l]}$ to $\tilde{z}^{[l]}$

Use back prop to compute $dW^{[l]}, d\beta^{[l]}, d\gamma^{[l]}$

Update as follows:

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

$\beta^{[l]} = \beta^{[l]} - \alpha d\beta^{[l]}$

$\gamma^{[l]} = \gamma^{[l]} - \alpha d\gamma^{[l]}$

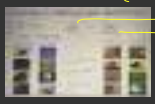
It can be updated through RMS prop, momentum or Adam also.

Why does Batch norm work?

Input distribution

→ If input distribution varies from lets say state 1 to state 2. A model trained for inputs of pattern state 1 then it will most probably not do so well on data from state 2 even if ground true function is same, if ground true function also changes then it's even worse. This moving from state to state is called covariate shift.

(like if img. is cat or not)

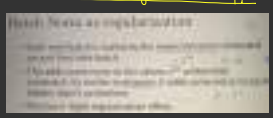


Why batch norm works?

→ The data from prev layers vary ie covariate shift occurs as model learns due to BN atleast mean & variance will be in a fixed range which helps the model to learn more quickly & efficiently. It limits how much the data changes due to prev layers. It ↑ stability & provides later layer of NN a firm ground to stand on.

→ The dependency of one layer on prev layer ↓ & layers are more free. It weakens the coupling between what early layer parameters has to do with later layer parameters

BN also has a regularisation effect



→ just like dropout which adds noise into layer by eliminating neurons at random this ensures later layers do not depend on previous layers much. In asimilar way BN adds additive noise since μ & σ^2 of minibatch is calculated so there is a slight noise introduced hence this brings about a slight regularisation effect.

→ As size of mini batch ↑ the noise introduced ↓ eg more examples are considered at a time hence the regularisation effect is ↓.

Normalization : collapse numbers between 0 & 1

Standardization: make mean 0 & variance 1.

Batch norm at test time

→ how to efficiently compute μ & σ^2 over entire minibatch. It is done using exponentially weighted average

We go through

$X^{\{1\}}, X^{\{2\}}, X^{\{3\}}, \dots$

$\mu^{\{1\}[l]}, \mu^{\{2\}[l]}, \mu^{\{3\}[l]}, \dots$

$\sigma^{2\,\{1\}[l]}, \sigma^{2\,\{2\}[l]}, \sigma^{2\,\{3\}[l]}, \dots$

& at test time we compute Z_{norm} using exponentially weighted avg of μ & σ^2

in frameworks they have more robust ways to calculate μ & σ^2 required for test time

Softmax Regression

C = # classes

output layer has C neurons → Σ output = 1

func :

$t = e^{z^{[l]}}$

$a^{[l]} = \frac{e^{z^{[l]}}}{\sum_{j=1}^{4} t_i}$

The decision boundaries are linear for small NN

" " " start to become more non-linear when depth of NN ↑

Training a softmax classifier

What is hardmax func. ? The largest number is assigned as one and all else are assigned as zero.

Loss func.

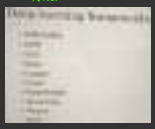
based on maximum likelihood estimation

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

$Y = [y^{(1)}, y^{(2)}, \dots, y^{(m)}]$ $\quad \hat{Y} = [\hat{y}^{(1)}, \dots, \hat{y}^{(m)}]$

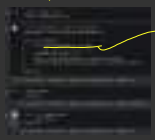
$y^{(i)} = \begin{bmatrix} y^{(i)}_{(1)} \\ y^{(i)}_{(2)} \end{bmatrix}$

Deep learning frameworks

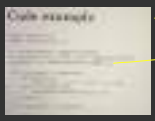


TensorFlow

→ If we implement forward prop in right way, it calculates back prop automatically.



→ just like a classic tf remembers what was done in forward prop then it plays it back to calculate derivatives



→ does same thing as left

→ it makes computation graph here to find all derivatives.

Structuring ML Projects

Module - 1

Suppose our model is done but we wanna further improve it. How do you choose what to do?

Orthogonalization

→ Its always better to have hyper parameters which perform different things when tweaked. It better if hyper parameters are independent & tweaking one shouldn't affect others.

→ When model doesn't fit train set ⇒ bigger network, different algo., diff optimizers etc.

- " " " dev " ⇒ Regularisation, bigger training set
- " " " test " ⇒ Bigger dev set
- " " perform well in real world ⇒ change dev set or cost func.

→ Early stopping is discouraged coz it tries to do 2 things at same time i.e handle fitting of train and dev set.

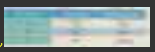
Single number evaluation metric

→ Suppose you have precision & recall now combine both to a single metric to evaluate the model. Evaluation is easier if there is only one number / variable to check.

Satisficing and optimizing metrics

→ Its not always easy to combine metrics into a single number, then we do this.

→ Say there are N metrics, then we choose one of them as optimizing and all others as satisficing.
↳ each of these will have a thresh hold only above which it will be accepted.

→ For example, 
Say we pick model which has run time below 100 ms with highest accuracy then,
accuracy is optimizing metric & runtime is satisficing metric.

Train / dev / test distributions

→ Dev & test sets must come from same superset of data.

→ The data given must be similar to what we expect in future.

Size of dev and test sets

→ 70-30 or 60-20-20 splits are ok for small size data but data is very large say a million or something then small amount like 1% or 1% for dev & test sets are also fine.

When to change dev/test sets and metrics?

→ Let's say there are two algos A & B; A has less errors & performing better on scores but due to various issues (like ethical) B is considered better, in these situations there is a need to change metric system or change dev/test set.

Why human level performance?

→ Bayes optimal error: the best possible error you can have on a function mapped from x to y.

→ Its normally close to human lvl performance Eg cat classification. Humans are pretty good at classifying cats. That's why after human level improving model is little hard.

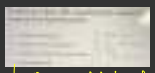
Avoidable bias	Ex1	Ex2
Human ≈ Bayes	1%	7.5% ← → the difference is called avoidable bias
Training error	8%	8% ↙
Dev error	10%	10%
	↳ Focus on reducing bias	↳ bias closer to bayes error ✓ focus on reducing variance

Understanding human level performance

→ Bayes error ← Avoidable bias
Training " ← difference
Dev " ← Variance

Bayes error ≈ human level performance

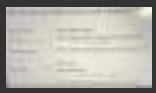
Surpassing human level performance



↳ these are all tasks where machine has access to huge databases

Humans are more better at Natural perception task such as speech recognition, computer vision etc though some models perform better than humans have too
∴ its harder for a machine to perform too well on a natural processing task.

Improving your model performance



Module - 2

Carrying out error analysis

→ Whenever we try to implement some possible sol^n to improve accuracy of model roughly pre-calculate by how much it could improve the model and then decide whether it is worth pursuing.

Cleaning up incorrectly labelled data

→ Depending on how much error is being caused by incorrectly labelled data we can remove or fix it.

→ Dev & test set must be from same distribution, train set can be from a different one

Build your first sys quickly and then iterate

→ Building a ok system then working on it and improving is better than building the perfect one at first go.

Training and testing on different distributions

→ what data we may encounter in real world similar to these must be at maximum in dev/test set.

Bias & variance with mismatched data distributions

→ Let'd say train error is 1% & dev error is 10% but train data & dev data is different, example train data has high res img but dev set has blurry img too

→ Now we do not know if variance is high or algo works well its just that dev set is hard.

→ train-dev error : a small portion train set is carved out as train-dev set then the model is not trained on this.

Bayes error : → avoidable bias
Train error : → variance
Train dev error : → data mismatch
dev error : → degree of overfitting
test error :
→ come from same distribution.

Addressing data mismatch

→ identify manually how data differs, try to synthesize data

→ There is a risk of overfitting as the small set of data that we are synthesizing Ex. All imgs may look like cars to us but machine may just about to recognize these as cars.

Transfer learning

We take a neural network trained for a similar task and alter it to suit the newer inputs. We alter the output layer(s) or add new layers as required & train them (fine tuning) if required train prev layers too (pre-training)

When is transfer learning used ⇒ → if both tasks have same input type

→ data for the task it was originally trained for was very abundant & for new task the data is less

→ Low level features features learnt in original model is useful for new model too

Multi-task learning

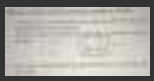
→ Ex computer vision model which detects multiple obj at same time, its gonna have 1 neuron for each obj, rather than having seperate neural networks for each object

→ even if data is missing labels it can be trained for whatever labels are available.

this is not softmax this is logistic regression for all neurons i.e each neuron has output between 0 & 1

here combining all outputs gives 1

→ When is multi-task learning good:



End to end deep-learning

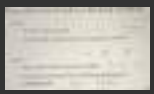
→ Ex speech recognition task it can recognize several features then phonemes then words then transcript it, now this requires us to hand design various things.

→ What if we want our machine to figure out all this on its own, then this is called end to end learning this can be done only when there is data in high amounts

→ Ex - face recognition first zooms in and crops the face then matches it with database

→ For higher complex tasks we need longer amount of data.

When to use end to end learning ?



→ Many a time a combination of hand design and end to end learning.

Convolutional Neural Network

Module

Computer vision

→ The data input here is very large say a 64x64 img then input is 64x64x3 (→ RGB) which is ok

→ Say a 1000 x1000 img then its 3 million inputs, the NN will have too many parameters and is really difficult to avoid overtraining.

→ Hence we use convolution operation.

Edge detection example for convolution operations

→ Say we a 6x6 img in grayscale ∴ input is 6x6x1.

→ We use a filter or kernel of say 3x3, then output will be 4x4 matrix

the elements are multiplied elem wise and summed up

orange box ⇒ 1x1 + 0x1 + 1x1 = 2

0x0 + 1x1 + 1x0 = 1 ⇒ ⇒ the filter is moved centrosally to convolve a matrix

0x1 + 0x0 + 1x1 = 1

→ Why do we convolve a matrix? ⇒ it helps identify features like edges etc.

Ex of a simple filter which detects vertical edge in gray scale image

Vertical edge filter has bright pixels on left & darker ones on right

More edge detection

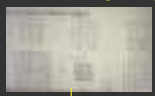
→ If img is reversed then values sign change, ofc this can be resolved by taking absolute value of the convolution matrix.

1	0	-1
2	0	-2
1	0	-1

Sobel filter

3	0	-3
10	0	-10
3	0	-3

Scharr filter



→ Most of the times all in filter are treated as parameters & are learnt, this may detect more complex edges at various angles too

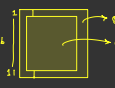
Padding

→ nxn img convolved with fxf filter gives n-f+1 x n-f+1 matrix

→ ∴ 1st downside we can only convolve a few times before img gets too small

→ 2nd downside : pixels at edges are used very less compared to pixels in middle, how a lot of matrices overlap. ∴ we are throwing away a lot of data at edges.

→ To solve both of these probs, we pad the img with say p pixels on all sides, then the dimension becomes n+2p-f+1 x n+2p-f+1



⇒ p = 1

There are normally two choices to pad: Valid & Same convolutions

- Valid: no padding
- Same: pad so that input size is same as output size i.e

$n = n + 2p - f + 1$

$p = \frac{f-1}{2}$ f is usually odd in computer vision

Strided convolutions

→ Normally we just move one step or one column but in strided we move s steps or s columns ahead

$$n \times n \quad * \quad f \times f \quad = \quad \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$$

padding : p

stride : s

Cross correlation vs convolution

What we are doing here is called cross-correlation in mathematical literature (& called convolution in ML literature)

Technically when convolving mathematically

the filter is flipped vertically & horizontally then operation is performed

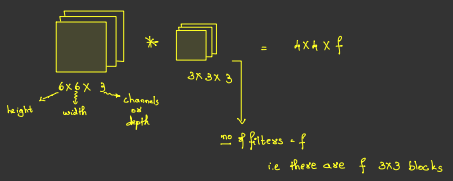
by doing this associativity property holds ie $(A * B) * C = A * (B * C)$

But we do not need this for ML hence we do not do this

Ex if filter is $\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$ $\xrightarrow[\text{mathematical}]{\text{in}}$ $\begin{matrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{matrix}$

Convolution over volumes

Suppose there are multiple channels like RGB etc then how do we represent?



how are these convolved?

The requirement is that the input & filters have same number of channels.

Row wise product is taken channel wise and summed up and written.

If there are f filters then the above operation is done f times and all output matrices are stacked.

One layer of a convolutional network

Say we have f number of $m \times n$ filters, how many trainable parameters are present?

$(m \times n + 1) f$

bias

Notation of convolutional layer l

$f^{[l]}$ = filter size $\quad s^{[l]}$ = stride

$p^{[l]}$ = padding $\quad n_c^{[l]}$ = no of filters

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

activations: $a^{[l]} \rightarrow n_H^{[l]} \times n_w^{[l]} \times n_c^{[l]}$

weights: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$

bias: $n_c^{[l]}$ ⇒ often taken as $1 \times 1 \times 1 \times n_c^{[l]}$

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

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

$$n_{H/w} = \left\lfloor \frac{n^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

If using batch/minibatch of size m then $A^{[l]} \rightarrow m \times n_H^{[l]} \times n_w^{[l]} \times n_c^{[l]}$

A simple convolution network example

Types of Layers in convolutional network:

- Convolution (Conv)
- Pooling (Pool)
- Fully connected (FC)

Pooling layers

This reduces the size of matrix by pooling various sub matrices. 2 types of pooling: Max pool – takes the max ele of sub matrix ⇒ most commonly used

Avg pool – " " avg of all ele in sub matrix ⇒ not often used

Say there is 4×4 matrix & is pooled with 2×2 filter

filter ⇒ $f = 2 \quad s = 1$

normal formula $\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$ holds

Hyperparameters while pooling

f: filter size

s: stride

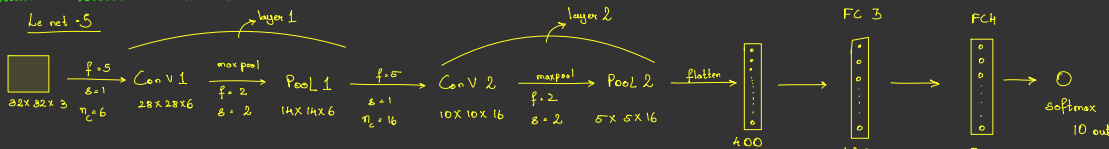
Max or avg pooling

no parameters to learn

input: $n_H \times n_w \times n_c$

output: $n_H' \times n_w' \times n_c$

Convolutional neural network example



A fully connected layer is just like normal ANN layer.

∵ pooling layers do not have parameters to learn conv layer and pooling layer together are referred as a layer sometimes

$w^{[3]} : (120, 400)$
$b^{[3]} = 120$

$w^{[4]} : (84, 120)$ (for Digit classⁿ)
$b^{[4]} = 84$

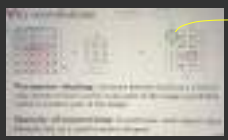
Why Convolutions?

Say we try to process by ANN then

for mapping $32 \times 32 \times 3 \xrightarrow[n_c = 6]{f = 5} 28 \times 28 \times 6$ ⇒ parameters : $5 \times 5 \times 3 \times 6 + 6 = 456$

3072 × 4704 (ANN) ⇒ parameters : 14,450,688 : too many parameters for a single layer which is processing a small image as 32 × 32.

Benefits of using convolutions



This is connected with only 1 sub matrix all others etc have no effect on this

This makes model less prone to overfit

Translation variance: even if we shift a few pixels a cat is a cat ∵ all lower level features are same this can be handled very efficiently

We build a CNN & use gradient descent to optimize parameters to reduce J.

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

Module - 2

Why look at case studies?

Due to large no of hyperparameters its difficult to build a model from scratch hence we use architectures similar to popular ones which have been built.

Ex LeNet-5, Alexnet, VGG,

ResNet, Inception

Classic networks

LeNet -5

⇒ At that time sigmoid/tanh were used and not Relu

⇒ Computation was slow hence filters looked at particular channels only

⇒ non-linearity is introduced after pooling layers

≈ 60k parameters

AlexNet

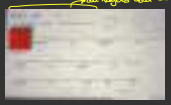
⇒ Relu is used

⇒ had Local response normalisation i.e it would normalise the filters output

≈ 60 M million parameters

In general as we proceed n_H, n_w ↓ & n_c ↑

VGG-16



≈ 138 M parameters

VGG 19 works almost same as VGG16 hence many use VGG16

Residual Networks (ResNets)

In this we stack residual blocks to form the network

Problem with going deep

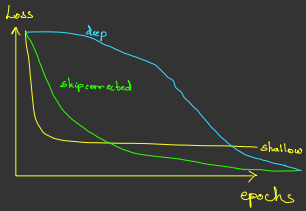
When we initialise the neural network all weights are randomly initialised

When data goes ahead by the time it reaches the last layer its reduced to random noise due to multiplications with random numbers (initially)

The loss which is computed at end travels through back prop to update gradients, now at later layers data was scrambled too much hence updating gradients there doesn't matter much.

Now the updates to initial layers also won't mean much because their gradients have been scrambled (i.e $w = w - \alpha \Delta$)

Hence deeper networks take long time to learn.



How to solve this?

→ We must find a way so that more meaningful data reaches later layers & make their inputs meaningful

→ " " " " " " less gradients to arrive at early " to " " updates "

We establish skip connections where data goes through & around blocks

We combine output & data at end giving it two paths to follow

How do we combine? ⇒ (1) we take both tensors & add them element wise or concatenate them

Advantages

→ each block augments the data. Now the work of each block is to figure out what it can add to input rather than figuring out what the input is.

→ Even if we are adding data instead of concatenating initially the weights are confined around zero so that initially we are passing data relatively unchanged.

→ ∴ Each block has simpler things to learn, has access to better info.

→ Shorter gradient paths — faster.

→ initially as the gradients are meaningless we can get good updates from shorter paths & later as the model learns more meaningful updates arrive from longer path. Hence less always ↓. Hence training becomes

→ Modularity ie its easy to add blocks.

Concerns

Shape don't match

if only ANN is involved ⇒ we have to match dimensions, so we need some way to reshape things.

if CNN is also involved ⇒ n_c must match, there are constraints on n_H & n_W. We cannot keep concatenating else the activation tensors will get too big & there will be an explosion of parameters (for ANN)

Hence if there are too many blocks we prefer addition over concatenation & there are usually skip connections which use addition continuously.

(Can refer to YTuid on secret of Professor Boyce)

Shape matching

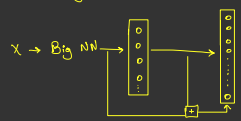
→ To simplify conv blocks we can use $s = 1$ & valid padding so that n_H & n_W is preserved

→ To adjust depth we can use 1×1 filter with required depth so that we can adjust the overall depth.

Why ResNets work?

Let's take an example

$X \rightarrow$ Big NN $\rightarrow a^{[l]}$



$a^{[l+2]}$

Now, $a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$

$= g(w^{[l+2]} a^{[l+1]} + b^{[l+2]} + w_s a^{[l]})$ *

Say $g(\cdot)$ is ReLu

Now even if the learning is insignificant hence the output will still be equal to $a^{[l]}$ itself hence not making model any worse.

Most often the problem is the learning becomes worse as we go deeper this is avoided here

W_s is vector multiplied to match shapes, it might be learnable or just a fixed matrix to pad

Network in Network and 1×1 convolutions

if we want final output to have depth d then use d no of filters

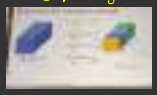
Why do we use this?

→ to ↑ or ↓ n_c

→ even if we decide to keep n_c same this still adds non-linearity hence will learn something.

Inception network motivation

→ Say after a layer we have to choose what we gotta do add a conv or a pool layer, inception network says do them all



→ instead of us having to choose we present all options to model & let it choose the meaningful ones among it.

→ Problem: computational cost

Say there is $28\times28\times192$ $\xrightarrow[\text{same}]{\text{conv } 5\times5,\ n_c=32}$ $28\times28\times32$

Convolve 5×5 with 28×28 ⇒ $28\times28\times5\times5$

there are 192 no of 28×28 & 32 no of 5×5

∴ total ⇒ $28\times28\times5\times5\times192\times32$

no of multiplications $28\times28\times5\times5\times192\times32 = 120$ million

Simple trick: multiple d times of filter & output

what if we use 1×1 conv?

$28\times28\times192$ $\xrightarrow[n_c=16,\ 1\times1\times192]{\text{conv}}$ $28\times28\times16$ $\xrightarrow[n_c=32]{\text{conv } 5\times5\times16}$ $28\times28\times32$

bottleneck layer

$28\times28\times1\times1\times192\times16 + 28\times28\times5\times5\times16\times32$

≈ 12.4 m

Inception network

→ Checks whether output can be predicted from this layer

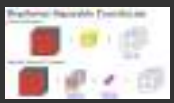
→ Validates how meaningful data is here

→ Acts as regularizer & prevents model from overfitting

MobileNet

→ This is computationally less heavy, hence we may run this on devices like mobile phone too.

→ Key idea: Normal vs depthwise separable convolutions



Normal conv ⇒ $6\times6\times3 \ * \ 3\times3\times3 = 4\times4\times n_c'$ (n_c' filters)

cost: $3\times3\times4\times4\times3\times n_c'$

i.e. cost = #filter parameters × #filter positions × #filters

$3\times3\times3 \times 4\times4 \times 5$

The cost reduced is generally: $\frac{1}{n_c'} + \frac{1}{f^2}$

Depthwise separable conv.

$n_h \times n_w \times n_c \ * \ f\times f = n\times n\times n_c$ (n_c filters)

Pointwise conv.

$n\times n\times n_c \ * \ 1\times1\times n_c = n\times n\times n_c'$

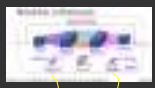
#filters = n_c'

Example: $6\times6\times3 \longrightarrow 4\times4\times5$

$6\times6\times3 \ * \ 3\times3 \xrightarrow[n_c=3]{3\times3} 4\times4\times3 \xrightarrow[n_c=5]{1\times1} 4\times4\times5$ ⇒ cost: $3\times3\times4\times4\times3 = 432$ + $1\times1\times3\times4\times4\times5 = 240$ ⇒ 672

Mobile net Architecture

Mobile net v1 ⇒ input → depth wise separable conv → output



Mobile net v2 ⇒ input → Exp^n → depthwise → projection → output

(bottleneck block)

projection is just pointwise conv. itself but it ↓ size hence the name. (channels)

Exp^n layer ↑ no of channels

Advantages of mobile net v2

∵ it has exp^n & depth wise the model learns richer functions there and then due to projection layer the memory taken is also reduced hence deployable on small devices.

Efficient Net

How to scale up or down a model based on device it's deployed on?

After resolution of img, depth of NN, or make layers wider.

We have to take care of trade off & implement this.

[Can have a look at paper mentioned in notes for further info]

Transfer Learning

Freeze a few layers & train others or depends .....

Data Augmentation

→ Rotating, Shearing, Local warping

→ Apply PCA on RGB channels and alter them

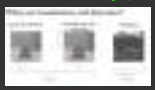
→ when data is too huge then data is taken from harddisk, distortions are applied & this is treated as batch and processed

State of Computer vision

when there is little data hand engineering is really important & tricks like transfer learning plays major role.

Module - 3

Object localization



Classⁿ with localization

Lets say we have to classify into

1 - pedestrian
2 - car
3 - motorcycle
4 - background

Output y =

- p_c → whether any object is present or not
- b_x, b_y → midpoint coordinates of bounding box
- b_h → what % of total height of image does bounding box occupy
- b_w → what % of total width of image does bounding box occupy
- c_1, c_2, c_3, c_4 → one hot encoding of which object is present

Landmark detection

This outputs whether the object is present or absent & various landmarks. Ex we needs locations of eyes, mouth, position of ears etc.

output y = $[p_c, l_{1x}, l_{1y}, l_{2x}, l_{2y}, \ldots, l_{nx}, l_{ny}]$ — n landmarks

Object detection

Sliding window detection

A sliding window crops parts of img & model tells if object is present in it ; then same is done for bigger size windows

Downside : computation cost

Convolutional implementation of sliding windows

Turning FC layer into conv layer

$\equiv$ $1 \times 1 \times n$

n units

Lets say our model took inputs of $14 \times 14 \times 3$ imgs but we have larger dimensions like $16 \times 16 \times 3$ or $28 \times 28 \times 3$

(ie it detects objects in this size)

Now in sliding window approach we would crop $14 \times 14 \times 3$ by stride = 1 & feed each of them to model hence making 4 computations for $16 \times 16 \times 3$ as 64 computations for $28 \times 28 \times 3$

This is not at all necessary as much of data is same hence what we do is feed the larger img directly to model , in case of $16 \times 16 \times 3$ it outputs $2 \times 2 \times 4$ matrix

Now this is same as sliding windows output upperleft window is 0,0 in matrix.

" right " " - 0,1 " "

bottom left " - 1,0 " "

" right " " - 1,1 " "

Same logic applied to $28 \times 28 \times 3$ gives us this :

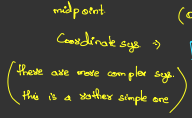
Bounding box predictions

Our goal is to output accurate bounding boxes

YOLO algorithm - You look Only once

We divide the image into grids say 3×3, the output will contain $\Rightarrow$

& compute it convolutionally, the object is assigned to the grid which has its midpoint



$$\begin{bmatrix} P_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

$c_1 \rightarrow$ car, $c_2 \rightarrow$ bike, $c_3 \rightarrow$ pedestrian

b_x & b_y will always be less than 1 coz its in box

So output will be $3 \times 3 \times 8$

img

b_h or b_w can be greater than 1 too if width or height of bounding boxes are greater than width or height of our grid

Again b_y & b_w are ratios of height & width compared to grid not points

↳ dimension of bounding box / " " grid cell

Intersection over union

Lets say there are two regions

IoU of this is $\Rightarrow$ $\frac{\text{Common region}}{\text{area of entire region}}$ $[0, 1]$

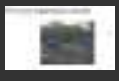
The higher the val. of IoU there is higher common region

Non-max suppression

Now, $\because$ grid size is small multiple grid cells may think they have the point hence there are multiple bounding boxes , how to deal with this?

All bounding boxes predictions are seen the one with highest one is picked & all other boxes which show IoU higher than 0.5 or 0.7 are eliminated

(highest P_c)



Anchor boxes

Say two objects then, how do we deal with it?

We draw two gt anchor boxes if object matches gt then it will be classified into it.

Say now we have n anchor boxes how will output look? (4+m classes)

$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} \rightarrow \text{anchor box 1}$$

$$\begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} \rightarrow \text{anchor box 2}$$

$$\begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} \rightarrow \text{anchor box n}$$

Limitations

We have only 2 anchor boxes but there are 3 obj in grid cell

2 objects share same anchor box shape

Advanced implementations may specify colors of objects in anchor boxes

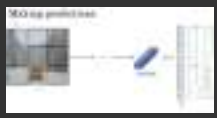
- include K-means to group together obj which have same shape & assign anchor box to it.

Putting it together YOLO algorithm

Let's say we are using $n \times n$ grid

then y is $n \times n \times$ #anchor $\times$ #1 unit box

here $p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3$



Now when we run it will look like this

→ Each grid cell will have atleast 2 prediction.

→ get rid of prediction with p_c

→ for each class run non-max suppression, to generate final predictions (separately, hence there will be atmost equal to no of classes)

final

Region Proposals [not used much]

The YOLO algo classifies many regions as it contains objects even though if it doesn't, to avoid this we run conv net only on certain regions. How do we determine these regions?

Segmentation algo.

This represents regions where object maybe present as blobs ⇒

Conv net is only run on these

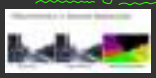
It was slower than YOLO hence changes were made

→ non-convolutional implementation

→ convolutional implementation

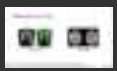
→ to predict regions instead of segmentation algo. convolutional layer was used

Semantic segmentation with U-Net



in obj detection we localize objects

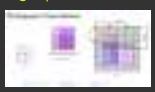
here we label all pixels



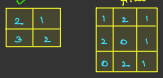
Our model will look like this where after a certain point the n_H & n_W ↑ again to form img. Here it classifies pixels to cat & not cat.

Transpose convolution

How do we increase the size of matrices again?

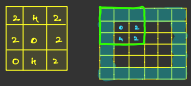


filter is placed on output

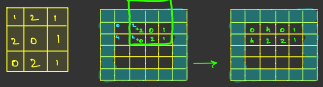


pick 1st ele. ie 2, multiply with all ele in filter & place it on output

where there is no padding write the values then



Then second ele. 1 ∴ stride is 2 shift filter accordingly



go on doing same to fill matrix

U-Net architecture

it utilizes skip connections from earlier layers. The present layer has high level contextual info & spatial info is taken from skip connection

ie it knows there is a cat in the region

takes exact region where cat is present from here

we are adding or concatenating

here no dimensions vary accordingly

Module - 4

What is face - recognition?

One shot learning

Learn from just one example to recognize person again

Similarity func.

d(img 1, img 2) = degree of diff btw img

If $d(img1, img2) \leq \tau$ ⇒ same person

$> \tau$ ⇒ diff.

Siamese network

An encoding is created out of an image through the model & d( ) is plotted

$$d(x^{(1)}, x^{(2)}) = \left| \left| f(x^{(1)}) - f(x^{(2)}) \right| \right|^2$$

↓ encoding

→ During training if A,P&N are chosen randomly then loss func is easily satisfied & model may not learn much.

→ While training model we need multiple img of a person but once it's fully trained to add new people to database one-shot learning can be used.

Triplet loss

The model looks at 3 imgs anchor, positive, negative

positive: same person; negative: different person

$$\mathcal{L}(A,P,N) = \max\left(\left| \left| f(A) - f(P) \right| \right|^2 - \left| \left| f(A) - f(N) \right| \right|^2 + \alpha, 0 \right)$$

$$\left| \left| f(A) - f(P) \right| \right|^2 \leq \left| \left| f(A) - f(N) \right| \right|^2 \quad \therefore \quad \left| \left| f(A) - f(P) \right| \right|^2 - \left| \left| f(A) - f(N) \right| \right|^2 + \alpha \leq 0$$

to avoid trivial solution

$$J = \sum_{i=1}^{M} \mathcal{L}(A^{(i)}, P^{(i)}, N^{(i)})$$

Face verification & binary classⁿ

Encode the input img. if encoding matches database output 1 else 0 ⇒ alternative to triplet loss

$$\hat{y} = \sigma\left(\sum_{k=1}^{128} \left| f(x^{(i)})_k - f(x^{(j)})_k \right| + b\right)$$

or

$$\hat{y} = \frac{\left(f(x^{(i)})_k - f(x^{(j)})_k\right)^2}{f(x^{(i)})_k + f(x^{(j)})_k} \Rightarrow \chi^2 \text{ similarity}$$

Database encoding can be precomputed & stored at a place to make things faster

Neural style transfer

Content - (c) Generative - (G)

Style - (s)

What are deep conv nets learning?

Initially it sees things like edges later on it starts recognizing complex shapes

Cost function

$$J(G) = \alpha J_{content}(C,G) + \beta J_{style}(S,G)$$

↓ how similar is content of original & generated img is?

? → how similar is style of img G & style of img S?

→ Initialize G randomly

→ Use gradient descent to minimize $J(G)$ ⇒ $G = G - \frac{\partial}{\partial G} J(G)$ ie we are updating pixel values

Content cost function

→ Say a hidden layer l is computing content cost, this layer cannot be at very early stage coz it will try to replicate actual img & can't be too late either else it will loose spatial info

→ Use pretrained conv net Ex VGG & compute $a^{[l](C)}$ & $a^{[l](G)}$, if they are similar then imgs are similar

→ $J_{content}(C,G) = \frac{1}{2} \left|\left| a^{[l](C)} - a^{[l](G)} \right|\right|^2$

Style cost func.

Say we are using layer l's activation to measure style. Style is correlation between activations across channels

If they have similar values then both of those features tend to occur together

Style matrix

$a^{[l]}_{ijk}$ = activation at (i,j,k). (i: height, j: width, k: channel)

$G^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$. $G^{[l]}_{kk'}$ measures how correlated are channels k & k' $\quad k,k' = 1,\dots,n_c^{[l]}$

$$G^{[l]}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_w^{[l]}} a^{[l]}_{ijk}\, a^{[l]}_{ijk'}$$

if they are correlated then values will be large else small (mathematically this is called unnormalized correlation). These matrices are called gram matrices

$G^{[l](S)}$ & $G^{[l](G)}$ (↓ of style img, ↓ of generative img.)

$$J^{[l]}_{style}(S,G) = \frac{1}{\left(2 n_H^{[l]} n_w^{[l]} n_c^{[l]}\right)^2} \sum_k \sum_{k'} \left(G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'}\right)^2$$

$$J_{style}(S,G) = \sum_l \lambda^{[l]} J^{[l]}_{style}(S,G)$$

(λ: hyperparameter)

1D & 3D Generalisation of models

14 x 1 * 5 x 1 = 10 x 1

Normally recurrent neural networks are used for 1-D data

3D

Data now has $n_H \times n_w \times n_d$ filter is also 3D now

14 x 14 x 14 x 1 * 5 x 5 x 5 x 1 ⟶ 10 x 10 x 10 x 16

($n_c = 16$)

Sequence Models

Module - 1

Why sequence models ?

Notation

Example: Name entity recognition

x: Harry Potter & Hermoine Granger invented a new spell.

$x^{<1>}$ $x^{<2>}$ $x^{<3>}$ . . . . . $x^{<9>}$

y: 1 1 0 1 1 0 0 0 0

$y^{<1>}$ $y^{<2>}$ . . . . . $y^{<9>}$

y = 1 if it part of name else 0

$X^{(i)}$ - i^{th} training example

$X^{(i)<t>}$ - t^{th} input in i^{th} training example

$T_x^{(i)}$ - length of i^{th} "

$y^{(i)}$ - i^{th} output

$y^{(i)<t>}$ - t^{th} ele. of i^{th} output

$T_y^{(i)}$ - length of i^{th} output

How do we represent words in sentence ?

→ We come up with vocabulary (dictionary) normally commercial applications have 50k - 100k words, bigger models may have it in millions

→ now we use one hot encoding to represent the words

→ ∴ each $x^{<t>}$ is a one hot encoded vector

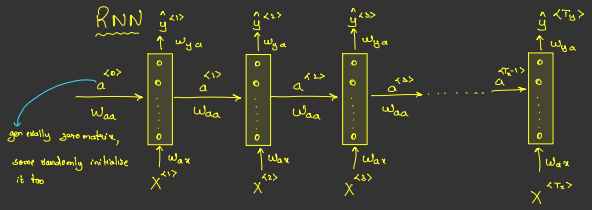
→ what if there is a new word which is encountered ? We create a new token called unknown word <UNK> .

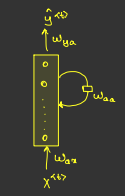
Recurrent Neural Network Model

why not a standard network ?

→ Inputs & outputs can be diff lengths in diff examples

→ Doesn't share features learned across diff positions of text





RNN is also represented this way

→ Here it recieves data from input & prev activation to predict output

→ RNN only recieves data from previously scanned words and not the words that come after it, Bi-directional RNN (BRNN) does it this way

→ Why is it important to recive data from words after ? Ex Teddy came to home. → here Teddy is name

I want teddy bear. → " " " not a "

Forward Propagation

W_{pq} ⇒ it computes p & it multiplied with q

$a^{\langle 0 \rangle} = \vec{0}$

$a^{\langle t \rangle} = g_1(W_{aa} a^{\langle 0 \rangle} + W_{ax} X^{\langle t \rangle} + b_a)$ ⇝ generally tanh sometimes Relu

$\hat{y}^{\langle t \rangle} = g_2(W_{ya} a^{\langle t \rangle} + b_y)$ ⇝ sigmoid or softmax

$a^{\langle t \rangle} = g(W_{aa} a^{\langle t-1 \rangle} + W_{ax} X^{\langle t \rangle} + b_a)$

$\hat{y}^{\langle t \rangle} = g(W_{ya} a^{\langle t \rangle} + b_y)$

Simplified notation →

$a^{\langle t \rangle} = g(W_a [a^{\langle t-1 \rangle}, X^{\langle t \rangle}] + b_a)$

$\hat{y}^{\langle t \rangle} = g(W_y a^{\langle t \rangle} + b_y)$

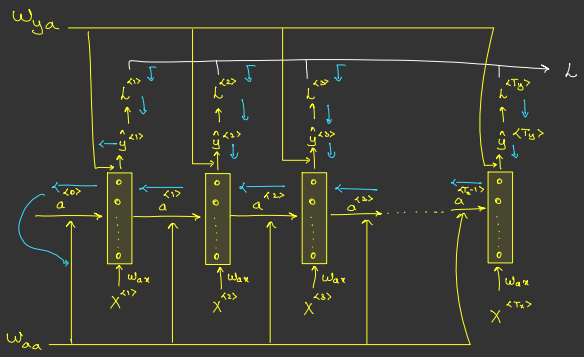
Simplified notation

$W_a = \begin{bmatrix} W_{aa} \vdots W_{ax} \end{bmatrix}$ (horizontally stacked)

$[a^{\langle t-1 \rangle}, X^{\langle t \rangle}] = \begin{bmatrix} a^{\langle t-1 \rangle} \\ X^{\langle t \rangle} \end{bmatrix}$ vertically stacked

∴ on matrix multiplication of these we get $W_{ax} X^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle}$

Back propagation through time



$L^{\langle t \rangle}(\hat{y}^{\langle t \rangle}, y^{\langle t \rangle}) = -y^{\langle t \rangle} \log \hat{y}^{\langle t \rangle} - (1 - y^{\langle t \rangle}) \log(1 - \hat{y}^{\langle t \rangle})$ → Binary cross entropy if sigmoid else sparse categorical cross entropy func.

$L(\hat{y}, y) = \sum_{t=1}^{T_y} L^{\langle t \rangle}(\hat{y}^{\langle t \rangle}, y^{\langle t \rangle})$

Different types of RNNs

Name entity recognition is many to many architecture. i.e many o/p & many i/p

Sentiment classification is many to one.

Music generation is one to many.

Machine translation is many to many. ⇝ has 2 parts an encoder & a decoder

Language model and sequence generation

Lets say there are 2 sentences : The apple & pair salad. - S_1

" " - pear " . - S_2

S_2 is correct but they are pronounced exactly the same, how does machine differentiate?

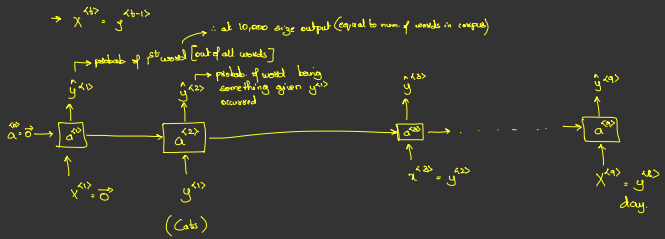
By Language model. ⇒ It outputs the probability of likeliness of that sentence

$P(y^{<1>}, y^{<2>}, \ldots, y^{<T_y>})$ is the output

To build such a model we need a large data set as a Corpus

→ First we tokenize the sentence [<EOS> ⇒ end of sentence token]

→ Next we build the model



Cats average 15 hrs of sleep a day. <EOS>

$y^{<1>} \ldots \ldots \ldots \ldots \ldots \ldots y^{<9>}$

∴ RNN learns to predict one word at a time.

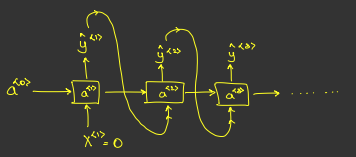
$$\mathcal{L}(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log \hat{y}_i^{<t>}$$

$$P(y^{<1>}, y^{<2>}, y^{<3>}, \ldots, y^{<m>}) = P(y^{<1>})\, P(y^{<2>} \mid y^{<1>}) \cdots\cdots P(y^{<m>} \mid y^{<1>}, y^{<2>}, \ldots, y^{<m-1>})$$

Sampling Novel Sequences

After we train a model, to get rough idea of what it has learnt we can sample a sequence

Sampling ⇒



→ the output from a particular unit is chosen at random from np.random then fed to next unit & its made to predict.

→ When to stop? If <EOS> exists, then until its generated else, until some fixed no of words

→ If we do not want <UNK> to be a part of o/p then we can code it so.

Character level language model

Instead of words it uses letters as i/p & o/p. It's rarely used & is more complex than normal models.

Vanishing gradients with RNNs

Recap: in back prop we use chain rule multiply many derivatives, i.e many small values hence as we go back further the gradients keep getting smaller.

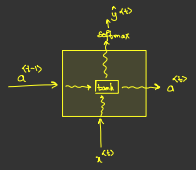
In sentences a word which came long ago influences word which came long time after it. Here, we need to find a way to overcome vanishing gradient problem.

Exploding gradient will just give NaN values & a efficient way to clip gradients.

i.e if gradients go above a certain threshold then we normalize it in some way.

Gated Recurrent units (GRU)

RNN unit >



$$a^{<t>} = g\left(w_a\left[a^{<t-1>}, x^{<t>}\right] + b_a\right)$$

The cats, which already ate ....., was full

Simplified

GRU unit has a new variable called 'c' which stands for memory cell.

$$c^{<t>} = a^{<t>}$$

At every time step we overwrite $c^{<t>}$ as $\tilde{c}^{<t>} = g\left(w_c\left[c^{<t-1>}, x^{<t>}\right] + b_c\right)$

then we have a gate (Γ) & has value b/w 0 & 1

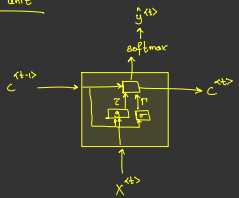
$$\Gamma_u = \sigma\left(w_u\left[c^{<t>}, x^{<t>}\right] + b_u\right)$$

u stands for update gate

gate decides by how much we are going to update the cell value

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \left(1 - \Gamma_u * c^{<t-1>}\right)$$

GRU unit



Γ_u generally is 0 or 1 i.e there is high or very very low update

∴ if $c^{<t>}$ is not updated it almost remains constant, hence addressing vanishing gradient problem.

$c^{<t>}, \tilde{c}^{<t>}, \Gamma_u$ all have same dimensions

Actual

→ notations also used in literature

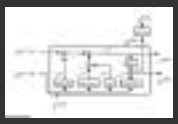
$\tilde{c}^{<t>} = g\left(w_c\left[\Gamma_\gamma * c^{<t-1>}, x^{<t>}\right] + b_c\right)$

γ $\Gamma_\gamma = \sigma\left(w_\gamma\left[c^{<t-1>}, x^{<t>}\right] + b_\gamma\right)$ ⟶ relevance gate i.e tells how relevant is $c^{<t-1>}$ to update $c^{<t>}$

u $\Gamma_u = \sigma\left(w_u\left[c^{<t-1>}, x^{<t>}\right] + b_u\right)$

h $c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u)\, c^{<t-1>}$

Long short term memory (LSTM) unit



Variations ⇒ peephole connection

$$\Gamma_i = \sigma\left(W_i\left[a^{<t-1>}, x^{<t>}, c^{<t-1>}\right] + b_i\right)$$

$i = f, u, o$

We can just take LSTM units & join many of em of make a model.

Bi directional RNN

→ Lets us take info from after & before

→ Info from start goes to next & from the end it comes to start too.

→ Disadv. ⇒ it needs to have complete sequence of data available to start processing, most applications do have complete sequence available beforehand itself except things like speech to text where entire sequence may not be present, these use more complex versions of BRNN.

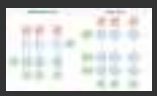
→ Normally the go to choice for NLP is LSTM with BRNN

Deep RNNs

→ Normally too many RNN layers aren't stacked on top of each other coz they are computationally very expensive as it temporal dimension is high.

→ A few RNN layers followed by normal ANN type of networks are pretty common.

→ Here instead of RNN, GRU or LSTM with RNN or BRNN can also be used.



Module-2

Word representation

→ Make model understand analogies like man : woman :: king : queen

→ Previously we represented words as one-hot-encoded vectors. Is there any disadv to it? Here we cannot establish any relationship between words.

Ex. king & queen are more related than king and apple. Ex. I will have orange ——. juice seems to fit both & if some how machine knew apple & orange related then it
" - " apple —— could learn this more easily.

So, how do we achieve this?

We build a feature embedding,

→ Features in which they are actually embedded won't be this conspicuous this is just an analogy.

→ if we project all these diffed embedded vectors to a 2D plane then we can see how similar words are grouped together.

→ This projection is done through t-SNE algorithm.

↳ we take features like these & assign values to words

→ represented as e_{5391} → 300 dimension vector

↳ its position in corpus

Using word embeddings

→ in net works previously we fed one-hot encoded data now instead we feed embedded data.

→ similar to encoding an img in siamese net work.

embedding

Properties of word embeddings

→ img mapping like this will hold true in original graph which had n dimensions, it may not hold true in 2D img given by T-SNE algo.

→ help in analogy reasoning

→ man : woman :: king : ?

$e_{man} - e_{woman} \approx e_{king} - e_?$

so we find it by $e_? \approx e_{king} - e_{man} + e_{woman}$

↳ found from corpus

similarity func. : $sim(e_w, e_{king} - e_{man} + e_{woman})$

generally we use cosine similarity $sim(u, v) = \frac{u^T \cdot v}{\|u\|_2 \|v\|_2}$

(dot product) → (magnitude of u & v)

gives cosine of angle between vectors.

Embedding matrix

What we are actually learning here is a embedding matrix.

Ex

10k

a aaron apple ..... zulu <UNK>

300 features

$$\begin{bmatrix} \vdots & \vdots & \vdots & & \vdots \\ \vdots & \vdots & \vdots & & \vdots \\ \vdots & \vdots & \vdots & & \vdots \end{bmatrix} = E$$

$O_i = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \\ 0 \end{bmatrix}$ → one hot encoded vector of word i

$O_i * E = e_i$

in practise we use specialised func. to look up an embedding.

Learning word embeddings

→ Input can be all words before
→ or a few words before
→ or a few words before & after
→ or a nearby 1 word all of these work well

[These are context target pairs]

Word 2 vec

Skip-grams

→ Come up with few context-target pairs to create supervised word problem

→ choose a random word as context & choose a random word as target & try to predict target here we are just trying to learn word embeddings & not make our model good in predicting. (maybe in a small window)

Context 'c' & target 't'

$O_c \to E \to e_c \longrightarrow \underset{\text{softmax}}{\bigcirc} \to \hat{y}$

$$p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10000} e^{\theta_j^T e_c}}$$

θ_t : parameter associated with output t

$$L(\hat{y}, y) = -\sum_{i=1}^{10000} y_i \log \hat{y}_i$$

Problem + computation : $\sum_{j=1}^{10k} e^{\theta_j^T e_c}$ is very heavy especially if there are many words

Sol^n : using a hierarchical softmax classifier

Classifies to branches which has that word
∴ time is $\log n$ instead of n

in practice hierarchical clasf^n classifies common words at top & uncommon ones deep down the tree

We can choose 'c' randomly but then we will end up mapping words which occur too frequently which is not good. There are many ways in which words are chosen carefully and mapped

Negative Sampling

Pick a context-target from sentence & label it as 1 (that's our true example & pick other words at random from corpus & assign zero to them ; say we do this to generate k negative examples.

if we have very large data then choose small k, maybe 2–5

" " " small " " " large k " " 5–20

then we supervised model to learn to map from c to t

$$P(y=1|c,t) = \sigma\left(\theta_t^T e_c\right)$$

now we train only these k examples instead of all 10k examples in one-iteration

→ how do we choose −ve examples?

Sample according to empirical freq of words i.e o/c to how often words appear, but this ends up taking words like a, an, the etc

or randomly sample using $\frac{1}{\text{vocab size}}$ which is other end of extreme

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{10k} f(w_j)^{3/4}}$$

this is how the words are chosen

GloVe Word Vectors

→ Global vectors for word representation

X_{ij} = no. of time j appears in context of i

(j ↑ t, i ↑ c)

Model

$$\text{minimize} \sum_{i=1}^{10k} \sum_{j=1}^{10k} f(X_{ij}) \left(\theta_i^T e_j + b_i + b_j' - \log X_{ij}\right)^2$$

$f(X_{ij})$ is weighting term $f(X_{ij}) = 0$ if $X_{ij} = 0$

→ for words like this, the make sure its not too high

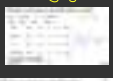
→ & for - - durian (appear rarely) it not too low

Roles of θ & e here are symmetric $e_w^{\text{final}} = \frac{e_w + \theta_w}{2}$

we can't guarantee that the individual embeddings are interpretable

Sentiment classification

Recognize words & avg there e_i vectors & run soft max on it to predict emotion



Problem this will classify last comment

also as good review coz word good appears so many times

a good alternative

is to use RNN for this.

Debiasing word embeddings

Gender, ethnicity bias which is learnt must be debiased

How do we do this?

→ Say there is gender bias, we select few words in which bias is right like e_{he}, e_{she}, e_{boy}, e_{girl} etc avg them & find $e_{male} - e_{female}$ ⇒ this is bias dir^n

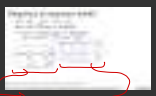
→ All other dir^n are non bias dir^n; all the words which need to be debiased are projected to non-bias dir^n ⇒ this is called Neutralization.

→ Equalize pairs ie pairs like e_{boy} & e_{girl} put them at equal dist from non-bias dir^n

Module - 3

Basic Models

→ Say we wanna build a machine translation model, how do we build it?



encoder n/w

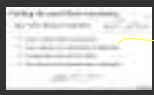
decoder n/w which o/p 1 word at a time

→ Say we have an img & we have to generate a caption or comment on that

→ The catch in these both is we want the most likely answer, not the random answer.

Picking the most likely sentence

→ if we have to pick most why not pick it greedily? it doesn't work out well coz more often used words will be labelled most likely even if its the best translation.



going is a word which appears often ∴ this will be picked as best if greedily picked

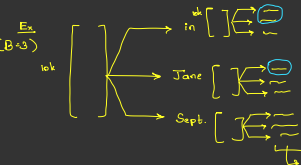
its called as conditional language mode coz second half is just a lang. model but instead of feeding zeros or random vectors we feed it a vector which captures input sentence.

∴ we need some algorithm.

Beam search

→ This has a hyperparameter called beam width (B), it always picks top B words & evaluates next word for each possibility & next out of all these again picks top B.

Ex. (B=3)



→ say ones circled in blue are picked, then it eliminated sept. as starting word.

$$P(y^{<1>}, y^{<2>} \mid x) = P(y^{<1>} \mid x)\, P(y^{<2>} \mid x, y^{<1>})$$

it evaluates 30k possibilities in one go

&

B copies of model will be made to evaluate them.

Refinements to Beam search

→ our func was $\arg\max \prod_{t=1}^{T_y} P(y^{<t>} \mid x, y^{<1>}, \ldots, y^{<t-1>})$, now all these are less than one, if we have a long sentence then computer will loose track of decimal values & suffer from rounding off problem. How do we overcome this?

→ instead of keeping track of product we take log & keep track of products ⇒ $\arg\max \sum_{t=1}^{T_y} \log P(y^{<t>} \mid x, y^{<1>}, \ldots, y^{<t-1>})$

→ Finally we also normalize it by multiplying $\frac{1}{T_y^{\alpha}}$ where α is hyperparameter defined by us, if $\alpha = 0$ ⇒ no normalisation ⇒ we choose a balanced value in between 0 & 1 like 0.7 or something

$\alpha = 1$ ⇒ complete " "

$$\arg\max \left(\frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} \log P(y^{<t>} \mid x, y^{<1>}, \ldots, y^{<t-1>}) \right)$$

if we choose too large B ⇒ better results, slower

" " " - small B ⇒ worse " , fast

Error analysis on Beam search

→ What if beam search is making a mistake, or is it our model that's making the mistake. How do we identify?

→ Lets say there is a sentence which is translated by human (y^*)

→ evaluate what % of errors is due to Beam search & what % due to RNN

if prob. is in Beam search we can ↑ Beam width or alter normalisation

" " " " RNN we have to evaluate our architecture & apply various ideas in course 3

Bleu score

Bilingual evaluation understudy

→ it gives high score if the output is close to human translation. This acts as an alternative to having a human evaluate the output

→ Say we have 2 references & both are correct, we have to score our output.

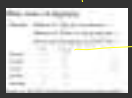
→ MT output : machine translation o/p

how many words match words in either of ref sentences.

So, precision of our o/p is $^{7}/_{7}$, this is a bad scoring egs, hence we clip the word count to max no of occurrences in both sentence

the word 'the' occurs twice in first sentence ∴ we clip it to 2

we call this modified precision.



→ clipped counts, now its demonstrated considering 1 word, we can consider n-words too

(called unigram) (n-gram)

→ How do we score? First we compute clip counts

$$P_n = \frac{\sum_{n\text{gram} \in \hat{y}} count_{clip}(n\text{gram})}{\sum_{n\text{gram} \in \hat{y}} count(n\text{gram})}$$

$$Score = BP\, e^{P_n}$$

brevity penalty

$$BP = \begin{cases} 1 & \text{if MT o/p len} > \text{ref o/p len} \\ e^{1 - \frac{\text{MT o/p len}}{\text{ref o/p len}}} & \text{otherwise} \end{cases}$$

why is BP added?

eg if sentences are too short they will end up getting high scores ∴ to tackle this we use BP

Attention model intuition

→ Say there is a really long sentence, then its hard to predict the sentence in such cases the machine gives bad o/p, how to modify this?

→ How is the machine translating at present? It looks at a sentence, memorizes it entirely, then translates. Now, how does human do it? We look at sentences in parts then translate it

→ So we will make machine do the same too.

→ attention model

→ normal mode

A separate RNN computes the o/p & it receives weights from french words too, which specifies how important is that word to generate that specific output.

$\alpha^{\langle t,t' \rangle}$ ⇒ how important is t' to generate t.

→ BRNN

Attention model

→ here attention weights are computed using a small NN

→ BRNN

[notations]

Speech recognition

Air pressure vs time graph → audio data

Take back o/p?

→ A spectro gram of how intense sound is with frequency is prepared as pre-processing step

→ How do we go about building speech to text sys?

We can do this by attention model itself, where it looks at a certain time frame & outputs the characters / words.

Cost func. → CTC (connectionist temporal classf°)

Say audio input was of freq 100 hertz i.e it 100 i/p per second, the characters are generated continously & a special character is introduced '_' & called blank

Eg "The quick ...."

ttt t_h_eee __ ␣ __qqq (␣ : space)


Collapse all char not separated by '_' into one : this becomes the q

Triggex word detection

→ Suppose triggex word is spoken here, it sets weights to 1 for a while coz that the model begins to hear.

Module-4

Transformers intuition

RNN processes the data sequentially. Transformers processes data parallely.

→ Attention + CNN made

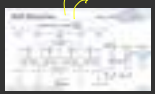
→ Self attention → say we have five words in sentence it computes 5 representations for it

→ Multi head → for loop over self attention process

Self Attention

A(q, K, V) = attention - based vector for representation of a word.

$$A(q,K,V) = \sum_{j} \frac{e^{\langle q, k^{<j>} \rangle}}{\sum_{j} e^{\langle q, k^{<j>} \rangle}} v^{<j>}$$



also called scaled dot product representation
denominator is scaled to prevent it from exploding

q_i = query, its like asking what's happening there?

k = key

v = value

q/k → its like answering the question

Now, embedding is done considering surrounding words.

Value tells it how other word should be represented due to this

W^q, W^k, W^v are weights

$q^{<i>} = W^q x^{<i>}$

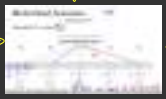
$k^{<i>} = W^k x^{<i>}$

$v = W^v x^{<i>}$

Multi-head attention

Each time we calculate self attention is called head

Second question : i.e Self attention for 2nd time



head = 3

for first time weights are

W^q, W^k, W^v & we compute it this way

Say highest softmax for W is highlighted with blue arrow.

$$\text{MultiHead}(Q,k,v) = \text{concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h) \cdot W_0$$

$$\text{head}_i = \text{Attention}(W_i^q Q, W_i^k, W_i^v)$$

all final answers of h no of heads are stacked together & given as o/p

Though it appears like we are iterating to calculate these we can calculate all these parallely.

Transformer network


start of sentence → i/p fed & its o/p fed n-times → initially only <sos> is given, then o/p n-times next word is generated

then <sos> next word both are given & its o/p n-times to predict next & so on....

Details

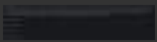
Positional encoding

$x^{\langle i\rangle}$ to $p^{\langle i\rangle}$
→ has same dimensions.

pos refers to position word 1 based indexing

i " " different dimensions of encoding

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right) \qquad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$



say $x^{\langle i\rangle}$ = [4 boxes] → d = 4

$p^{\langle i\rangle}$: i=0 k=0; i=0 k=1; i=1 k=2; i=1 k=3

$p^{\langle i\rangle}$ & $x^{\langle i\rangle}$ are added & they contain contextual semantic embedding & positional encoding info.

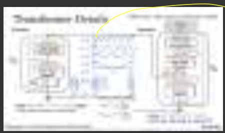
→ Along with this we use residual network too to pass on positional info

→ Transformer also uses a layer very similar to batch norm, its purpose is to pass along positional info

→ " " " " " called adda norm, similar to batch norm

→ Mask multihead attention, this is used mainly while training, here it has access to correct translation too. What it does is it hides a word & pretends everything before it predicted correct - ly & asks model to predict the hidden word.

→ Bert, Bert Distill are adv versions of this model.



→ along those positions which are dotted, the values are read for $p^{\langle i\rangle}$