Distinction btwn ML & DL:

* ML works really well with smaller datasets. whereas DL works comparatively well with larger datasets.
* **Feature Engginering**: choosing the relevant features that are required for our ML algorithms by mathematical analysis is called feature engg. It also includes **scaling (Normalization, Standardization)**.
  Feature Engg = Feature Cunning.
* In ML, we have to manually carry out Feature Engg. But in DL, everything is done automatically, by **Neural Networks**.
* **Principle Component Analysis**: Dimensional Reduction. It reduces the no. of features into less no. of features.
  (eg) 5 features into 2 features.

Interview Question ①: Can we do reverse process against PCA?
Yes, we can convert the 2 dimensional features into 5 dimensional features using **manifold learning**.

Interview Question ②

* **Feature Extraction**: Creating new features from the existing set of features. (i.e) Dimensionality Reduction (PCA, Auto encoders, LDA). The extracted features are then loaded into ML Algorithms.
* **Feature Selection**: Changing the no. of features from 5 to 2 by selecting only the important ones without changing their size (or) dimension.
  (eg) Recursive feature elimination (RFE), Lasso / Ridge Regularization.

Artificial Intelligence: Any Agents (cars, phones, larger machines) that can take decisions on its own & implement them is called AI agents & the process is called AI. This does not require human interaction.

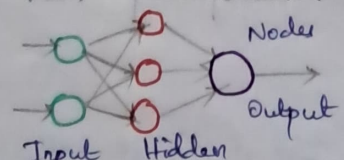
"Neural Networks are the core components of Deep learning".

AI : ML : DL

Reinforcement Learning: Reinforce - change. Concept: we learn from our mistakes & never commit it again.

Deep Learning:
→ DL uses Tensorflow & pytorch for building models.
Framework

* **Topology of a Neural Network**:



→ A NN with multiple Hidden layers is called a **Multi layer perceptron**.

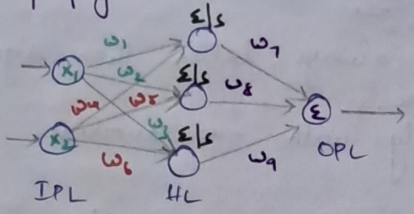
→ The input layer nodes corresponds to the number of features in the dataset.
→ The Hidden layer nodes is a choice of the individual. we can take any number.
→ The no. of Hidden layer is also of user's choice.

→ If the no. of node in the output layer is 1 then it is a **Regression problem**, if it is more than 1 then it is a **classification problem**.

→ **weights** : Tells the importance / weightage of features being passed into the NN. If $w \uparrow$, the influence of that feature in the predicting the o/p is very high.

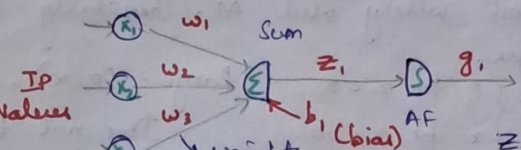
→ **Bias** : It is always added to the hidden layer to do course correction. (Preference for a particular feature).

→ **Forward propogation** :



Σ - Summation. (Values of x_1, x_2, x_3 are combined together).

S - Activation function. (Always applied to the hidden layer).



weights are defined randomly at the intial stage.

$$z_1 = w_1x_1 + w_2x_2 + w_3x_3 + b_1$$

$$y = m_1x_1 + m_2x_2 + m_3x_3 + c$$

$y = mx + c$: This equation is used to create a line for 2-D data. (i.e.) Two features in the data.

* The output of z_1 is always linear. But the real world data is often non-linear in nature. Now, to introduce **non-linearity** into the NN model, we use **Activation functions**.

$\frac{1}{1+e^{-x}}$: **The Sigmoid function** : used in logistic reg to produce classified results. This will return in a sigmod curve.

Activation function (g_1) : $\frac{1}{1+e^{-(z_1)}}$. This would return the g_1 value with non-linearity.

$$y\text{-pred} = g_1 \cdot w_7 + g_2 \cdot w_8 + b_3$$: y-pred is the actual o/p of our NN model.

→ **Backward propogation** : Now if the first set of predicted values doesn't match with the actual values, we do BP and tweak & change the **weight & bias values** to obtain better results. This is done based on the error rate. Neural Networks are also called as **Universal function Approximators**.

* **chain rule**: A concept in calculus. The y-prod value is calculated based on various factors such as w_7, w_8, b_3. And again w_7, w_8, b_3 values are depending on various other factors such as w_1, w_2, w_3 etc the process of mapping a relationship between these values is called as **chain rule**.

* **Gradient descent** : It is the way by which we increase the values of our weights & baises. To find the weight of w_7,

$$w_7^+ = w_7 - \eta \frac{\partial \text{cost}}{\partial w_7}$$ η - eta : is a very small number - 0.01.

∂_{cost} = Cost function = $(y_{prod} - y_{actual})^2$: This is the error

Classification: * A classification problem with 2 classes (eg) Yes/No is called as Binary classification.

* More than 2 classes, Multi-class classification.
* **Multi label:** Multiple labels. → for MLC, we use sigmoid func.

(Eg) An image containing dog & cat. op could be more than 2 labels.
(Multi labels)

Activation Functions:

→ **Tanh function:** Hyperbolic trignometric function.

$$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1.$$

* The tanh works almost always better than sigmoid as optimization is easier here.
* It can deal more easily with -ve numbers.
* Range: (-1, 1).

→ **ReLU function:** Most widely used AF. Generally implemented in the Hidden layer.

$$f(x) = \begin{cases} 0 \text{ for } x < 0 \\ x \text{ for } x >= 0. \end{cases}$$ Range: $(0, \infty)$. Does not take any -ve value.

* ReLU: Rectified Linear unit.

→ **Leaky ReLU function:** Allows a small -ve value to pass during the back propagation. This overcomes the drawback of using normal ReLU. $f(x) = \begin{cases} 0.01x \text{ for } x < 0 \\ x \text{ for } x \geq 0. \end{cases}$ Range: $(-\infty, \infty)$.

→ **Softmax:** Is used when we have multiple classes. It is useful for finding out the class which has the max. propability.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, \quad j = 1, 2 \ldots K \qquad \text{Range: } (0, 1).$$

* It is ideally used in the o/p layer of the NN where we are actually trying to attain the probabilities to define the class of each I/p.

→ **General tip:**

modules & libraries & packages & frameworks. Generally libraries
↳ cannot be edited, deleted. can only be imported.
But with framework, we can edit anything we want.
(Eg) Tensorflow, PyTorch.

Collection of — Collection of

→ **Tensorflow:** Open-source software library for high-performance numerical computations. It is developed by Google.

→ **Keras:** A high-level API that runs on top of TensorFlow, CNTK etc.

↳ **Tensors:** A multi-dimensional array where data is stored.

→ Epoch: How many times the Neural Network has to go through the entire dataset. It is defined by us to arrive at the correct weights & biases. this takes a lot of time. To solve this time consuming issue, we need to batch the dataset into different collections.

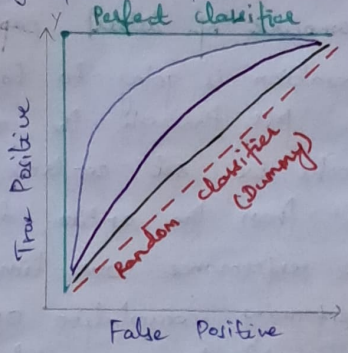
→ To display all the columns in the given dataset:
pd.set_option('display.max_columns', None).

→ df['gender'].value_counts() // used to determine whether a feature is categorical or not.
O/P: Male 2144
Female 3461

→ df['gender'] = df['gender'].map({'Female': 0, 'Male': 1}).
Converts Female into 0 & Male into 1 in the entire dataset. We can do this instead of label encoding if we have only 1 column that needs to be transformed.

→ AUC-ROC: Area Under Curve - Receiver Operating Characteristic

* Works only for classification problems.
* It is usually plotted for False Positive Rate (x), True positive (y).



— - Logistic Reg
— - RF
— - Some other classification Algo. (SVM)

The ROC Curve determines that the LR is the best suited Algo for this problem statement as the curve is more bent towards perfection (covering max area).

* It gives us auc_roc metrics. It should be btwn 0 to 1.

→ Dask: Similar to pandas. Has all the methods listed in Pandas. Organisations uses dask instead of pandas as it can handle multi-high purposes data.

Convolutional Neural Networks: Majorly used for image processing. It is more advanced than ANN's and are used when the data has a spatial structure.

* Image: 0 - 255 (Black light) Generally images are represented in RGB: 3 channels

* ANN's are used for general, normal data: It won't work efficiently on images & videos dataset. └(eg) text data.

* When an image is passed to a CNN, it goes through numerous layers & the layers identifies multiple features of that image (eg) edges, contours, diagonal edges, diagonal contours etc.

* CNN uses a core principle called Convolution Operation. It is a way to detect patterns in an image & translate them into features the network can use to make sense of the image. Applying a filter or a small window to an image to extract important features such as edges (or) textures.

* The filter is always a smaller grid that is slided across the image, one step at a time, covering a new section of the image on each step. The filter is also called as **Kernel**.

Image Grid	*	Kernel Grid	=	Output image
(6 × 6)		(3 × 3)		(6-3+1) × (6-3+1)
(m × m)		(n × n)		(4 × 4)
				(m-n+1) × (m-n+1)

The size, data of the Kernel will be decided by the CNN itself.

* For every convolutional operation that we do, the original dimension of the image will reduce. This is a problem. For a 3-dimensional, 3-channels convolutional operation, a kernel of 3-channels (RGB) is used for getting the output.
* The no. of kernels required is also decided by the CNN itself, As Feature engineering, Feature Extraction is done by DL itself.
* **Strides**: It is the no. of steps the filter slides across the image. We generally have Convolutional Stride = 1. The kernel here moves one step at a time in the image. This impacts both the size of the o/p feature map & the amount of detail captured.
* **Epoch = 50**: Technically, an operation is going to take place 50 times. The operation is: One complete pass through the entire dataset during training. The no. of epochs we set controls how many times the model sees and learns from the entire dataset, which helps in fine-tuning the model's performance over time.
* **Padding**: In General, while performing convolution operation, some numbers in the image grid would be represented multiple times, mainly the middle elements while some numbers would be represented not frequently, mainly the border elements. This causes an uneven feature extraction. To avoid this, we go for padding (i.e.) adding extra columns & rows at the top & bottom & sides, filling it with 0's.

→ padded values.

0	0	0	0	0	0
0	1	2	3	4	0
0	5	6	7	8	0
0	1	2	3	4	0
0	5	6	7	8	0
0	1	2	3	4	0
0	0	0	0	0	0

(5 × 4) to (7 × 6)

→ This will make sure that all the border elements also get represented well.

→ If padding = 1, we add one row & one column at all sides. If it is 2, 2 rows & 2 columns will be added.

* **Important reasons for padding:**
  → To retain the original dimension of the image.
  → Feature representation on the borders.

* **Pooling**: Is a technique used to reduce the spatial dimensions (width & height) of the I/p feature map while retaining the important information. It helps reduce the computational cost, prevents overfitting, & reduces the complexity, making the model more robust. Types:
  - → **Max pooling**: Takes the maximum value from a patch of the feature map.
  - → **Average pooling**: Takes the avg. value.
  - → With pooling, the extracted features are very crisp and they make the o/p feature map (image) more clear & enhanced.
* We can create a multi-layer convolutional operation (CNN) and can add ANN to the model. This is a complete architecture.
* **Flattening**: Converts a 2D array into 1D array. This is because ANN's only accept 1D arrays & to combine ANN's with CNN's we flatten the output of the CNN & establish the connection.

(eg) $\begin{bmatrix} 1 & 6 & 3 & 2 \\ 4 & 2 & 1 & 5 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 \\ 6 \\ 3 \\ 2 \\ 4 \\ 2 \\ 1 \\ 5 \end{bmatrix}$

* **Numpy** is the most important library to use all kinds of Neural Networks. Alongside we need to import **Tensorflow & Keras**.
* **MNIST Dataset**: We generally use this dataset for all our Neural Network related works & use-cases as it is small & has 28×28 dimensional data.
* **Method chaining**: using one method inside another method.
  (eg) tf.keras.utils.to_categorical().
* **Data augmentation**: Technique used to artificially increase the size & diversity of a training dataset by creating modified versions of the existing data. This helps the model generalize better & improves its performance by exposing it to a wider variety of examples.
  Techniques: Flipping, Rotation, Zooming, Brightness Adjustment.
  Concept: By following these concepts, instead of one image, we have multiple versions of the same image.

Steps in writing Code: (Simple NN):

→ **Import libraries**: Numpy, pandas, Tensorflow (Keras), Keras dataset: MNIST.

→ **Split train, test**: Divide into 4 categories:
  * x train images, y train label. } = mnist.load_data().
  * x test images, y test label.

→ **flatten the data**: Convert multi-dimensional input into 1D.

→ **Normalize the data**: Normalize the data so that the data points can be between 0-1.

→ **Encode the data**: Convert categorical to numerical data.

Perceptron: ①

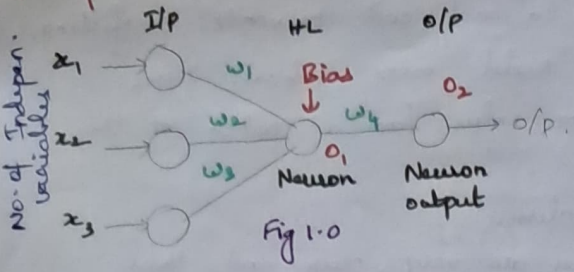


Fig 1.0

→ We can have any no. of hidden layers & any no. of neurons.

→ In hidden layer, there would be two operations:

* $\Sigma x_i w_i$.
* Activation functions.

$$\Sigma x_i w_i = x_1 w_1 + x_2 w_2 + x_3 w_3 \Rightarrow w^T x$$

This equation is passed into the Activation Function.

* **Weights**: Adjustable settings that determine how much importance one input has in predicting the output. Weights help the neural network learn from data & make more accurate predictions by emphasizing the most relevant information.
* **Bias**: It helps the network make better predictions by allowing it to adjust the o/p slightly, even when all the I/P's are already considered.
* **Activation Function**: AF's purpose is to introduce non-linearity into the network, which means it allows the network to learn & represent more complex patterns in the data.
  - → ReLU (Rectified Linear Unit): Outputs the inputs if it's positive, otherwise, it outputs zero.
  - → Sigmoid: Outputs a value between 0 & 1 for probability-based predictions. (For Binary Calculations).
  - → Tanh: Outputs a value btwn −1 & 1, useful when the data is centered around 0.
* This entire process is called as **Forward propagation**.
* In our dataset we have y, that is the target variable. In our neural network we get the output as y_{pred} (or) $\hat{y}$. The diff btwn y & $\hat{y}$ $(y - \hat{y})$ is called as Loss function. (Actual − Predicted) The diff. should always be close to 0.
* **Back propagation**: The process of going back & updating our weights & baises inorder to reduce the loss function.
* **Optimizers**: Algorithms (or) methods used to adjust the weights & baises during training to minimize the error.
  - → Stochastic Gradient Descent (SGD)
  - → Adam (Adaptive Moment Estimation).
  - → RMSprop.
* Weight updation Formula: This takes place in the back propogation.

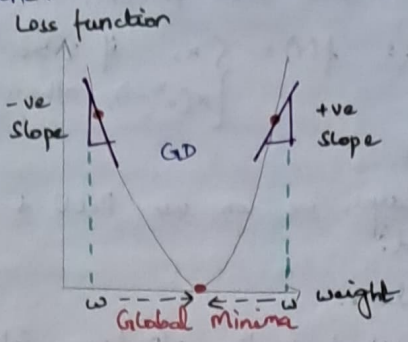
$$w_{new} = w_{old} - \eta \frac{\partial L}{\partial w_{old}}$$

(η → learning rate.)

↓

Gradient of the loss function with respect to the weight.

* **Gradient Descent**: This is a fundamental optimization algorithm, to minimize the loss function by iteratively updating the weights of a Neural Network.



→ For (-ve) slope, the w_{new} is greater than w_{old}.
→ For (+ve) slope, $w_{new} < w_{old}$.
→ The main aim is to converge to the global minima by taking a smaller learning rate, $\eta = 0.01$.

* **Chain Rule of Differentiation**: The main goal of CRD is to find out the $\frac{\partial L}{\partial w_{old}}$ in the weight updation formula. For example, the gradient descent of the loss w.r.t. the weight w_2 is:

$\frac{\partial L}{\partial w_2}$, using chain rule:

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z_2} \times \frac{\partial z_2}{\partial w_2}$$

The chain rule of derivative is basically calculating the loss with.r.to every o/p & weights

Similarly using our Fig (1.0) to find $\frac{\partial L}{\partial w_4} = \frac{\partial L}{\partial o_2} \times \frac{\partial o_2}{\partial w_4}$.

→ All these things are done by "optimizers".

* **Vanishing Gradient Problem**: Common problem in training DNN, especially those with many layers. It occurs when the gradients of the loss function become very small as they are propagated backwards through the network. This causes slower (or) stalled learning.
→ Several techniques help mitigate VGP:
* ReLU Activation * Batch Normalization * Residual Networks.

VGP = $w_{new} \simeq w_{old}$

→ The VGP generally takes place in Sigmoid Activation function as the derivatives of the sigmoid squashes the value bt. 0 & 0.25. The derivatives are always calculated in the Back propagation.

Activation Functions: (2)

* Sigmoid Function: $\sigma = \frac{1}{1+e^{-x}}$ (Not zero centric).
→ o/p Range: 0 to 1.
→ Derivatives: 0 to 0.25.

* tanh Function: $x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ (zero-centric).
Meaning: the curve passes through 0.
→ o/p Range: -1 to 1.
→ Derivates: 0 to 1.
→ Also called Hyperbolic tangent function.

Slightly solves the VGP.

* ReLU Function: $\max(0, x)$
→ o/p Range: 0 to max
→ Derivates: 0 to 1.
→ When the I/P is -ve, the ReLU will die.
→ The performance is the best in ReLU.
→ Not zero centric.

* Leaky ReLU: $f(x) = \max(0.01x, x)$.
  → o/p Range: Small -ve number to max.
  → It solves the Dead ReLU problem.

* Exponential Linear Units: $f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha(e^x - 1), & \text{otherwise} \end{cases}$

→ For Binary classification:
  * In the Hidden Layer, we can use ReLU & in the o/p layer, we use Sigmoid.

→ For Multiclass classification:
  * In the HL, ReLU & in the o/p layer, Softmax.

→ For Regression problems:
  * In the HL, ReLU & in the o/p layer, Linear Activation.

(3)
Loss functions: The loss functions are different for Reg & classif. problems.

* Regression:
  → Mean Squared Error (MSE)
  → Mean Absolute Error (MAE)
  → Huber Loss

* Cost Vs Loss function: the loss function is calculated for per individual example, while the cost function aggregates the loss over the entire dataset (or) a batch of data. The loss function provides feedback for a single example, while the cost function provides an overall measure of how well the model is performing across the dataset. The training process aims to minize the cost function, avg of the loss functions across all examples.
  (eg) Grading a single student: Loss.
  Grading all the students in a class & calculating the class average: Cost.

* classification:
  → Multiclass: Categorical cross Entropy.
  → Binary: Binary cross Entropy.

(4)
Optimizers: Refer previous page & Gradient Descent for more details.

* Epochs: One forward & one back propagation = 1 Epoch.

* stochastic Gradient Descent: In Gradient descent algorithm, it computes the gradient (the partial derivatives of the loss function w.r.t the weights) over the entire training dataset at each step. this approach known as Batch Gradient Descent, is computationally expensive when the dataset is large. Stochastic GD addresses this issue by updating the weights more frequently, using only one (or) a small batch of training sample at a time.

1 Data point → Epoch 1 (FP → BP) = Iteration 1

Normal SGD: This would take more processing time & effort as we are processing only one datapoint / Epoch.

* **Mini Batch SGD:** To overcome the problem of Normal SGD, we use mini batch SGD.

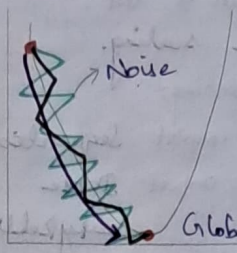
1000 Datapoints → FP → Batch_Size = 1000 → Iteration 1 → BP

We give 1000 Datapoints instead of 1 in 1 Epoch.

→ Resource Intensive, Less time complexity, convergence will be better.
→ Moderate Noise occurs when coming to the global minima.
→ GD will have no noise while the SGD has the most.

* **Mini Batch SGD with momentum:**

→ To remove this noise, we use momentum.
→ It smoothens the journey of the data points reaching the Global Minima



SGD - High Noise
MB-SGD : Minimal Noise
GD : No noise.

→ **Exponential weighted Average:** Used to give more importance to recent data points while still considering past data. This is used to remove the noise & smoothen the curve.

$$Vdw_t = \beta * Vdw_{t-1} + (1-\beta) * \frac{\partial L}{\partial w_{t-1}}$$

Vdw_t / Vdw_{t-1} : value of derivative; β : Hyper parameter

Recap:
→ **Gradient Descent:** Time, Resource complexity, can't be used for large datasets.
→ **How to remove noise in GD:** smoothen the curve with optimizers such as Mini Batch SGD with momentum.

* **Adaptive Gradient Descent (Adagrad):** Deigned to adapt the learning rate (η) for each parameter during training.

$$w_{new} = w_{old} - \eta' \frac{\partial L}{\partial w_{old}} \quad \text{where} \quad \eta' = \frac{\eta}{\sqrt{\alpha_t + \epsilon}}$$

The learning rate is not fixed.

This value gets decreasing as the data point reaches the Global minima ← $\eta' = \frac{\eta}{\sqrt{\alpha_t + \epsilon}}$

ϵ - Small value added to avoid η getting divided by 0.

$$\alpha_t = \sum_{i=1}^{t} \left(\frac{\partial L}{\partial w_{old}} \right)^2$$

* **Adam Optimizer:**
Momentum + RMSprop (Adaptive Learning Rate).
→ It adjusts the learning rate for each parameter dynamically (Adaptive) and makes use of momentum to accelerate the optimization process. This is applicable for both weights & biases.

Final weight & bais updation Formula

$$Vdw_t = \beta \times Vdw_{t-1} + (1-\beta) \frac{\partial L}{\partial w_{t-1}}$$

$$Vdb_t = \beta \times Vdb_{t-1} + (1-\beta) \frac{\partial L}{\partial b_{t-1}}$$

* Smoothening
* Learning Rate becomes Adaptive.

ANN Implementation : (5)

* Libraries: Tensorflow, Numpy, Pandas, matplotlib, scikit-learn.
* Load the dataset.
* The first step is always to divide the dataset into dependent & independent features.

```
X = dataset.iloc[:, 3:12] // Includes all the rows, columns = 3 to 12
y = dataset.iloc[:, 13] // Includes all the rows, only 13th column.
```

* Convert the categorical into numerical data - **Feature Engineering**.
  → we can use : pd.get_dummies(X['column Name'])
* Split the features into train & test.
* The next thing is **feature scaling**.
* Create ANN. Import the following:

```
tensorflow.keras.models import Sequential
tensorflow.keras.layers import Dense
"  "  "  "  LeakyReLU, PReLU, ELU.
"  "  "  "  Dropout.
```

Libraries & Activation Functions for our ANN.

Defines the shape & structure, network of the neurons

* **Sequential** : A model-building approach in Keras that allows us to stack layers one after the other, defining a NN in a linear sequence. It's a convinient way to build simple neural networks.
* **Dense** : A Dense layer is a fully connected layer in a NN. This means every neuron in the current layer is connected to every neuron in the previous layer. Parameters of Dense:
  → **Units** : No. of neurons in the layer.
  → **input_shape**: Shape of the I/P data. (First layer).
* **Dropout**: It is a regularization technique used in NN to prevent overfitting. It works by randomly dropping out (i.e. **setting to zero**) a subset of neurons during training iteration.
  → Prevents overfitting.
  → Improves Generalization.
  → Acts as a form of model Averaging.
  → This is applied only on the training data.

  (eg) dropout = 0.5
  50% of the neurons will be deactivated.

* Initialize ANN:

```
classifier = Sequential()
```

* Add the I/P layers:

```
classifier.add(Dense(units = 11, activation = 'relu'))
```

creates a fully connected I/P layer with 11 neurons & the relu activation is applied to the next hidden layer.

* Hidden layer:

```
classifier.add(Dense(units = 7, activation = 'relu'))
```

* O/P layer:

```
classifier.add(Dense(units = 1, activation = 'sigmoid'))
```

* Train the entire Neural Network:

```
classifier.compile(optimizer = 'adam', loss = 'binary_crossentrophy', metrics = ['accuracy'])
```

By default Adam uses the learning rate of 0.01

model_history = classifier.fit(x_train, y_train, validation_split = 0.33, batch_size = 10, epochs = 1000).

* **Early stopping**: Let's say we set epochs = 1000. At one point of time, the accuracy remains stagnant after certain no. of epochs. Early stopping makes sures that the model execution stops at that particular epoch & doesn't proceed executing the remaining epochs. This saves time & memory. Stop training when a monitored metric has stopped improving

* Predict the test set results.

  y_pred = classifier.predict(x_test).

* classifier.get_weights(): Displays all the weights used in the NN in the form of array.

Black Box Model vs White Box Model : (6)

* BBM: Internal workings are not easily interpretable or accessible to human understanding.

  (Eg) Deep Neural Networks: ANN, CNN, RNN, RF etc.

* WBM: Decision-making process is transparent, understandable by humans.

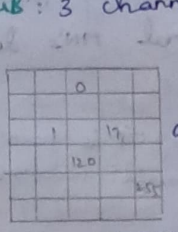
  (eg) Decision Trees, Linear Reg., Log Reg.

(7)

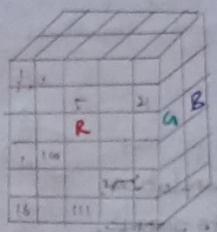
Convolutional Neural Networks: Used for Image processing, object detection & other complex processes. In image there are 2 types:

→ Black & white: 1 channel. 0-255 values.

→ RGB: 3 channels.

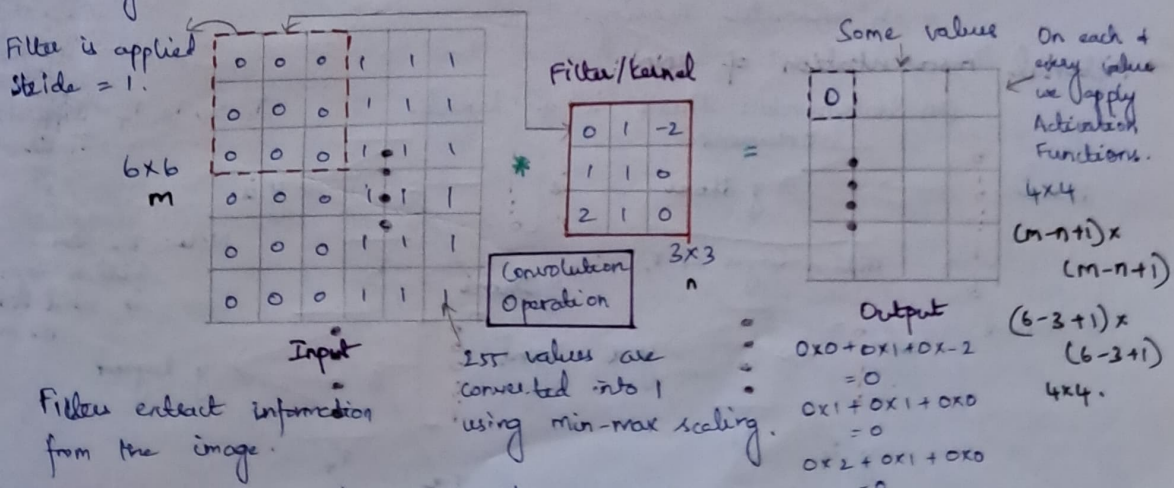


6×5×(1) channel



6×5×(3) channels.

* **Min-Max Scaling**: A Normalization technique used to scale the pixels values of I/P images to a specific range btwn 0 & 1. CNN perform better when the input features are on a similar scale. This ensures that the pixel values do not skew the gradients during training. (i.e.) 0-255 is converted to 0-1.

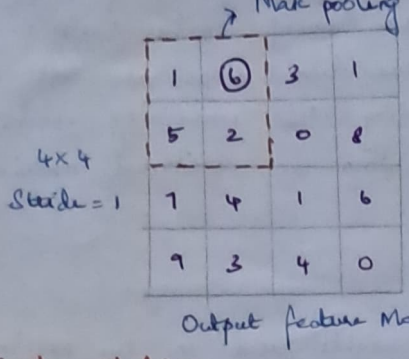


Filters extract information from the image.

* **Padding**: Refer previous page for detailed explanation.

This is used to retain the original dimension of the image.

* **Max pooling**: Extracts more useful info from the output by reducing computational cost using a window.



Implementation:

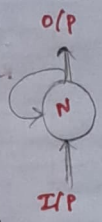
* Again we use model.Sequential().
* Create convolutional layers:

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3,3), activation='relu', input_shape=(32,32,3)))
```

(32 → No. of filters; (3,3) → Shape of filters.)

(2) `model.add(layers.MaxPooling2D((2,2)))`.

Recurrent Neural Networks: Designed to process sequences of data, such as time-series data, text or speech. RNN's have a memory that allows them to retain information from previous inputs, making them useful for tasks where context or order matters. Traditional ANN's don't remember past inputs.



The o/p is given back to the Neural Network itself.

RNN's are mainly used for NLP.
Other uses / use cases: translation, Sentiment Analysis, Auto-suggest, Named Entity Recognition (NER).

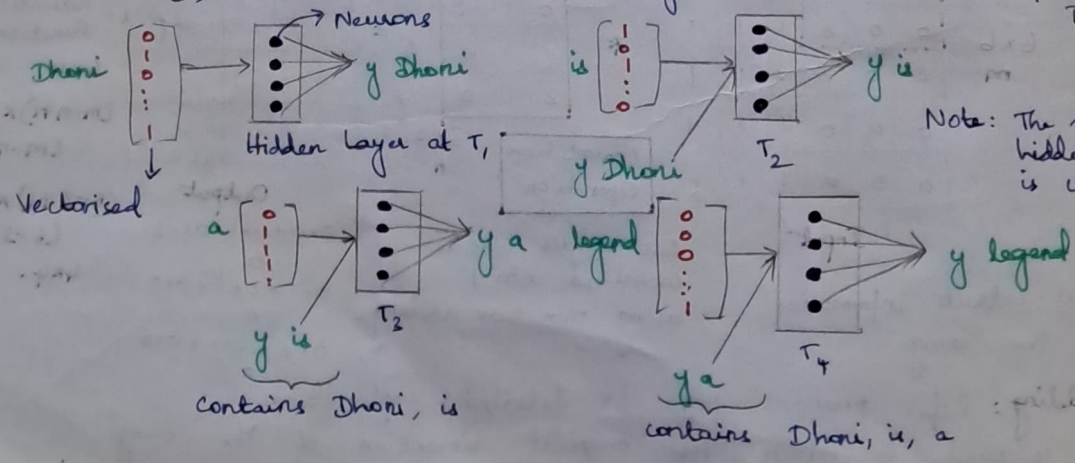
Sequence Model Problems: The sequence of the I/P & o/p is really important. For this problem, normal neural networks cannot be used. When we use ANN instead of RNN:

* Variable size of I/P & O/P neurons.
* Too much computation.
* No parameter sharing.

→ **Pictorial representation of RNN:**

(Eg) Statement: Dhoni is a legend.

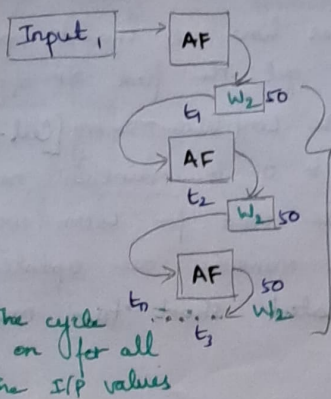
T_1, T_2, T_3, T_4 – Different Timestamps



* Encode the sequence: Convert words into vectors.
* Feed words sequentially: Input each word into the RNN, updating the hidden state (memory) with each step.
* Update hidden state: Store info from previous words & combine it with the current word.
* Predict the O/P: Generate predictions (eg. next words) based on the final hidden data.
* Calculate loss: Compare predictions to actual outputs & compute error.
* Backpropagation: update weights using BP, to reduce the error.
* Iterate: Continue training the model with new data.
* No matter how many times we unroll a RNN, we never increase the weights & baises that we have to train.
* One big problem with RNN is that, the more we unroll / add inputs a RNN, the harder it is to train. This is called the **vanishing / Exploding Gradient problem.**

→ **Exploding Gradient problem:**

* General RNN: with EGP



Let's say the weight W_2 is 50. The same W_2 is maintained at all the timestamps & the I/P is amplified 50^2 (in our case) times before it displays the O/P. This huge weight of 50 causes the **Exploding Gradient problem.**

* In order to find the parameter values that gives us the lowest value for the Loss function, we usually want to take relatively small steps in our gradient. (Fig 1.1)

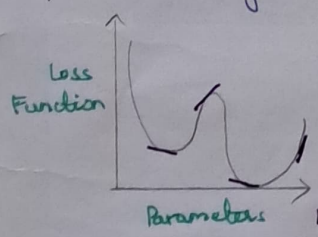


Fig 1.1

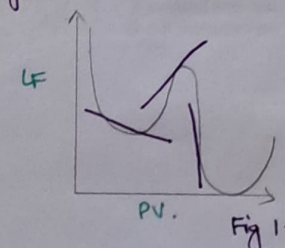
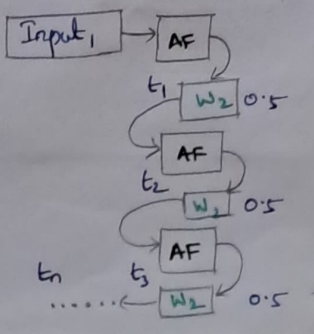


Fig 1.2

* However, when the gradient contains a Huge number (50), we'll end up taking relatively larger steps. (Fig 1.2).
* One way to prevent the Exploding problem would be to limit weight values < 1.

→ **Vanishing Gradient problem:**

* General RNN: with VGP



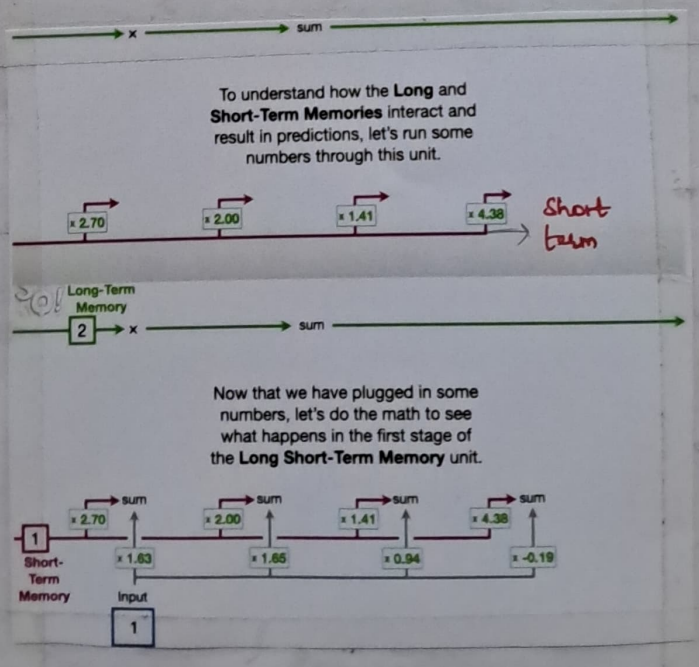
In this case, we have assigned W_2 to 0.5. We have 3 iterations (i.e.) $t = 3$ thus 0.5^3 results in a value very close to zero. If the I/P gets multiplied by this value the O/P would also be very close to 0. This is the **Vanishing Gradient Problem.**

Long-Short term Memory (LSTM RNN):

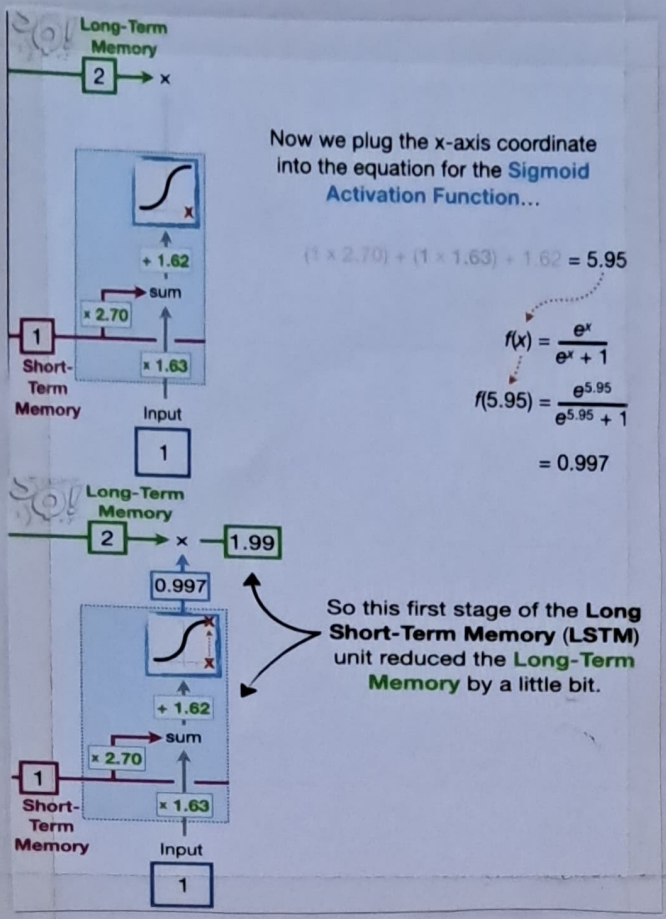
LSTM's are the solution for this ① Exploding/Vanishing Gradient Problem that occurs in normal RNNs. In general, Vanilla RNNs are hard to train because the gradients can explode (or) vanish.

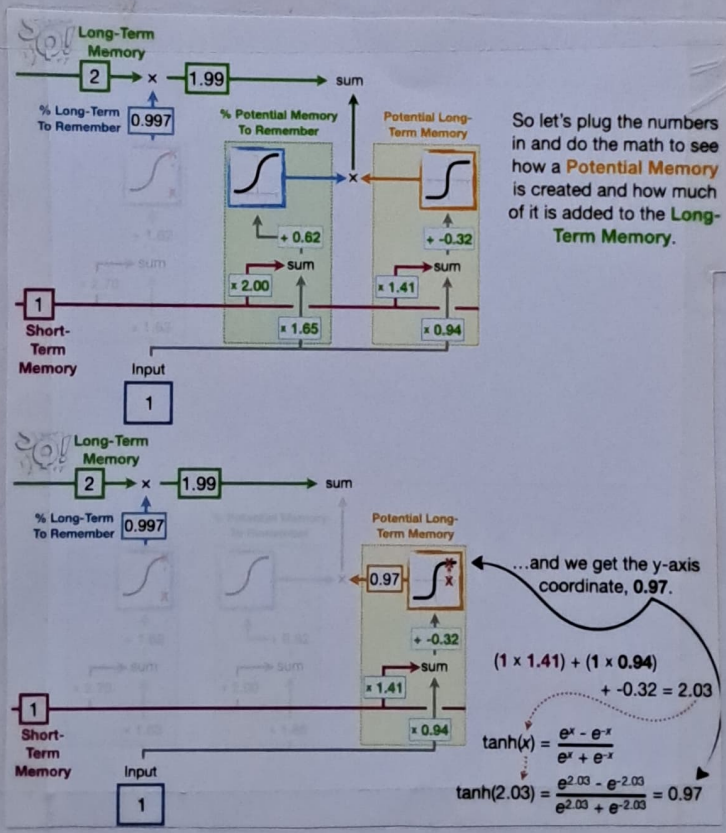
* The main idea behind how LSTM works is that instead of using the same feedback loop connection for events that happened long ago & events that just happened yesterday, LSTM uses 2 separate paths to make predictions about tomorrow. One path for **Long-term memories** & one is for **short-term memories**. This is a much more complicated unit unlike Vanilla RNN's.
* Unlike the other RNN's, the LSTM uses **Sigmoid & Tanh Activation Functions**. Refer Activation Func. section in the previous notes for more clarity on how these AF works **(REF-2)**.
* The different stages of LSTM:
  - → **Forget Gate**: Determines what % of the Long-term Memory will be remembered.
  - → **Input Gate**: Determines how we should update the LT Memory.
  - → **Output Gate**: Determines how we should upate the Short-term memory to get the final o/p of the LSTM.
* During the Initial process, the Long-term Memory [Cell-State] & the short-term memory [Hidden-State] are set to 0 to maintain neutral start.
* The same weights & baises are used for LSTM everytime we unroll it for each input. The LT & ST memories are updated at each cycle. The final o/p is the **updated short-term memory**.

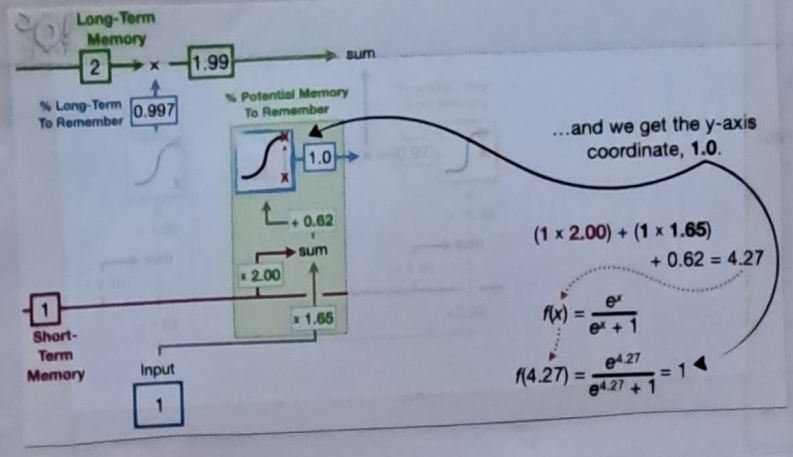
LSTM Working in a pictorial Representation.



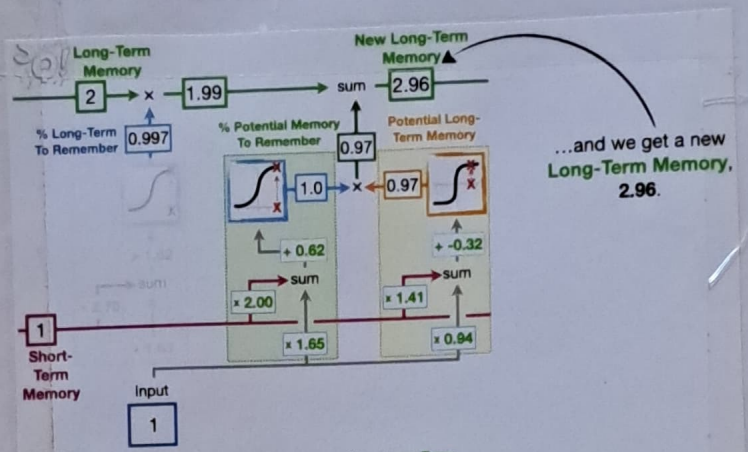
Initially the LTM is set to 2, STM to 1.

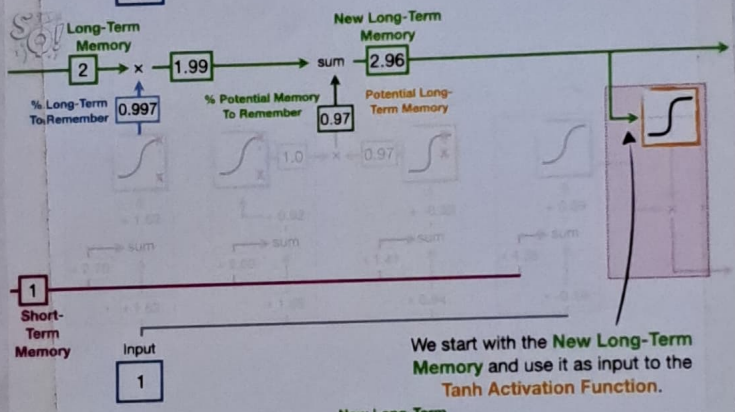

Long-Term Memory
2
x
+ 1.62
sum
x 2.70
x 1.63
1
Short-Term Memory
Input
1
Now we plug the x-axis coordinate into the equation for the Sigmoid Activation Function...
(1 x 2.70) + (1 x 1.63) + 1.62 = 5.95
$f(x) = \frac{e^x}{e^x + 1}$
$f(5.95) = \frac{e^{5.95}}{e^{5.95} + 1}$
= 0.997
Long-Term Memory
2
x
1.99
0.997
+ 1.62
sum
x 2.70
x 1.63
1
Short-Term Memory
Input
1
So this first stage of the Long Short-Term Memory (LSTM) unit reduced the Long-Term Memory by a little bit.
First stage: Forget Gate

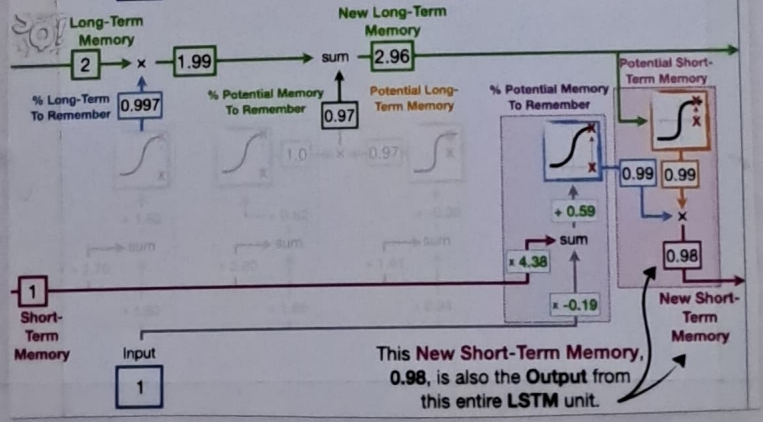

Long-Term Memory
2
x
1.99
sum
% Long-Term To Remember
0.997
% Potential Memory To Remember
Potential Long-Term Memory
+ 0.62
+ -0.32
sum
sum
x 2.00
x 1.41
1
Short-Term Memory
x 1.65
x 0.94
Input
1
So let's plug the numbers in and do the math to see how a Potential Memory is created and how much of it is added to the Long-Term Memory.
Long-Term Memory
2
x
1.99
sum
% Long-Term To Remember
0.997
Potential Long-Term Memory
0.97
+ -0.32
sum
x 1.41
x 0.94
1
Short-Term Memory
Input
1
...and we get the y-axis coordinate, 0.97.
(1 x 1.41) + (1 x 0.94) + -0.32 = 2.03
$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
$\tanh(2.03) = \frac{e^{2.03} - e^{-2.03}}{e^{2.03} + e^{-2.03}} = 0.97$



2nd Stage: Input Gate







Third stage: Output Gate

(16) **Transformers**: Successor of LSTM, Transformers are deep learning models that process sequences using a "self-attention mechanism" allowing them to focus on different parts of the input simultaneously. This makes them highly efficient at capturing long-range dependencies & relationships in data, without needing to process the I/P step by step like traditional RNN's or LSTM's.

* ChatGPT is fundamentally based on Transformers.
* In general, Neural Networks only accept numbers & not words. So, to convert words to numbers we use multiple techniques & one most widely used technique is called **Word Embedding**.

→ **Word Embedding**:

* Rather than just assign random numbers to words, we can train a relatively simple Neural Network to assign numbers for us. The advantage is that it can use the contexts of words in the training dataset to optimize weights that can be used for the embeddings.
* This can result in **similar words ending up with similar embeddings.**
* This makes the language process easier.
* **Embedding**: Dense, continuous vectors in a high-dimensional space. Embeddings capture the semantic meaning of words by placing similar words close to each other in this space.

(eg) The word "King" has the vector like [0·6, 0·4, 0·2 ...], while "Queen" has [0·5, 0·3, 0·2 ....]

* **Word2Vec**: A technique that uses a neural network to generate word embeddings by predicting words based on their context. It uses 2 main methods:

→ **Bag of words**: Predicts a word based on the surrounding context words.

(eg) Dhoni (is) a legend.

uses Dhoni, a, legend to predict "is".

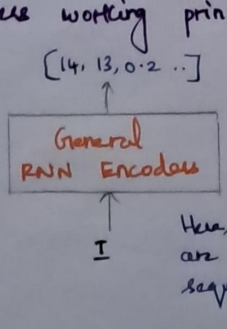
→ **Skip Gram**: Given a word, predicts the surrounding context words.

(eg) Dhoni is a legend.

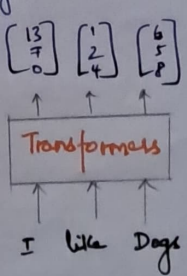
use "is" to predict the other words.

* In real-world use case we use 1000's of Activation Functions to map a certain word to 1000's of Embeddings.

→ **Transformers working principle**:



Here, the words are passed sequentially.



Vectors are created for all the words parallely.