Course 2

Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization

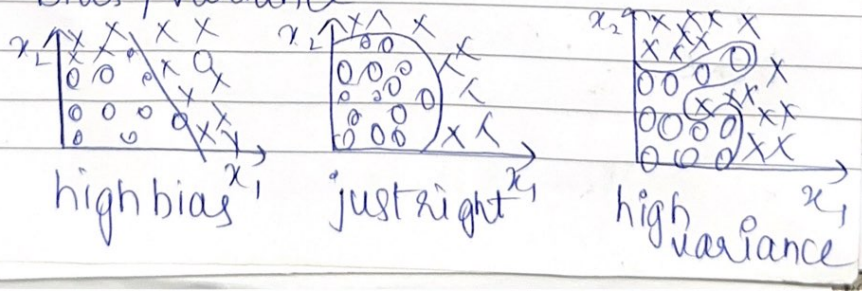
WEEK 1

→ Train / Dev / Test sets

Data divided → ① Training set
② Development set
③ Test set

- Above sets are used to evaluate the performance of machine learning algorithm
- Traditional ratios for Train / Dev / Test sets are 98% / 1% / 1% where there is a million examples.

→ Bias / Variance



→ Regularization

→ Logistic Regression

$$\min_{w,b} J(w,b) \qquad w \in \mathbb{R}^{n_x},\ b \in \mathbb{R}$$

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

	Error		
Training set	1%	15%	15%
Dev set	11%	16%	30%
	(high variance)	(high bias)	(high bias and high variance)

→ Basic Recipe for Machine learning

→ Regularization

→ Logistic Regression

$$\min_{w,b} J(w,b) \qquad w \in \mathbb{R}^{n_x},\ b \in \mathbb{R}$$

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

$$+ \underbrace{\frac{\lambda}{2m} b^2}_{\text{↳ this term is omitted}}$$

→ L_2 Regularization

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

→ L_1 Regularization

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

λ → Regularization Parameter

→ Basic Recipe for Machine Learning

	Error	(High variance) High bias	High bias and high variance
Training set	1%	15%	15%
Dev set	11%	16%	30%

→ How regularization reduces Overfitting

• Regularization helps with overfitting by reducing the variance of the model.
• Regularization is done by adding an extra term to the cost function that penalizes the weight matrices from being too large
• Intuition for why regularization helps is that if the regularization parameter is large, the weights will be relatively small, and the activation function will be relatively linear.
• Fix variable in a simple network to avoid overfitting

→ Neural Network

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

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

"Frobenius Norm"

w: → matrix of dimension $(n^{[l]} n^{[l-1]})$

$$\cdot dw^{[l]} = (\text{from backprop}) + \frac{1}{m} w^{[l]}$$

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

"weight decay"

$$w^{[l]} = w^{[l]} - \alpha \left[(\text{from backprop} + \frac{1}{m} w^{[l]} \right]$$

$$= w^{[l]} - \alpha \frac{1}{m} w^{[l]} - \alpha (\text{from back-prop})$$

→ Why Regularization Reduces Overfitting

- Regularization helps with overfitting by reducing the variance of the model
- Regularizatn is done by adding an extra term to the cost functn that penalizes the weight matrices from being too large
- Intuition for why regularization helps is that if the regularization parameter is large, the weights will be relatively small, and the activation function will be relatively linear
- This results in a simpler network that is less prone to overfitting
- When implementing regularization, make sure to plot the new definition of the cost functn that includes the regularization term.

→ Dropout Regularization

- It works by randomly eliminating nodes in the network and all of the

outgoing connections from those nodes
- This results in a much smaller network, which is then trained using backpropagation
- Implementation of Dropout
  "Inverted Dropout"

```
l = 3, keep-prob = 0.8
d3 = np.random.rand(a3.shape[0],
     a3.shape[1]) < keep-prob
a3 = np.multiply(a3, d3)
a3 /= keep-prob
```

→ these steps ensure that the expected value of the activations remains the same

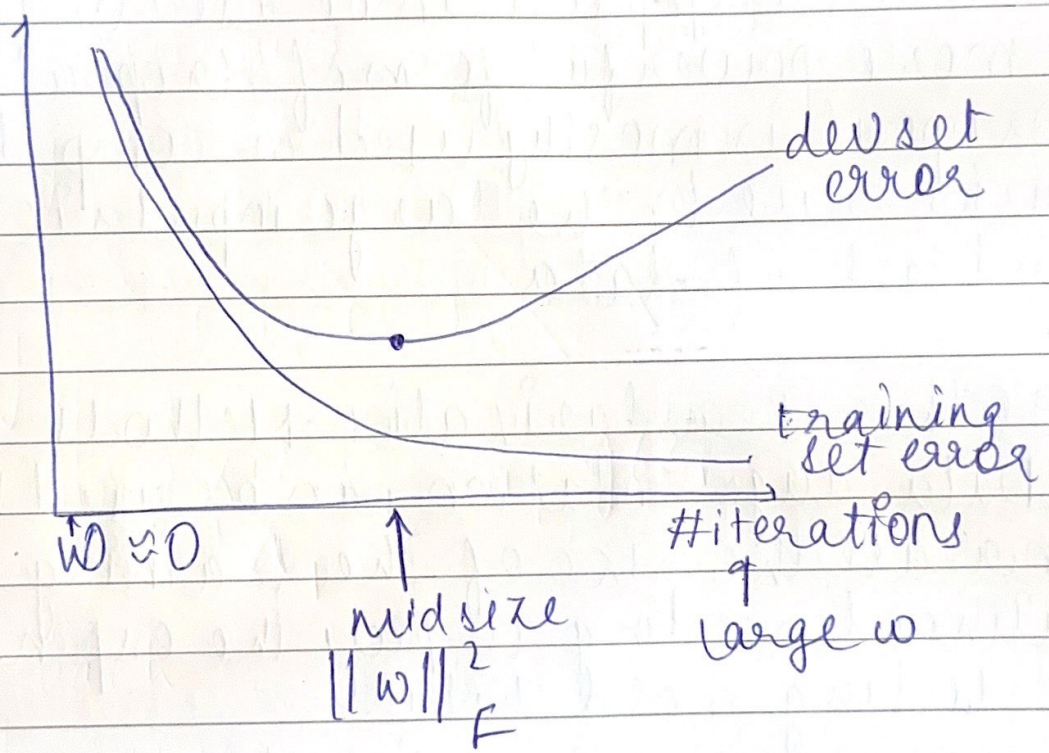
- At test time, dropout is not used as it would add noise to the predictions

→ Understanding Dropout
- Due to randomness, we cannot rely on any feature.

- This encourages the unit to spread out its weights, which has similar effect to L2 regularization
- It is possible to vary the keep-prob by layer
- For layers with a lot of parameters, a lower keep-prob can be used to apply a more powerful form of dropout
- Dropout is mostly used in computer vision due to the large input sizes and lack of data

→ Other Regularization Methods
- Data augmentation can be used to increase the size of the training set without needing to pay the expense of collecting more data
- Eg: flipping images horizontally, taking random crops of the image, and imposing random rotations and distortions
- Early stopping is another technique used to reduce overfitting, which involves

plotting the training and dev set errors and stopping the training of the neural network when the dev set error and stopping the training of the neural network when the dev set error is at its lowest.



- Early stopping couples the tasks of optimizing the cost funct^n and reducing variance, which can make the search space of hyperparameters more complex

SETTING UP YOUR OWN ~~OPTIMIZING~~ OPTIMIZATION PROBLEM

→ Normalizing Inputs

- It is a technique used to speed up training of a neural network
- It corresponds to two steps:
  ① subtract out / zero out the mean (μ)
  ② Normalize the variances (σ)

Subtract mean

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

$$x := x - \mu$$

Normalize variance

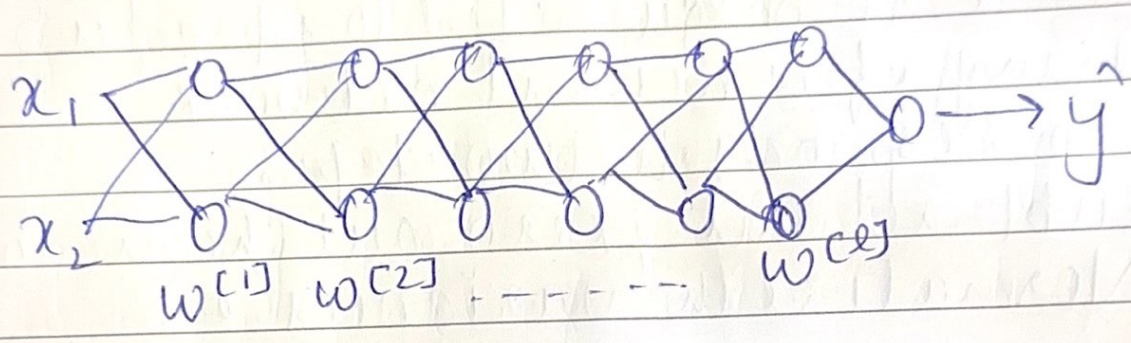
$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} **2$$

$$x /= \sigma$$

- Normalizing inputs helps to make the cost function more symmetric and easier to optimize
- If input features come from different scales, then it is important to normalize the features.
- If input features come from ~~same~~ similar scales, normalizing is less imp. but beneficial

→ Vanishing / Exploding Gradients

- These problems occur when training very deep neural networks



$g(z) = z$

$b^{[l]} = 0$

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

$w^{[1]} x = z_1 = a_1 = g(z_1)$

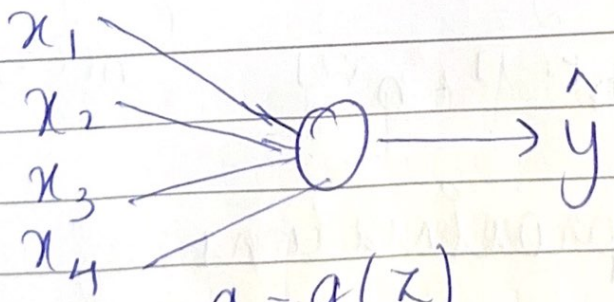
$w^{[2]} a_1 = z_2 = a_2 = g(z_2)$

- If each weight matrix $w^{[l]}$ is just bit larger than one times the identity $\hat{y}$ will be very large and grow exponentially with no. of layers
- Similarly if weight matrix $w^{[l]}$ is less than one then $\hat{y}$ will decrease exponentially.

- The same argument can be used to show that the derivatives or gradients will also increase/decrease exponentially as a function of the no. of layers

→ Weight Initialization for Deep Networks

- Vanishing and exploding gradients are problems that can occur in very deep neural networks
- To solve the above problem, we must be more careful for the choice of random initialization for the neural network



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

- To make z not blow up / become too

small, the variance is set to $1/n$, where n = no. of input features

- If using ReLU funct^n, the variance of w should be set to $2/n$
- This helps reduce the vanishing and exploding gradients problem
- Other variants of initialization are used are:

① Xavier Initialization

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

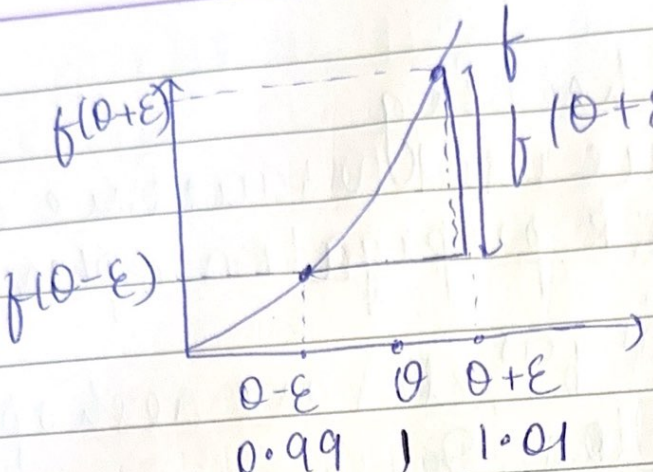
tanh $= \sqrt{\frac{1}{n^{[l-1]}}}$

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

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

→ Numerical Aproximations of Gradients

- Numerical aproximations of gradients can be used to verify the correctness of a backpropagation implementation



$$\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta)$$

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

$g(\theta) = 3\theta^2 = 3$

approx error $= 0.0001$

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

$$f'\theta = \frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon} \longrightarrow \theta(\varepsilon)$$

This two sided difference formula is what is used when doing gradient checking.

→ Gradient Checking

- It is a technique used to save time and find bugs in back propagation implementation
- Parameters ($W^{[l]}$, $b^{[l]}$, etc) are reshaped into a giant vector, θ.
- Cost function (J) is now a function of theta
- Implementation:

for each i:

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

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

$$d\theta_{approx} \approx d\theta$$

- $d\theta_{approx}$ and $d\theta$ are compared by calculating Euclidean distance b/w them

$$\| d\theta_{approx} - d\theta \|_2$$

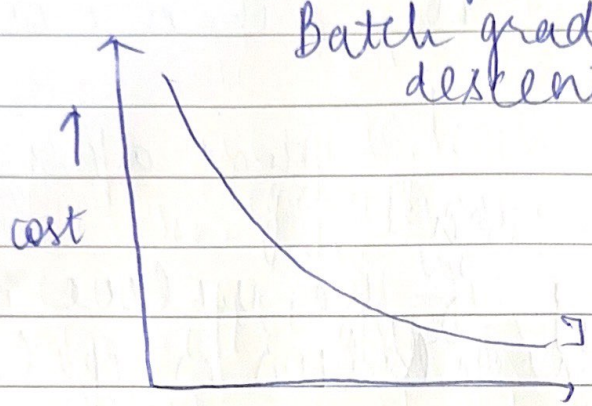
- If distance is of order 10^{-7} or smaller, the derivative approximation is correct
- If distance is $> 10^{-5}$, then a bug is suspected
- If distance is $> 10^{-3}$, then a bug is definitely present
- After debugging, if the value is small, the implementation is likely correct

→ Gradient Checking Implementation Notes

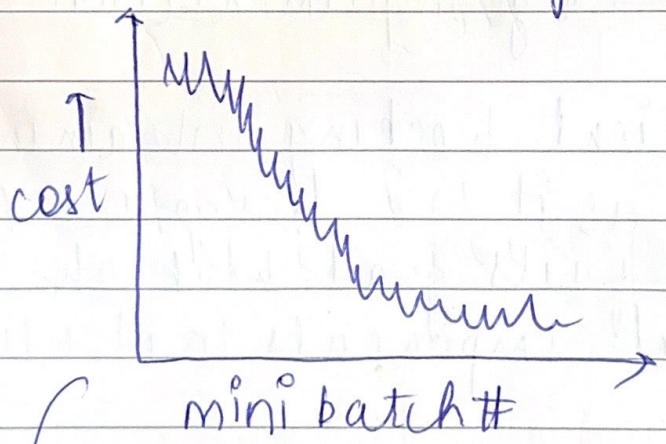
- Don't use gradient checking in training, only to debug as it is a slow process
- If algorithm fails grad check, check the individual components to identify the bug
- Remember regularization
- Grad check doesn't work with dropout so turn off dropout and use grad check to double check the algorithm without dropout
- Run at random initialization; perhaps again after some training

WEEK 2 : OPTIMIZATION ALGORITHM

→ Mini - batch gradient descent



Mini batch gradient descent



- The cost may not decrease on every iteration, but should trend downwards
- Size of minibatch should be b/w 1 to the size of training set.

- Batch gradient: takes longer time per iteration
- Stochastic gradient is noisy and never converges
- Mini batch gradient descent is more efficient method
- Stochastic gradient is noisy and can be reduced by using smaller learning rate
- If training set is small, then use gradient descent
- Typical mini batch sizes are b/w 64 and 512, it is faster if size is a power of 2
- Make sure the mini-batch fits in CPU/GPU memory
- Mini batch is a hyperparameter that can be searched to find most efficient value

→ Exponentially weighted Averages
- It is a faster optimization algorithm
- General formula EWA $\Rightarrow V_t = 0.9 V_{t-1} - (0.1)\theta_t$

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

→ Gradient descent with momentum
- The basic idea is to compute an exponentially weighted average of gradients and use that to update weights
- This algorithm helps to dampen oscillation in the vertical direction and move quickly in the horizontal direct^n
- Compute dw and db on the mini batch
- $V_{dw} = \beta V_{dw} + (1-\beta) dW$
  $V_{db} = \beta V_{db} + (1-\beta) db$

$w = w - \alpha V_{dw}$
$b = b - \alpha V_{db}$

→ RMSprop
- Root Mean square prop
- It works by keeping an exponentially weighted average of the squares of the derivatives
- Compute dw, db on current mini batch
  $S_{dw} = \beta S_{dw} + (1-\beta) dW^2$
  $S_{db} = \beta S_{db} + (1-\beta) db^2$

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

→ Adam Optimization Algorithm

- Implementation:

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

- On iteration t, compute dw and db using current mini batch
- $V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw$ } momentum
- $V_{db} = \beta_1 V_{db} + (1-\beta_1) db$

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

- corrected

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

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

→ Learning Rate Decay

- Reduces learning rate for faster algorithm
- This helps to reduce noise in the mini batches and allows the algorithm to converge to the minimum more quickly

$$\alpha = \frac{1}{1 + \text{decayRate} \times \text{epochnumber}} \times \alpha_0$$

$$\alpha = 0.95^{\text{epochnumber}} \cdot \alpha_0$$

$$\alpha = \frac{k}{\sqrt{\text{epochnumber}}} \cdot \alpha_0 \text{ or } \frac{k}{\sqrt{t}} \cdot \alpha_0$$

WEEK 3

→ Normalizing Activations in a Network

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

- Implementing Batch Norm
  Given some values in NN
  $z^{(1)} \ldots\ldots\ldots\ldots z^{(m)}$

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

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

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

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

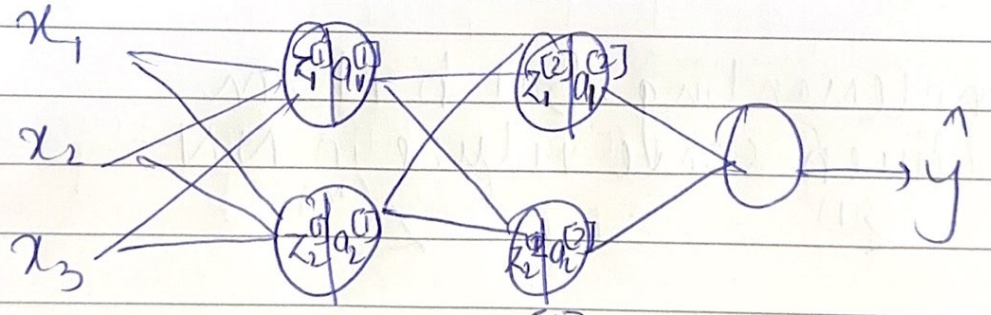
learnable parameters of model

if $\gamma = \sqrt{\sigma^2 + \varepsilon}$ (and) $\beta = \mu$

then

$$\tilde{z}^{(i)} = z^{(i)}$$

→ Fitting Batch Norm into a Neural Network



$$X \xrightarrow{W^{[1]} b^{[1]}} z^{[1]} \underset{BN}{\overset{\beta^{[1]} \gamma^{[1]}}{\rightleftarrows}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]})$$

$$\downarrow W^{[2]} b^{[2]}$$

$$a^{[2]} \leftarrow \tilde{z}^{[2]} \underset{BN}{\overset{\beta^{[2]} \gamma^{[2]}}{\rightleftarrows}} z^{[2]}$$

- working with mini batches

$X^{\{t\}}, Y \xrightarrow{w^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{[1]} \rightarrow g^{[1]}(\tilde{Z}^{[1]}) = a^{[1]} \rightarrow \dots \rightarrow Z^{[L]}$

- Implementing gradient descent

for $t = 1 \dots$ no. of mini batches

compute forward prop on $X^{\{t\}}$

in each hidden layer, use BN for $Z^{[l]}$ with $\tilde{Z}^{[l]}$

- use backprop and compute $dw^{[l]}$, $d\beta^{[l]}$, and $d\gamma^{[l]}$

update parameters

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

$\beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]}$

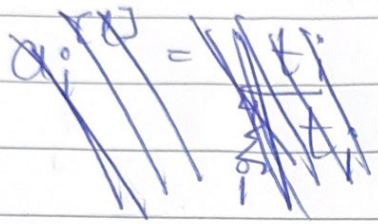
$\gamma^{[l]} := \dots$

→ Softmax Regression

- It is a type of logistic Regression used for multi-class classification
- C: no. of classes.
- $\hat{y}$: 4 dimensional vector, containing the probabilities of each of the four classes

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

$a_i^{[l]} = t_i$



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