# **DL**

Overlapping content:

Types of the NN / ANN:

1. **FNN (Feed forward neural network) Multilayer perceptron (MLP):**
2. **CNN**
3. **RNN**
4. **GAN**
5. **AUTO-ENCODERS**
6. **TRANSFORMERS**

Intuition of NN works,

Perceptron takes the multiple inputs, each input is taken in certain proportion, decided by the weights (Trained by the backpropagation). Multiple perceptron like this a layer and each layer has many of this, each neurons takes all the inputs, (for first layer these are real inputs), onwards the inputs are outputs of the previous layer in similar manner of proportionate decided by weights. This structure helps understand the complex, non-linear patterns. (better this intuitive understanding).

* **Weights** determine the influence or importance of each input feature on the output. By adjusting weights, the perceptron learns which features are most relevant for classification, allowing it to orient the decision boundary (the separating line or hyperplane) in the input space .
* **Bias** allows the perceptron to shift the decision boundary away from the origin. Without a bias, the perceptron could only create boundaries that pass through the origin, limiting the kinds of patterns it can learn. The bias enables the model to fit data where the optimal separation does not go through the origin, increasing its representational power and flexibility

**What is the intuition of the perceptron trick?**

The perceptron trick starts by randomly assigning some values to the weights and biases. The algorithm then goes through each object and checks if it is classified correctly based on the current weights and biases... If it is misclassified, it adjusts the weights and biases in a way that moves the decision boundary closer to the misclassified object.

Multiple loss function can be used in the perceptron, though traditionally uses perceptron loss for binary classification problem.

**Perceptron Loss vs. Hinge Loss**

Perceptron Loss:

Penalizes only misclassified points.

No margin: correctly classified points have zero loss, no matter how close they are to the decision boundary.

Hinge Loss:

Penalizes both misclassified points and correctly classified points that are close to the boundary (within a margin).

The margin encourages confident, well-separated classifications.

Key Difference:  
Hinge loss adds a margin, making the model prefer not just to correct, but also confident classifications. Perceptron loss only cares about whether points are classified correctly, not how confidently.

Main drawback of the perceptron is inability to work on the non-linear data (XOR).

What is XOR function? when both the inputs are same return 0 else return 1.

**Multiple layers vs number of neurons?**

Having more neurons in a single layer allows a neural network to learn many features at one level of abstraction, but it struggles to efficiently represent complex, hierarchical patterns and may require an impractically large number of neurons for difficult tasks. In contrast, using multiple layers with fewer neurons each enables the network to build hierarchical representations, efficiently combining simpler features into more complex ones, which leads to better generalization and performance on real-world problems. Thus, while both approaches can theoretically approximate complex functions, multilayer networks are generally more effective and practical for most deep learning applications.

In some cases, especially in regression tasks, the same formula—like **RMSE—can be used both as the loss function during training and as the evaluation metric after training**. This means the model is optimized to minimize RMSE, and performance is also reported using RMSE. However, in most other cases (such as classification), the loss function and evaluation metric are different: the loss guides model training, while the metric (like accuracy) simply measures performance. So, while they can be the same in specific scenarios, they usually serve separate purposes.

MSE Advantages:

* **Highly sensitive to outliers:** Outliers can disproportionately increase the loss and skew model training.
* **Large errors penalized more:** Squaring errors means big mistakes have a much greater impact than small ones.
* **Units are squared:** The loss is in squared units, making interpretation less intuitive.
* **No error direction insight:** Only measures error magnitude, not whether predictions are too high or too low.
* **Assumes normal error distribution:** May not reflect performance well if errors are not normally distributed.
* **Can encourage overfitting:** Model may overfit to outliers due to their large influence on the loss.
* **Scale-dependent:** Not directly comparable across datasets with different scales.

**MSE disadvantages:**

* **Highly sensitive to outliers:** Large errors are penalized much more than small ones, which can skew training.
* **Units are squared:** The loss is in squared units, making interpretation less intuitive.
* **No error direction insight:** Only measures error magnitude, not whether predictions are too high or low.
* **Assumes normal error distribution:** May not reflect performance well if errors are not normally distributed.
* **Can encourage overfitting:** Outliers can dominate the loss, leading the model to fit them excessively.
* **Scale-dependent:** Not directly comparable across datasets with different scales.

**Advantages of MAE:**

* **Simple and intuitive:** Easy to understand and interpret, with error in the same units as the data.
* **Robust to outliers:** Less sensitive to outliers compared to MSE, so extreme values do not overly influence the metric.
* **Treats all errors equally:** Every error, large or small, is weighted the same, providing a neutral measure of model performance.

**Disadvantages of MAE:**

* **Not differentiable at zero:** The absolute value function is not differentiable at zero, which can complicate optimization for some algorithms.
* **All errors treated equally:** Does not penalize large errors more than small ones, which may be undesirable if large errors are especially problematic.
* **Less sensitive to error magnitude:** May not be ideal when it is important to prioritize reducing larger errors.

**Huber loss?**

**Above delta MAE, below delta MSE.**

The subtraction and multiplication by delta in the Huber loss formula ensure the loss function is smooth and continuous at the transition point between the quadratic (MSE-like) and linear (MAE-like) regions, so the penalty changes seamlessly and the optimization remains stable

Mean Squared Error (MSE) and Mean Absolute Error (MAE) are both popular metrics for evaluating regression models, but they differ in how they treat errors: MSE squares the differences between predicted and actual values, which means it penalizes larger errors much more heavily and makes the model focus on reducing big mistakes, while also providing a smooth, differentiable surface for optimization. In contrast, MAE takes the average of the absolute differences, treating all errors equally regardless of size, making it less sensitive to outliers and ensuring that small and large errors are prioritized at the same level. As a result, MSE is useful when you want to discourage large errors and can tolerate sensitivity to outliers, whereas MAE is preferred when you want a robust, straightforward measure of average error that is not disproportionately influenced by outliers.

MSE is valuable because it prioritizes reducing large errors, but its sensitivity means that outliers can have a disproportionately large impact on model training and evaluation. Therefore, **when using MSE, it is very important to perform outlier detection and data cleaning** beforehand. This ensures that the model’s performance is not dominated or distorted by a few extreme values, allowing MSE to serve its purpose of minimizing significant prediction errors without being misled by anomalies or noise in the data

If there are a few points with huge errors that are *not* outliers but genuine, important data points, then using MSE is justified and even desirable. MSE is designed to heavily penalize large errors, so it will push the model to fit these high-error points more closely, ensuring that significant deviations are minimized. This is especially important in applications where large mistakes on genuine data are costly or critical to avoid. The key issue with MSE arises only when those high-error points are outliers or noise; in that case, they can distort the model unnecessarily.  But if the points with large errors are valid and meaningful, MSE’s sensitivity helps ensure the model pays extra attention to them, which is often exactly what you want

Till Gradient descent in Notebook Notes,

Backpropagation?

What is chain rule?

The chain rule is a fundamental concept in calculus that allows us to compute the derivative of composite functions—functions made by applying one function to the result of another, like f(g(x)) *f*(*g*(*x*)). Intuitively, it tells us how a change in the innermost variable (like x*x*) propagates through each layer of functions to affect the final output. You can visualize this process with the analogy of interconnected gears: imagine x*x* turning the first gear, which in turn rotates a second gear u=g(x)*u*=*g*(*x*), and that gear then drives a third gear y=f(u)*y*=*f*(*u*). The overall effect of turning the first gear on the last is the product of the effects at each step—just as the chain rule multiplies the derivatives at each stage. In essence, the chain rule lets us break down and “chain” together the rates of change through each function, so for y=f(g(x)) *y*=*f*(*g*(*x*)), the derivative is f′(g(x)) ⋅g′(x)*f*′(*g*(*x*)) ⋅*g*′(*x*), capturing how changes ripple through the composition.

let's suppose from one parameter let's say one weight if there 100 different paths which you came across then in that case you, calculate these 100 derivatives sum them multiply the sum with learning rate and subtract from the original values.

**Unit balance in Gradient descent formula?**

**And why not normal maximization principle?**

Forward Propagation:

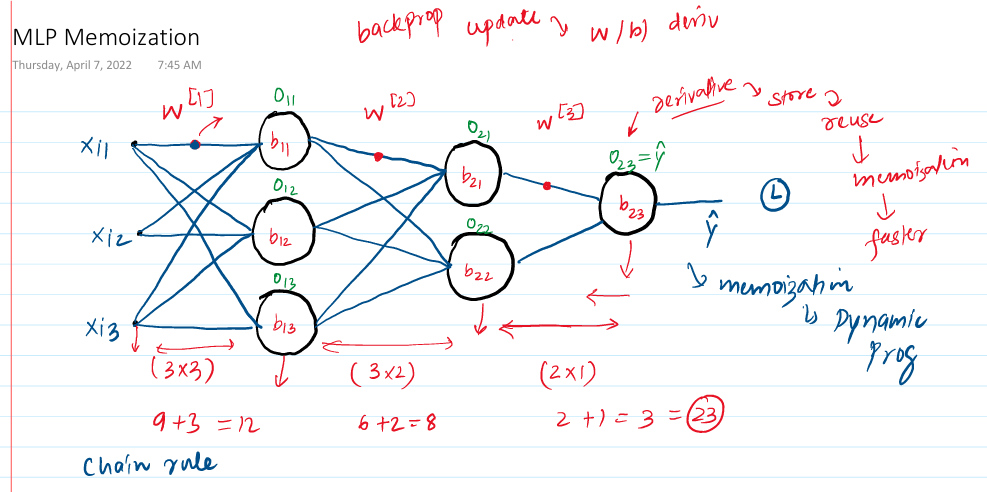
Weights matrix in forward propagation is not the transpose matrix but the normal weight matrix with the shape as (current layer nodes, previous layer nodes). And model.get\_weights() return list with arrays as elements with each element.

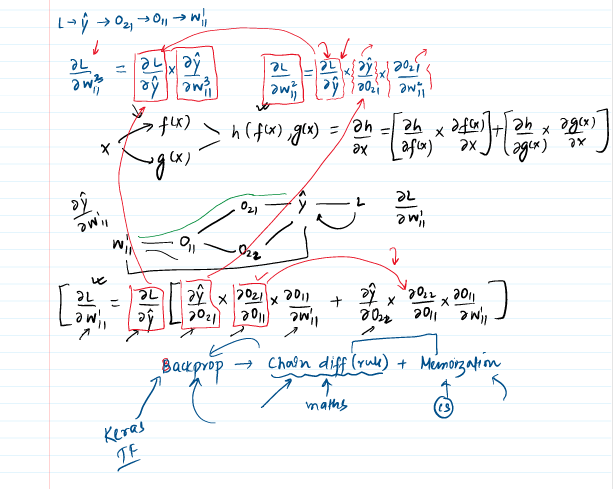
Loss functions also in notes and mostly same as ML.

**Memoization:**

**A technique in which we use storage to reduce the computation. i.e. a repetitive calculation is stored and called on instead of calculating again, which makes the computation fast on cost of storage.**

**Memorization along with chain rule is the backpropagation.**

****

****

**Gradient Descent same as ML**

**Vanishing Gradient in notes and Colab.**

# Vanishing Gradient

* When the partial derivative of gradient is less than 1, particularly for deep neural network in that case the multiplication of those gradient becomes very low value as those gradients are interdependent for deep layers and calculated using chain rule. i.e. 0.1 \* 0.1 \* o.1 turns out to be very low value and eventually gets low gradient value which ultimately gets very low change in new weights and basically no updatation happens. And that is what refer as gradient is vanishing.
* Happens particularly for sigmoid and tanh activation functions.

## How to recognize

* Loss function
* on changes: while training an algorithm if don't see changes or decrement in the loss function as the training happens then it means there is vanishing gradient as weights are not getting updated.
* value of weights itself, can be checked during the training and if change is constant then that is slam dunk case of the vanishing gradient.

## **Reduce Vanishing Gradient**

1. Shallow Neural Network

2. Changing activation function

3. Proper weight initialization (Xavier or He initialization)

4. Batch normalization.

5. Residual Network (CNN-Resnet).

**Techniques to improve the Neural Network. (notes ss)**

1. **Hyperparameter Tuning:**
   1. **No. of hidden layer:** The no. of hidden layers significantly decides how complex patterns are derived from the data. It is preferable to have higher no. of hidden layers compared to the more neurons in the lower no. of hidden layers (two ways to extract complexities either increase the higher no. of neurons or increase the no. of hidden layers). Cause the having more no. of hidden layers helps in a way where initial layers prick the more outline structure of the data, and the deeper layers extract the more intricate and detailed aspect of the data.

The same concept comes very handy in transfer learning, where pre-trained model on generalized data is used after fine tuning on the specific data which extract the fine/specific patterns of the data

* 1. **No. of Neurons:** Input and output nodes are fixed depending on the input and output whereas hidden layer’s nodes are initially supposed to be following the pyramid structure, but that is not the mandatory. No. of neurons are supposed to be sufficient, and you stop adding more neurons when model starts overfitting.
  2. **Batch size:** Two approaches, first one is small batch size more computation with good accuracy always and second one larger batch size with techniques to improve accuracy, but it takes smaller computation and time.
* For smaller batch size it usually taken as 8-32, will take more time but will be accurate, should be used if the other one does not give good accuracy.
* Larger batch (8192) which does not give you high accuracy directly but computationally efficient and accuracy can be increased with a technique called as warming up of learning rate where the learning rate is increased towards the later epochs and initially kept low. Learning rate scheduler is needed.
* **Learning rate warmup** is a training technique where the learning rate starts small and gradually increases to its target value over the first few steps or epochs. This helps stabilize training by preventing large, erratic updates early on, especially in deep or complex models, and leads to more reliable and effective learning.
  1. **Epochs:** You start with any high epoch value but make sure to use early stopping with callback which stops the training after an epoch where there is no significant improvement in model accuracy.
  2. **Learning rate: During slow training**
  3. **Optimizer.**
  4. **Activation function: During vanishing gradient.**

1. **Solving problems:** 
   1. **Vanishing / Exploding problems.**

* **Weight initialization:**
* **Activation function**
* **Gradient normalization**
* **Clipping**
  1. **Not enough data:**
* **Transfer Learning**
* **Unsupervised pre-training**
  1. **Slow training:**
* **Optimizer**
* **Learning rate scheduler**
  1. **Overfitting:**
* **L1 and L2 regularization**
* **Drop out lay**
  1. **Not enough data.**
  2. **Slow training.**
  3. **Overfitting.**

**Early Stopping:** (Colab).

**Feature scaling**: Like ML.

Drop Out Layers. (Screen shot).

To improve the overfitting of the neural networks we use different techniques like,

1. More Data.
2. Reduce complexity. (reducing the no. of parameters).
3. Early stopping.
4. Regularization.
5. Drop out.

* **Dropouts:** reduces the no. of nodes which intern reduces the overfitting because it reduces the connection and complexity and parameters. And secondly, by drop outing nodes randomly for each epoch NN does over rely on single pattern or does not give more importance to few certain weights only, rather it gets more generalized with no special importance given to any weight/pattern.
* During training with dropout, a random subset of neurons is temporarily deactivated for each batch, meaning their outputs are set to zero and their weights are not updated during that iteration—they remain unchanged until the neuron is active again in a future batch. This random deactivation prevents the network from relying too heavily on specific neurons, encouraging it to learn more robust and generalized features, which helps reduce overfitting. During inference, all neurons are active, and no dropout is applied, ensuring the full network contributes to predictions.
* Technique is you usually randomly disconnects the specific no. of neurons each epoch from each layer which improves the accuracy up to 2%, statistically seen.
* How the drop out layer in NN are like the random forest. Like random forest ensemble learning where multiple DT are aggregated to predict similar input just like that multiple NN are trained/created each epoch with different node combinations and at the end sort of aggregation is done while testing/ predicting by taking the (1-p) portion of the weight value. i.e. (1-p) probability of that weight being available for training for all the epochs.
* P = drop out probability/Ratio.
* The **dropout rate is typically specified per layer, not for the entire neural network as a whole**
* During prediction, dropout is not applied—all neurons are active, and the weights remain unchanged. To ensure consistency with training, the outputs of each neuron are scaled by the keep probability (e.g., 0.8 if the dropout rate was 0.2). This adjustment matches the expected output levels seen during training, ensuring reliable predictions without altering the learned weights.
* During training with dropout in deep learning, a random subset of neurons is deactivated (their outputs set to zero) in each batch, which means those neurons do not participate in the forward or backward pass for that batch and their associated weights are not updated. This randomness prevents the network from relying too much on specific neurons, promoting redundancy and helping the model generalize better to unseen data by reducing overfitting. During testing or inference, dropout is turned off—all neurons are active—so to maintain consistency with the training phase, the outputs of each neuron are scaled by multiplying with (1-p), where p is the dropout rate. This scaling ensures that the average activation values during testing match those during training, keeping the model’s predictions stable and reliable. Throughout this process, the weights themselves are only updated during training when their corresponding neurons are active; during testing, weights remain fixed and only the neuron outputs are adjusted.

Practical Tips and tricks:

1. Overfitting (increased p), underfitting (decrease p).

2. Start with application of dropouts after the last layer. (Last layer extract more intricate and details patterns and causes overfitting so to reduce that overfitting).

3. CNN- 40 - 50% P.

4. RNN – 20-30% P.

5. ANN – 10-50% P.

Drawbacks of dropdown layers:

1. Delayed convergence. As we sort of use less nodes each epoch it results into delayed convergence of the weights.
2. Notes.

**Regularization:**

* With increase in the no. of neurons model ability of capture the complex relationship increases but at the same time it’s tendency to overfitting increases as well.
* To reduce the overfitting, the effect of the nodes which are causing the overfitting is reduced by diminishing their magnitude value, which makes sure that it helps in capturing the complex average pattern with those diminishing values but does not capture the details of those complex deep patterns in details as we reduce the values of those weights.

Ways to solve overfitting:

1. More data.
2. Data Augmentation (mainly in CNN, image data).
3. Drop out layers.
4. Early Stopping.
5. Regularization.

There are their ways of regularization:

1. L1:

Add SS from Note,

Basically, in L1, a sum of all the weights is added in the loss function and alpha proportionate value of it taken after dividing it with no. of rows.

Means, you add some proportionate sum of the weight which increase the loss function and ultimately reduces the weights can also make them zero in some cases when first and second term at numerator becomes equal in magnitude.

Penalty term is L1 norm. Sparse model in L1 norm.

1. L2: Mostly used and same as L1 Except instead of normal sum, sum of the square of the weights is done. Which tends to make the weights go towards zero.

Intuition of how increase the loss makes the weight zero, from notes, but main intuition is backpropagation when we you increase the L without much change in “w” i.e. denominator the minus term becomes bigger and new “w” becomes smaller. **Weight decay.**

Penalty term is L2 norm. No sparse model in L2 norm.

1. L1 + L2:

**Activation Functions:**

Activation functions add the ability to capture the non-linear relationship in the data.

Proof from notes which is the final equation of the output comes after which does not have any degree terms involved if linear activation is used which makes it impossible to capture the non-linear relationships in the data.

How human mind processes the images?

* Light from retina goes through optical nerves (cells) to the thalamus and from there to the V1. Electrochemical single is the method through which light moves thorough the nerves.
* Main takeaway is different cells process different aspects.

# **CNN:**

Convolutional Neural network, also known as convnet, or CNNs are a special kind of neural network for processing data that has a known grid-like topology like time series data(1D) or image(2D).

Why not use ANN on Image Data: let’s see what happens when you use ANN on image data:

1. **High computational cost:**

The first and obvious thing you must do while using the ANN on image data is, you must convert the horizontal pixels into vertical inputs and stack each horizontal layer of pixel vertically and ultimately give that many number of the inputs to the model, which makes very high no. of inputs to the models and that many connections will be made. Let’s suppose you have image of 1000 X 1000 then (1000000) inputs will be given to the model which makes computation so heavy.

1. **Overfitting**: As there are so many no. of connections and parameters there is chance of high overfitting in the model.
2. **Loss of important spatial arrangements of the pixels**: e.g. from the image, where the 2D essence of the data is lost. i.e. when you convert a 2D data into one 1D it loses the 2D meaning representation. (e.g. dog eyes and nose distance will be lost now in one 1D as they are sequential now earlier there was concept of distance between eyes, nose and lot when those pixels were in 2D)

Image

Applications:

1. Image classification on labeled data.
2. Object detection: Detecting objects in the image.
3. Facial recognition: recognized unique facial features and hence unique face.
4. Semantic Segmentation. Pixel level object detection. i.e. each pixel is labeled as particular object.
5. And many more.

Resemblance with Human brain image recognition:

CNN are inspired by human brain and has similar image processing mechanism.

Initial lower-level features detect the simpler outer features of the images like the human brain cell does, for different image features different brain cell gets activated meaning different brain cell detects the different image feature. Human brain has simple and complex brain cells, simple brain cells detect the simpler outer features whereas the complex brain cells get triggered and detects the more complex patterns.

In human brain has simple cells, complex cells use for image detection, where the simple cells have lower receptive field and are used to detect simpler and initial basic patterns like edges, orientation etc. and i.e. these cells are called orientation cells or feature detectors. Or preferred stimuli.

Whereas the complex cell with bigger receptive field is used to detect the complex patterns in the data.

**Intuition** of the CNN: Exact similar thing happens in the CNN as well the simpler features are detected by initial filters and as layer by layer you move with different filtering you sort of add those outputs with more complex features detected by the later filters and this is how CNN recognizes the entire image pattern.

Architecture is as such where you have Image arrays as input and then you apply filters on it and do the convolutional process. (with padding, strides or without them). And then add pooling to address few issues raised such as

* Reduced array shape.
* More impact to the centered pixels.
* Overfitting.
* Computational efficiency.
* Memory issue.
* And translational invariance.

Channel is the depth layer of the color image.

How does the convolutional operation happen?

* 2D: A black and white image is applied with the filter and gets you reduced 2D array with cross multiplying each corresponding element and then adding all the products (dot product).
* For 3D you do the same but with 3D filter the output you get is 2D array only after every entire operation. Cause the 3D convolutional operation happens with the filter having the same depth as your input and you don’t stride/move your filter backwards, but you only move them in 2D and i.e. you get 2D array output as a **FEATURE MAP.** Size of the feature map is always a **2D array X no. of filters.**
* **Feature Map:** One feature Map after one filter operation. i.e. you combined those feature maps (obviously as many feature maps as no. of filters into one nD array with n being no of filter for one convolutional layer.
* Yes, the feature map produced by one convolutional layer becomes the input to the next convolutional operation in a Convolutional Neural Network (CNN). Each convolutional layer applies its filters to the output (feature maps) of the previous layer, allowing the network to learn increasingly complex and abstract features at deeper layers. This process enables the network to build hierarchical representations, starting from simple patterns like edges in early layers to more complex structures in later layers.
* The real working intuition of the CNNs is developing those filters or rather finding out the right filter to find the right pattern in the image which is simple features/patters which is the case in the earlier layers and deeper convolutional layers has the filter to find out the deeper complex patterns/features. The basic intuition is having the right filter like for simple task like vertical edge detection the filters are simple and known, and in CNN the things is you need to find the right filters to find the same patterns which are specific to our data.
* **Here’s the summary of what happens in CNN.**
* So, the input to the CNN could be 2D or 3D array given dimensions, and this 2D or 3D decides the depth of the filters in convolutional layers, for 3D input the filter depth is 3D and for the likewise for the 2D input. This extra depth is called as channel.
* But each filter operation produces the output known as feature map or channel which is always 2D, the number of channels is equal to the number of filters, now we have “N” number of channels or feature maps which are equal to the number of filters. Pooling is applied on this feature maps for obvious reasons. And then these features maps are combinedly treated as the input to the next convolutional layer with shape as 2D X channel (No of feature maps / number of filters) and obviously the shape 2D is dependent on the filter shape, strides, padding etc. this is the core of the CNN. After one convolutional operation, activation function (relu) is applied which add little bit of linearity and then the pooling is done. And the filter shape for the next convolutional layer is decides by the output of the this pooling layer.
* **Images:**

**And notes for the 3r video as well.**

Trainable parameters: What you calculate weights and biases rights. (Keep the same principle here as well):

What could be the trainable parameter?

Feature maps does look like a trainable parameter, but it is not, is it? Those arrays are something like the output the ANN. Not something but kind of exactly like the output of the ANN.

And then what is like the weights?

* Filter are neurons.
* Filters are what we want to create and developed for a particular image because those are the arrays or rather values which are getting applied on our input and producing output right. So, we must decide those values and those exact values are nothing but the comparatives of the weights.
* And the biases of each filter

So, trainable parameters are size of the (filter array \* No. of. Filters) + biases.

Same for the ANN is, (No. of input \* No. of neurons) + no. or neurons

Main thing about the CNN that trainable parameters do not depends upon input parameters at all which was the case in ANN that was creating the high computational complexities and memorization error.

**Padding:**

Padding in Convolutional Neural Networks (CNNs) is the technique of adding extra pixels—typically zeros—around the edges of an input image or feature map before applying convolutional or pooling operations. This process is crucial for preserving spatial information at the borders, preventing data loss that can occur when filters overlap only the central regions and ignore the edges. Padding helps maintain consistent output dimensions, especially when using larger filters or when it is necessary for the output size to match the input size (as with "same" padding), and it ensures that features near the image boundaries are not neglected during feature extraction. There are several types of padding: "same" padding adds enough pixels to keep the output size equal to the input, "valid" padding uses no extra pixels and results in smaller outputs, and "causal" padding (used in sequence models) adds padding only to one side to preserve temporal order. Choosing the appropriate padding strategy directly impacts the effectiveness, stability, and flexibility of CNN models in tasks such as image classification, object detection, and segmentation

**Strides:**

Stride in Convolutional Neural Networks (CNNs) is a fundamental hyperparameter that determines how many pixels the convolutional filter moves at each step as it slides across the input image or feature map, directly influencing the spatial size of the output feature map and the computational efficiency of the network. A larger stride reduces the output dimensions and computational load by skipping more pixels, which is particularly useful when the goal is to extract high-level, global features and when fine-grained details are less important. This down sampling effect helps the network focus on broader patterns, making strides especially valuable in scenarios where efficiency and abstraction are prioritized over detailed feature extraction. However, increasing the stride also risks losing important information, so the choice of stride should be carefully balanced based on the specific task and dataset requirements. In summary, strides are primarily used to control output size, computational cost, and the level of feature abstraction, playing a crucial role in the design and performance of CNNs. Not used mainly these days.

**Stride of 2 skips the single row/column whereas the stride of 3 skips 2 columns / rows.**

**How to calculate the shape after strides and padding**

**((N+2P – f) / s ) + 1 , floor in case of decimal.**

For each feature map,

**Pooling:**

* **Pooling solves following problems.** 
  1. **Memory Issue (dimensionality reduction):** Pooling down samples the width and height of feature maps, which decreases the amount of data and computation required as the network grows deeper. This helps manage memory and processing demands, especially for large images or deep networks.
  2. **Translation Invariance** (Only for minor translation chances): By pooling we can remove the location dependency of the extracted features. E.g. would features extracted of cat image like nose, ears, eyes are not specific the location of these features or location of the cate itself. Feature (eye, ear etc.) is a feature irrespective of the location of the cat.

Pooling makes the network less sensitive to small shifts or distortions in the input, so features detected in slightly different locations are still recognized

**3. Overfitting prevention**: By reducing the spatial dimensions, pooling layers help prevent overfitting by providing a form of regularization.

**4. Feature Hierarchy:** Pooling layers help build a hierarchical representation of features, where lower layers capture fine details and higher layers capture more abstract and global features.

228 X 228 X 3 is the usual input in standard CNN architecture.

Types of the pooling: In keras we have only 3, Max, min and global

1. **Max pooling:** Selects the maximum value from each region of the feature map. It is the most widely used pooling method because it retains the most prominent features and provides strong translation invariance. Use max pooling when you want to focus on the most important activations, such as in image classification and object detection tasks.
2. **Min pooling:** Min pooling is a pooling technique that down samples feature maps by selecting the minimum value within each defined region (e.g., a 2×2 window) of the input. Like max pooling, it reduces the spatial dimensions while preserving some information, but instead of capturing the strongest activations, it focuses on the least prominent or lowest values in each region. This can be useful for highlighting background or less active features. However, min pooling is rarely used in practice, especially with ReLU activations, because it often results in many zero or very low activations, which can hinder learning and reduce the effectiveness of the network. It may find limited application in specialized tasks where capturing minimal or background features is important, but for most vision tasks, max pooling or average pooling are preferred due to their better performance and informativeness.
3. **Avg pooling:** Average pooling reduces the size of feature maps by taking the mean of values in each region, resulting in smoother, more generalized features. It’s useful when you want to preserve overall information rather than just the strongest activations. L2 pooling.
4. **Global pooling:** Global pooling is a technique in CNNs that reduces each feature map to a single value by applying an aggregation function—such as averaging (global average pooling) or taking the maximum (global max pooling)—across the entire feature map, and is typically used right before the output or dense layer to minimize parameters, prevent overfitting, and make the model robust to varying input sizes. logic of max global and avg global is same as only max and only avg.
5. **L2 pooling:** L2 pooling is a pooling technique in convolutional neural networks where, for each region of the feature map, the Euclidean norm (L2 norm) is calculated by taking the square root of the sum of the squares of all values in that region. This method is less common than max or average pooling but can be useful in cases where a different form of regularization or feature aggregation is needed. L2 pooling provides a balance between capturing strong activations and preserving overall energy in the region, but it is more computationally expensive than max or average pooling and is typically used in specialized applications rather than standard image classification tasks.

How to do it?

* Pooling is done on individual feature map/channel.
* Size (2,2) with strides of 2 is mostly used for aggregation.

What pooling creates feature invariance?

Pooling provides translation invariance in CNNs by summarizing the most important features within small regions of a feature map called as **receptive field,** so that small shifts in the position of features do not significantly affect the output. For example, if an object in an image moves slightly to the left or right, max pooling will still capture the strongest activation within each region, regardless of its exact position. This means that even if a feature like an edge or a corner shift within a pooling window, the pooled output remains largely unchanged, allowing the network to recognize the object in different positions. As a result, pooling helps CNNs focus on the presence of features rather than their precise locations, making the network robust to minor translations in the input image.

Advantages of using pooling?

* + Down sampling (reduced size).
  + Transitional invariance.
  + Enhanced features in case of the max pooling.
  + No training needed.

**Why pooling is better than only strides in CNN?**

Pooling is often considered better than using only strides for down sampling in CNNs because pooling provides translation invariance by focusing on the presence of features within regions rather than their exact positions, is computationally cheaper and parameter-free compared to learnable convolutions, simplifies network design by decoupling feature extraction from down sampling, and can improve gradient flow in deep networks, making models more robust and easier to train.

Architectures:

**LeNet-5** processes a 32×32 grayscale image through a sequence of layers: the first convolutional layer outputs 28×28×6 feature maps, followed by average pooling to 14×14×6; the second convolutional layer produces 10×10×16 feature maps, then average pooling reduces this to 5×5×16; a third convolutional layer outputs 1×1×120, which is flattened and passed through two fully connected layers with 84 and finally 10 units for classification, efficiently extracting and condensing features at each stage for robust image recognition.

**CNN vs ANN**

Similarities in CNN, filter are equivalent to the neurons of the ANN with each filter element are trainable parameters like the weight of the ANN, with each filter having one bias just like the neuron. Just like what happens in neuron the weights are multiplied with the weights of the input and then bias is added to the output of the neuron similarly, after a convolutional operation of the filter on entire input (neuron is connected to entire input) the resultant feature map is added by the bias of the filter. And the pooling layer is applied, and the operation is continued.

This is very analogous to what we do in ANN.

Now the difference is.

* **Computational cost:** In CNN the trainable parameter is the total filter parameters + their biases, it not at all depends on the input shape. Which is not at all the case it the ANN where the trainable parameters are dependent on the input which increases rapidly with increase in the input though the architecture could be similar.
* **Overfitting:** With the increase in the number of weights to capture this complicated image patterns the overfitting increases.
* **Loss of important feature like spatial arrangement of the feature:** Due to filter sliding the spatial correlation between the features is captured.

Make sure to complete the backpropagation of the CNN’s in details later.

**Pretrained model in CNN:**

**WHY?**

* Normal DL models for supervised classification problem needs labelled data in huge quantity for DL.
* Training time is huge as well.

That is why using already pretrained model is handy.

ImageNet dataset: visual database for image with 14 million images of 20000 different classes. These images are labelled in detail with hierarchal details. Around 1 million data is bounding box which is useful for object localization.

Ho: crowd sourcing by amazon service called as mechanical turck.

**ImageNet** competition (ILSVRC): started in 2010 used subset of the original ImageNet dataset with 1 million images with 1000 classes.

Initially the ML models got the better accuracies with 28%, 25% error rates then comes the AlexNet in 2012 which is the landmark year for DL (image data) by Jeoffrey Hilton which produces the error rate of the 16%(used relu in CNN).

Over the year different model came which are like this:

* ML (2010) 🡪 28 %
* ML (2011) 🡪 25%
* AlexNet (2012) 🡪 16.7%
* ZFNet (2013) 🡪 11.7%
* VGG (2014) 🡪 7.3%
* GoogleNet 🡪 6.7%
* ResNet 🡪 3.5%

You can simply get ResNet or any other model and make it predict any image which you want to.

**What does CNN sees?**

See the video first.

**Transfer learning?**

Why do the transfer learning reason is above which is same as the pretrained model’s reason.

Infect transfer learning is a method of using the already trained model on your problem.

**Definition:** Transfer learning is a research problem in machine learning that focuses on storing the knowledge gained while solving on problem and applying it to a different but related problem.

Using one domains knowledge to another. E.g. fundamental skills which are like many tasks so use those skills which are learned while learning something different but similar for your current task. Just like learning to ride bicycle and then ride bike. Underlying skillset is similar, so the same basic underlying skills are used for riding bike.

How do we do it?

So, any image model is comprised of Conv2D layers and FC layers, where the convolutional layers are used to extract the spatial features or mainly the initial layers which are mainly CONV2D are used to extract the initial features underlying features. The purpose of the dense layers is to classify the image.

So ultimately you freeze the conv2d part and train your own FC layers. Continued the rest part?

As they say don’t reinvent the wheel, build the car out of it.

Ways of doing transfer Learning?

**Feature Extraction:** In this method, you use a pre-trained model as a fixed feature extractor. You freeze all the layers of the pre-trained model so their weights do not change, and you only train new layers that you add on top for your specific task. This approach is efficient, requires less data, and is less computationally intensive, but it may be less adaptable if your new task is very different from the original one

**Fine-Tuning**: This method goes a step further. After initially training the new layers (as in feature extraction), you unfreeze some of the pre-trained model’s layers (often the deeper ones) and continue training the whole model on your new dataset. This allows the model to adjust its previously learned features to better fit your specific task, potentially improving performance, especially if you have a larger or more diverse dataset.

Code in the colab.

**Functional API :**

# **RNN: Recurrent Neural Network**

NN which works on the sequential data and can preserve the sequence of the data. E.g. NLP task, stock prices or any time series data, sound data etc.

**Why do we need RNN’s** only (why can’t using ANN won’t work)?

* Variable input: Variable input needs to be handled by zero padding.
* Zero padding: zero padding creates the unnecessary computations,
* Prediction Problem (Incase the larger length of the input needs to be predicted).
* Totally disregarding the sequence of the sequential info.

**Intuition of the RNNs?**

* There is very much similarity between the RNN and ANN mainly the difference lies in two ways, first one is RNN’s ability to handle the variable size input by use of timestamps and second one is the RNN’s ability to have feedback loops (which gives it the ability to capture the sequential info).
* Nodes in RNNs not only receives the weights and bias as a input but also the input from the previous layers, which gives it a sense of what’s the learning/prediction from the last timestamp, such as every other prediction/ learning of the neuron would have previous learning as a input to it and this way the one input in various timestamp is fed and processed in the RNNs. Output of the last timestamp of the previous input is not fed to the first timestamp of the current input/row in processing, while it receives newly initialized outputs which are then carried/process/updated till all the timestamps are processed for the given row/input.
* In deep technical details the, when one neuron (let’s initial layer neuron) learns from the some aspect (basic info or context about the input) input based of the weights learned by backpropagation it takes into account the last timestamps prediction as well which makes it learns the relationship of the current prediction (some basic/specific aspect which at the end will go add and then we’ll have complete output) with previous output.

How is the input given to the RNN, now as we have seen that we cannot give zero padded input like even if we try ANN.

So input is given to the RNN in timestamp, and for each record (row) the no. of timestamp is as equal to the no. of words in the record.

First, let’s understand how do we encode the data?

Record: My name is BMF.

The ultra strong MF.

No. of unique words = 8

my = [1,0,0,0,0,0,0,0]

name = [0,1,0,0,0,0,0,0]

is = [0,0,1,0,0,0,0,0]

and likewise…….

Now the record would be represented as 2D array of [4 X 8] each row is word of 1 X 8.

And the final input to the RNN would be a 3D tensor of [2 X 5 X 8].

(batch size, sequence length, feature size)

Batch size = N/number of records

Sequence length = max number of words in any of the record.

Feature size = embedding size same for all the words

I plane/row for each entry/input and likewise you have n number of entries for entire dataset.

That one entry looks like from top as max words in input times the embedding size for each word.

i.e. arrangements are as such where one plane for one row and like second horizontal plane for second input.

5 for max no. of word in all the records, that means whenever we have a word less than the max of words in a row there will be an empty row for that record, meaning there will be **padding** in RNNs as well.

This is how the tensor is given as the input to the RNN.

Now let’s see how each input is feed as a timestamp to the RNN.

For row one, there would be 4 timestamp and first would word “My” with its encoding and let’s say we’ve RNN architecture as such as

input timestamp [ for word “My” 8 X 1] followed by 3 RNN nodes and followed by 1 fully connected node.

So, when for the first timestamp the “My’ enters with 8 inputs of [1,0,0,0,0,0,0,0] and connects to the RNN layer nodes.

Extra thing here is the output of the first-time timestamp [01] will be fed back to the nodes.

And then the processed value is sent to the next node.

Before the formulae of forward propagation let’s calculate the no. of trainable parameters, here.

W1 = 8 X 3

W1R = 3 X 3

W2 = 3

Biase1 = 3

Biase2 = 1

Total = 40

Formula is

(W1. X11) + (W1R. O1) + b1

W1 = [3 X 8]. X11= [8 X 1] + W1R= [3 X 3]. [3 X 1] + [3 X 1] = [3 X 1]

One thing for the first RNN layer as we don’t have any of the outputs, we select either zero-zero or random outputs. And this is how we calculate forward propagation.

How do

Architecture Diagram?

In RNN we have hidden recurrent layer which gives back the output of the first input as a second input along with weight for second timestamp.

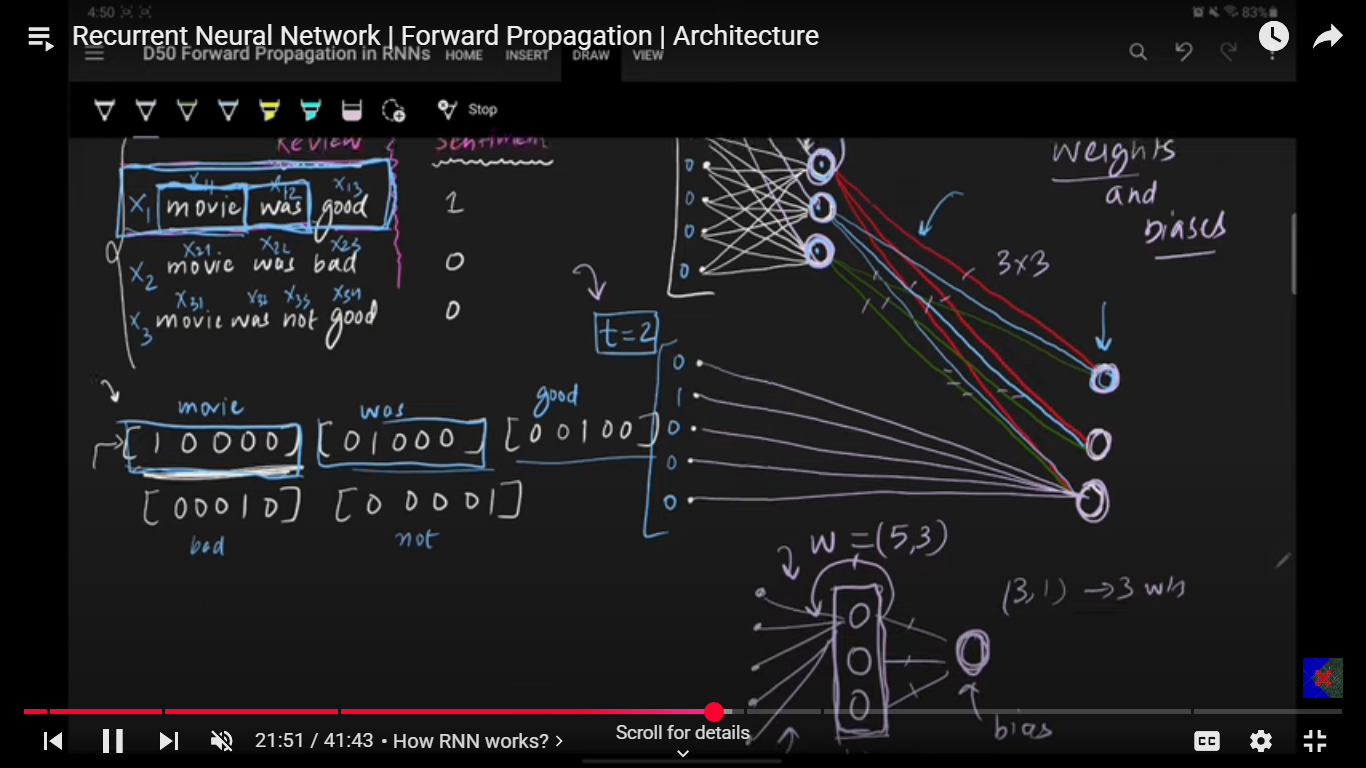
**Unfolding through time in RNNs** means processing the input sequence step-by-step across different timestamps, while **the weights for the entire layer (both for the input-to-hidden and hidden-to-hidden connections) remain the same for every timestamp**. This is called **weight sharing** or **parameter sharing**. Backpropagation is very similar to the ANN, nothing sort of complicated.

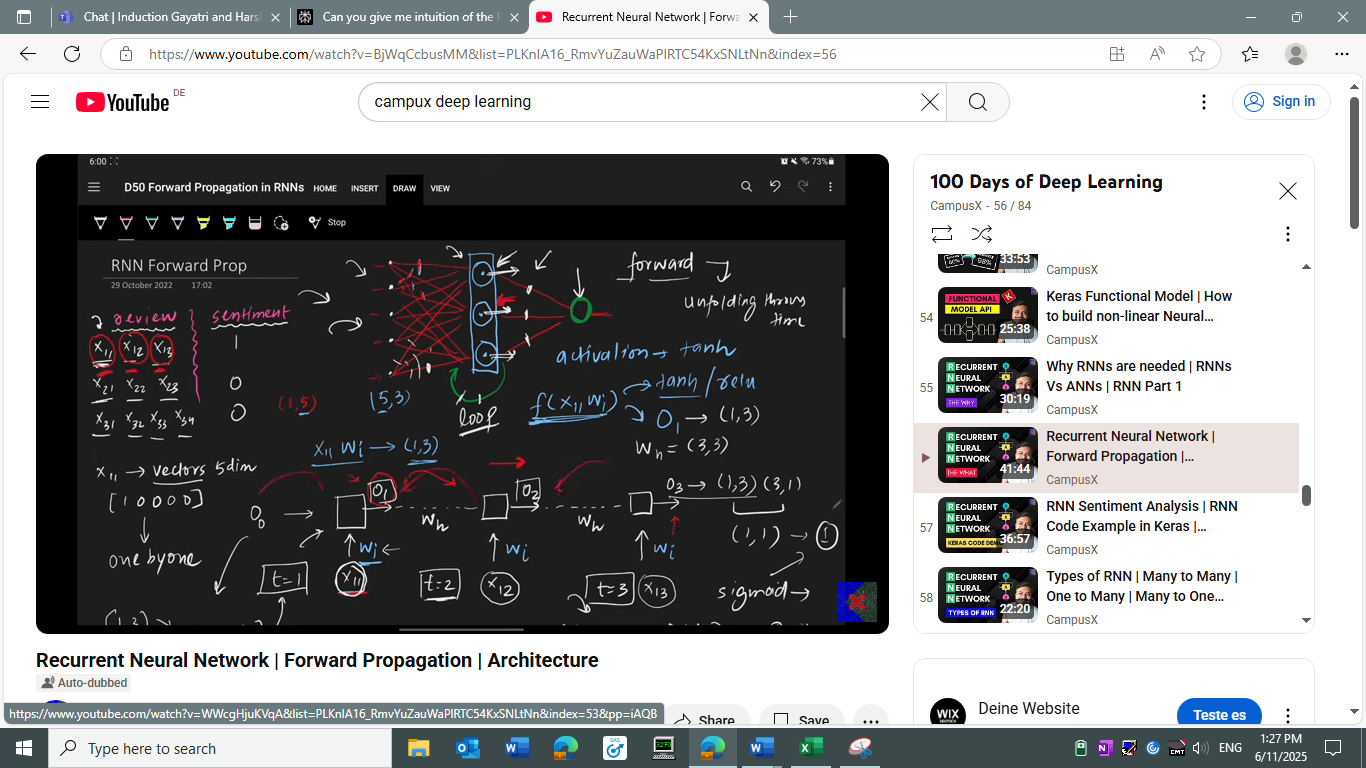
Encoding techniques:

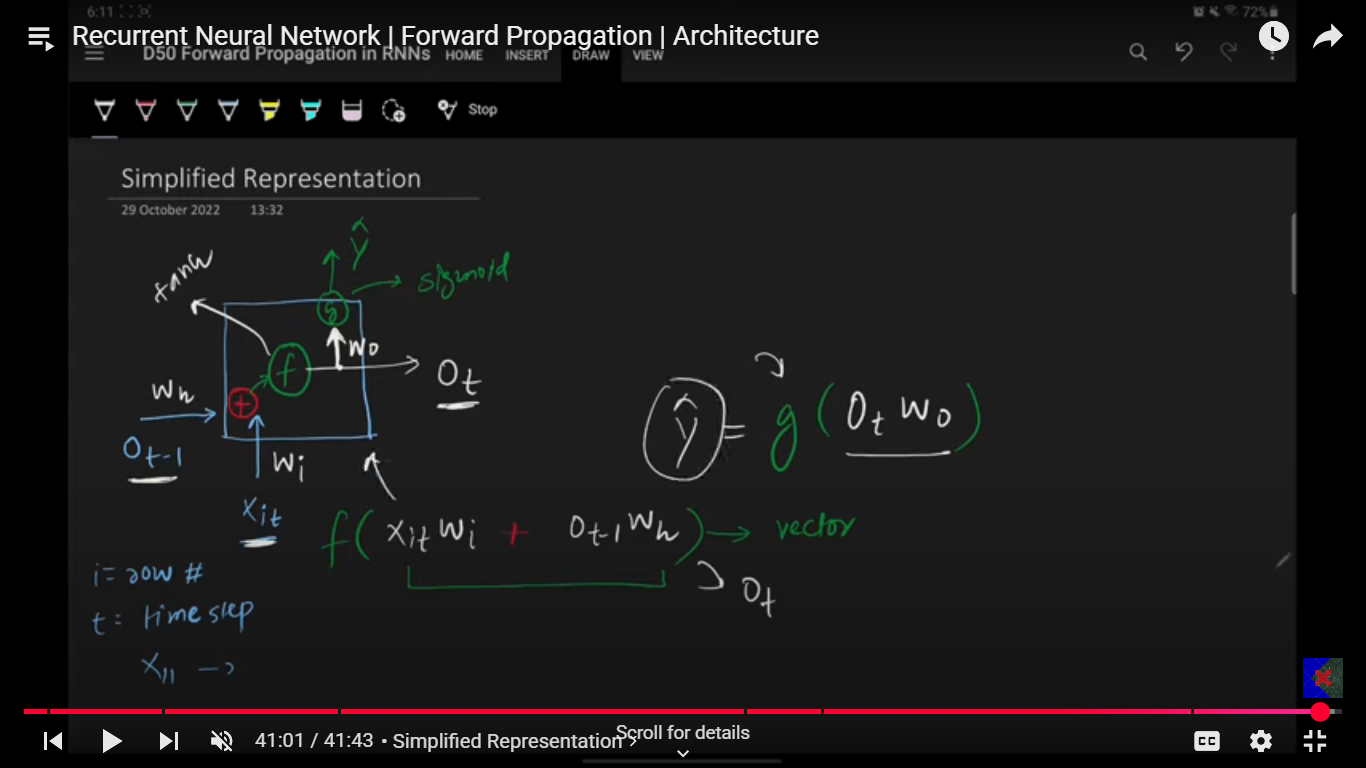
OHE, we represent a word by a vector and sends the 3D tensor as an input to the RNN.

**Recurrent Neural Networks (RNNs)** are designed to process sequences, such as text, by reading one token at a time and updating an internal memory called the hidden state at each step. At every point, the RNN combines the current token with the previous hidden state, gradually building up a context that reflects the entire sequence so far. After the last token, the final hidden state contains a summary of the whole input, which is then used by the network to make a prediction—such as classifying whether a transaction description is fraudulent—based on the accumulated context. This ability to remember and integrate information across a sequence makes RNNs powerful for tasks where order and context matter.

In RNNs, the same weights are used at every time step, and training optimizes these weights so that the final hidden state after processing the entire sequence provides the best possible summary for making accurate predictions. The goal isn’t to perfect the hidden state at each individual step, but rather to ensure that, by the end of the sequence, the accumulated context in the hidden state leads to the most effective overall result for the task at hand.







**Types of RNN:**

1. **Many to one: Many inputs and one output (General one which we have studied till now).**
2. **One to many: Image description.**
3. **Many to many (Sequence to sequence)/ (encoder decoder).**
4. **One to one.**
5. **Many to one**
6. **Many to many**
7. **One to Many**

**Backpropagation in RNNs?**

**Issues in RNNs:**

**There are majorly two issues in RNN’s:**

**Problems are problem of long-term dependencies and problem of stagnated training.**

1. **Vanishing gradient descent. Or inability to capture longer term dependencies.**
2. **Exploding gradient problem.**
3. **Vanishing gradient:** It is the main problem lying under which is inability to capture the long-term dependencies.

* Usually for longer timestamps, when you update the weight by gradient descent those values are interdependent on all the previous timestamp derivative values. In the above scenario if you have if you the derivative values to in between 0-1 and tanh is always in between 0-1 then the gradient comes to be very small and that implies the impact of that timestamp would be very less.
* Solution:
* Activation (relu, leaky relu)
* Bette weight unitization
* LSTM

1. **Exploding gradient:** It is the main problem where the gradient values are way more consistently and therefore those values comes out be very high for later stamps.

* What is the solution:
* 1. LSTM
* Tanh
* Clipping

**LSTM’s Core Idea:**

**LTM, STM and interaction between both.**

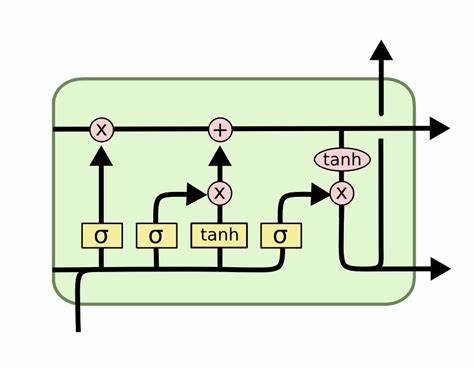
As the main reason that RNN’s cannot keep track of the old context, we must make sure somehow that RNN’s keep track of the longer-term dependencies. That is exactly what is done in LSTM’s. So, what does LSTM’s do is, along with short term dependencies like in normal RNN’s they also keep track of the long-term dependencies. So, in LSTM we have one **LTM** (Longer term memory) which track the long-term meaning/memory/context and **STM** (short term memory) tracks the shorter-term memory.

So now in LSTM instead of one STM recurrent state we have STM as well as LTM inputs feedback to the nodes of the layers.

In LSTM we have three gates and two operations:

Operation 1: Forget things from cell state and add things in cell state.

Operation 2: Calculates the hidden state.



1. Forget gate.
2. Input gate.
3. Output gate.

Cell state is LTM, and hidden state is STM:

So, every state we either forgot something from the LTM or add something to the Cell state (LTM),

Whereas updating the STM.

**How does the everything looks like mathematically in LSTM.**

Cell state, hidden state looks same where they are vectors of both same sizes. **(Ct, Ht).**

**Like the above two there are four more vectors like Ft, It, Ct, Ot and interestingly they are of same shape as Ct, Ht. i.e. means these six vectors are of same shape**

**Xt –** input vector (can be of any length not restricted to any).

**Pointwise operations:**

Arrays pointwise operations like (+, X)

**Nodes:** Each gate array is neural layers with activation function. It is a hyperparameter where the no. of nodes is decided, and they are same for all the layers. The no. of nodes is same as the no. of elements in those arrays like LTM, STM, Ft etc.

LTM and STM are of same shape at both the states, but the input Xt can be different.

**Forget GATE:**

Let’s suppose we have [1 X 4] input in Xt with 3 nodes in forget gate layer, in that case

The total input to the forget gate layer would be (Xt and ht) right which is called concatenation input, in this case that would be (4 + 3) which would be [1 X 7]

And weight matrix would be (3 X 7) as there are 3 nodes in the layer.

So, the output of the forget gate would be:

= (3 X 7)

= (3 X1)

= (4 X 1)

Then the output of this operation would have point wise operation with Ct. So, basically what is happening is we have previous hidden state on processing that hidden state with input of current state we get Ft which is nothing but what to remove from the cell state and that is why we do point wise operation between the Ft which now knows what to remove from the cell state.

Let’s suppose our cell state is [ 5,10,20] and our Ft comes out be [0.5,0.25,0.1] then current cell state after removal would be [2.5,2.5,2] and this is what happens in the forget gate.

**Output GATE:**

On magnifying the gate, we we’ll two layers in output gate first one is input layer with It array and other one is candidate cell state.

Ct is nothing but the potential information to be added in the cell state whereas it decides what and which of that potential info to add. It has sigmoid loss function whereas Ct has tanh.

The output equations look the same as forget gate with one change in error function.

For candidate cell state it looks like:

Though the product of weight and input state is same, but the values are different.

And similarly, for It**:**

On calculating the It and Ct we have multiplicative point wise operation which decides the which information out of the potential info is needed to be added in the cell state.

This quantity (Ct X It ) which is the actual filtered info to be added into the cell state is added in the cell state with point wise addition operation.

So, ultimately LSTM could add and maintain the whichever pervious info with the help of this gates. So, the main problem in the RNN which was vanishing gradient or loosing info over the longer distance can be easily fixed in the above case with Ft being one that is nothing is removed from the cell state and the product of Ct and it is zero then the actual cell state is maintained and carried throughout without losing the info.

**Output Gate:**

Output gate calculates the hidden state from the CELL STATE.

The output hidden layer has equal number of neurons as all the hidden layers with activation function as sigmoid.

Output array is calculated form the hidden state input ht and Xt concatenation with its weight matrix and the multiplicative point wise operation is done with the cell state array after applying tanh activation function. And that gives you the new cell state which is basically calculated from the output cell.

The equations are the same there is not much change as such. Just the direction is different.

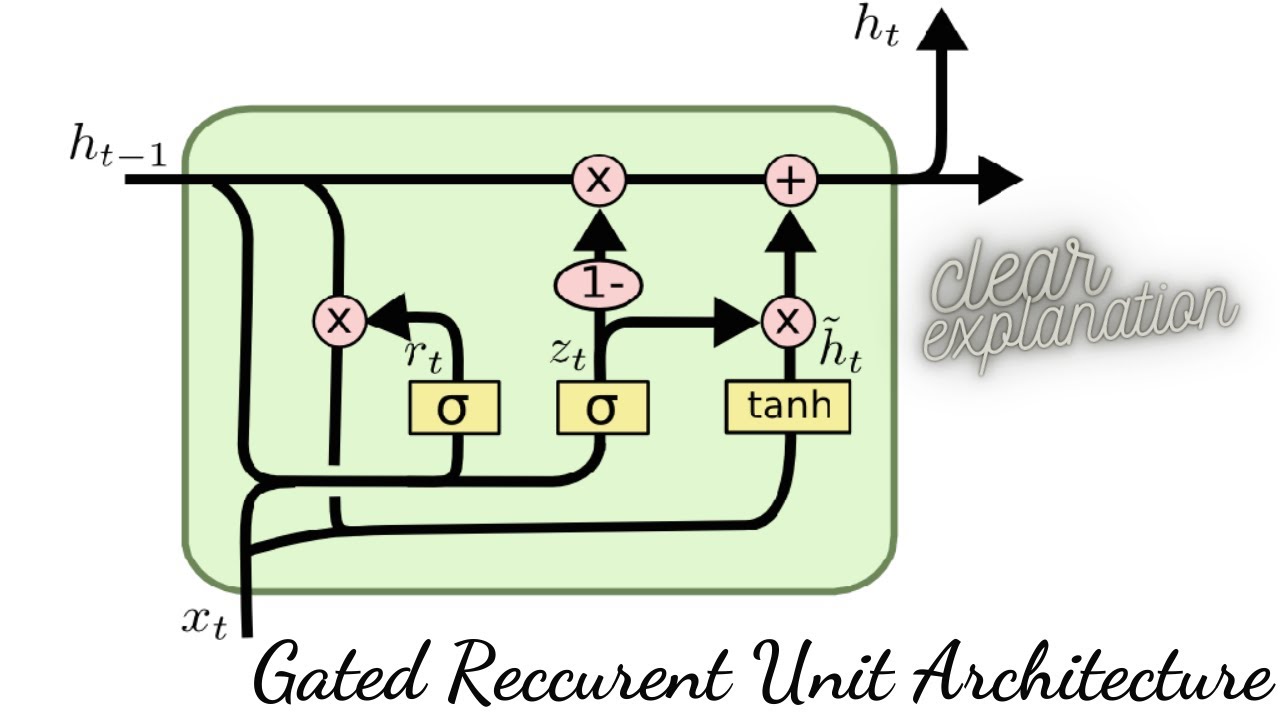
ADD the fuckinn diagram here.

**GATED Recurrent Unit (GRU) :**

GRU’s does the same work as LSTM’s but in more efficient way. GRU’s has much less trainable parameters compared to LSTM’s but their performance is comparable to the LSTM’s. Infect in some cases the GRU’s Outperform the LSTM’s.

GRU’s has only two gates compared to the 3 Gates in the LSTM’s and on top of that GRU’s don’t need any cell state to maintain the long term context it works with the same hidden state.

Setup:



Two gates:

1. Restore Gate (Rt) :
2. Update Gate (Zt) :
3. Candidate hidden state () :
4. Input state (Xt) :
5. Previous hidden state(ht-1) :
6. Current hidden state (ht) :

The shape of all the above arrays are nothing but the same except for the Xt whose shape depends on input state where as the other shape is same as the no. of neurons in the hidden layers.

Input (Xt):

Input state is a vector based on the vectorization. (OHE, BOW, Embeddings).

BAT CAT RAT

RAT RAT CAT

CAT BAT BAT

1. [[1,0,0],[0,1,0],[0,0,1]]
2. [[0,0,1],[0,0,1].[0,1,0]]

So, the input (Xt) would be a vector.

Architecture:

In GRU’s what’s happening is you have input Xt and ht-1 from these two we are calculating the ht.

The entire work of calculating the ht from ht-1 is divided into four steps:

1. Calculating the reset gate (Rt)
2. Calculating the () (candidate hidden state)
3. Calculating the (Zt).
4. Calculating the current hidden state (ht).

Flow of the GRU architecture.

1. We calculate the ht from the ht-1 and with the help of the update gate (Zt).
2. Whereas we calculate the from the input Xt and ht with the help of the reset gate.

The process of the GRU happens in two steps:

1. **First you calculate the candidate** from the current input which basically has the essence of the current state information. This has the potential to be the final ht, or at least some portion of which will become or influence the ht. If we directly take the candidate state as the ht then it will be normal RNN only and we’ll lose the sense of the previous info. So instead of totally replacing the previous ht with candidate memory we’ve add the appropriate extent of it or remove so that it has sense of past and present like LSTMs. So, candidate is nothing but the current hidden state. (RESET GATE).
2. **On calculating the current hidden state from hidden state,** we have update the required portion of that state which is candidate state to a previous hidden state in proportionate amount decided by update gate. **(UPDATE GATE)**

Let’s take an example of the story of the Vikram.

Vikram fought kali.

Vikram lost and died.

Vikram’s son was brave as Vikram.

He fought and lost to kali.

Vikram’s grandson was not as might as previously kings but he was smart.

He fought kali, was losing initially and but eventually beat kali with his smart strategies and won.

So, during all this story ht would look like:

Let’s suppose ht store contexts like [ power, conflict, tragedy, revenge] .

1. [0.7, 0.8, 0.1, 0]
2. [0.5, 0.5, 0.9, 0]
3. [0.8, 0.3, 0.3, 0]
4. [0.7, 0.8, 0.9,0]
5. [0.8, 0.5, 0.4, 0]
6. [0.7, 0.9, 0.1, 1]

This is how the ht gets updated with respect to the current state on top of the previous ht’s.

How does the calculation flow occurs:

Put the screenshot from the notes.

But anyways, we calculate Rt from the Xt and ht-1, with the help of the Rt X ht-1 and Xt we calculate the candidate state .

From ht-1 and Xt we have Zt and finally form Zt,ht-1,and we calculate the ht.

So all we have to calculate is Rt and Zt.

Reset Gate (Rt):

Representation [ 0.1, 0.5, 0.9, 1]. It is an array with numbers from 0 - 1.

It is a gate which reset the previous hidden state to a proportionate degree.

**How do you calculate Reset Gate(Array):**

* You have neural layer with sigmoid as an activation function.

Lets suppose you have a neural layer with 3 neurons and input state is 4 element array.

Then the concatenation input with input Xt and ht-1 would be [1 X 7 ] and Wr = [ 7 X 3 ] so the output would be [ 1 X3 ] reset gate array after adding bias and putting everything into activation function.

Ht-1 = [ 0.8, 0.6, 0.9, 0.4]

Rt = [ 0.5, 1, 0.66, .8]

With the help of the reset gate and input (Xt) you calculate the candidate array with pointwise multiplicative operation that gets you **modularize hidden state.**

= [ 0.4, 0.6, 0.6, 0.32]

Now this output along with the input (Xt) when put into a neural layer of tanh activation function we get candidate hidden state.

**Update Gate :**

Intuition: Zt decides which state to give more importance to current potential ht which is candidate hidden state or the previous hidden state in formation of the current hidden state.

If we have high Zt then more importance is given to the current hidden candidate state and if the Zt is low then more importance is given to the previous hidden state.

So, Zt decides which state to give more importance to in calculating the current ht. where as Rt decides which factors are more important from the current state which can be made as potential candidate for the actual current hidden state.

So Rt, decides the important parameters with their weights for the candidate hidden state as Rt being the gate where as Zt decides which of the two i.e. previous hidden state(ht) or the candidate hidden state should have higher impact in calculating the actual current hidden state.

Ht = (1- Zt) \* (Ht-1) + (Zt \* )

So, from the Zt one input goes to ht-1 as (1-zt) and other goes to the and finally we have a additive operation between both to ultimately decide for the current hidden state.

**Deep RNN’s:**

**The** Idea in deep RNN is to stack multiple RNN layers stacked upon each other. Results into a extraction of more complex features.

The output of the one RNN which is hidden state is given to the second RNN which stacked on top of that first RNN. Don’t get confused here with output given to the second timestamp.

This output of the ht is given to the second layer RNN for the same timestamp only. And likewise, we complete the timestamp one processing for all RNN layers and then followed for the second timestamp again for the deep layers.

Check code for better understanding.

In deep **LSTM’s** only the ht is given to the second layer of the LSTM and not the Ct as well. Only **Ht**.

**Story of the LLM’s:**

Based on the input output structure there are many types of the RNN’s:

1. Many to One RNN models: multiple input but has one output. i.e. sentiment analysis.
2. One to Many Analyses: One input generates multiple output. image description module.
3. Many to Many:

3.1 : Synchronous : where input length and output length is same. E.g. Part of speech tagging for each word. Named entity recognition for each word.

3.2 : Asynchronous : Where input length and output length is not same. E.g. Language translation, text summarization, question and answers, speech to text (subtitles).

Sequence to sequence problems:

Solves the problem of many to many asynchronous problems.

Which problems does it solves: Language translation, text summarization, question and answers, speech to text (subtitles).

Story of the sequence-to-sequence models consist of 5 parts:

1. Encoder Decoder.
2. Attention Mechanism
3. Transformers
4. Transfer Learning
5. LLM’s
6. Encoder Decoder: Encoder block compresses the input words and pass it to the decoder block.

* Now what decoder does is it uncompressed the input and returns the output.
* Encoder on compression keep the info in context vector which has the essence of all the information.
* So encoder get the information in the form of context vector. But as the words get more and more i.e. sentences gets higher and higher context vector is not able to keep the context of much previous words and it becomes recency biased context vector.
* Sentences with usually words greater than 30 is an issue.

1. Attention Mechanism:

* To overcome the drawbacks of the encoder decoder architecture we have attention mechanism.
* Unlike Encoder decoder which has only one context vector to summarize all the information of all the timestamp which then given to the decoder, Attention mechanism has access to all the states of encoder i.e. all the ht’s and ct’s of the encoder for each step unlike one context vector.
* So, for each specific word it does not rely on the compressed context vector which has the context of all the states in one vector.
* Attention mechanism has one neural layer attention layer which decides which of those all the stored state outputs is to be selected for that specific word prediction.
* So, it kind of put attention on the specific needed states which are needed and that happens with the help of the neural layer.
* Higher computational complexity.
* Both the Encode decoder as well as Attention Mechanism both has LSTM in them.

**Transformers:**

What transformer did, as they changed the landscape through which we process the NLP task.

Attention is all you need. The paper which first describes the transformers.

It does not have any LSTM’s, but it only emphasises attention.

**Drawbacks:**

**Training time**

**Hardware**

**Cost**

Transfer Learning:

Initially transfer learning would be used only in the CNN. But with

Complete after you watched that content again.

Though the points are as such:

1. Pre-Training
2. Fine- Tuning

Language models:

Transfer learning applicable model which are trained on very large amount of data.

GPT: GPT is a language model which is trained on the very large data.

GPT 3.5 was 45 TB of data with near about the 175 billion parameters.

Whereas GPT 4 is 1.8 trillion parameters.

Chat GPT is chat bot product developed from the GPT language model.

1. **Encoder Decoder**

Initially for tabular data we had ANN which works fine for tabular data without any sequence in it. For image data we had CNN and for sequential data where there is sequence in the data, we use RNN’s.

But in all the above case we have M inputs and 1 output we can manage variable inputs with zero array but in case there is n outputs it is bit tricky and for the same use sequence to sequence models.

Sequence to sequence models works on m variable inputs and n variable outputs.

Sequence to sequence models basic overview is very simple with one encoder block and other decoder block. Encoder block has LSTM / GRU in it and receives sequential array’s of input and generates a context vector which has the context of the input fed. This context vector is sent to the decoder which receives the one element of the context vector for one timestep and return the final output. But for firsts timestamp it receives one flag called as start which starts the decoding operations and starts producing the outputs whereas at the end decade’s LSTM receives the End flag which stops the anymore output being produced. Now this context vector is nothing but the cell state of the LSTM not both cell as well as hidden state.

Put the images from the Notes here,

Training the architecture by back propagation

**Forward propagation.**

* How does data look like: In our case let’s take an example of language translation model where we input we have language and output we have translated meaning of that sentence or word.
* Encoding the input to the Encoder and pass it in timestep with multiple timestep generating the context vector. After encoding both the input and output sequential data we have numerical sequence to sequence data whose patters needs to be learned by the sequence-to-sequence architecture.
* Encoder generates the context of the data which is nothing, but it learns the meaning of the sentence of the in encoding only to a degree and passes that learned meaning context to the decoder.
* Now decoder has one fully connected SoftMax layer which is used to predict the output corresponding to each time step. The no of nodes in the SoftMax layers are equal to the timestamp of the decoder. The max probability of the SoftMax layers prob vector is the output of that layer or rather the words corresponding to the index of the max probability as per the encoding.
* And likewise, you predict the next word for the next timestamp based on the same SoftMax probability.
* Here while learning if the decoder predicts the wrong word a right word is still given to the next timestamp as input that is overwriting the wrong output of the previous timestamp which is called **forced teaching** and this what led you to have optimized fast learning. And obviously the initial weights are defined randomly or initialization techniques. That forced teaching is sort of having proper target column for the input to learn from in decoder. Instead of learning from the wrong learn from the right.
* The probabilities where the prediction is correct are very high compared to the wrong prediction.

**Evaluation:**

After forward propagation we need to evaluate the prediction accuracy we do that with the help of the loss function.

Put the loss images :

Categorical cross entropy:

Calculate the loss for each timestep with the help of categorical cross entropy and aggregate the entire loss either mean or sum of all the losses of each timesteps.

The loss for the timestep where model was right is lower compared to the losses where model has made mistake.

Once we get the loss now, It’s time for **backpropagation.**

Initialization of the weight 🡪 Forward propagation 🡪 output 🡪 loss 🡪 gradient 🡪weight update (optimizer) 🡪 learning rate 🡪 new\_weight 🡪 new record 🡪 next epoch.

On calculating the loss, we calculate the gradient and with the help of the optimizer we update the weight parameters with involvement of the learning rate.

This is how the training takes place with final weights.

**Prediction:**

Pretty much like the training with the difference of force teaching is not there.

And that is how we calculate the new output.

Put the diagram form the notes.

Improvement in sequence-to-sequence models:

1. **Embeddings:**

Using embedding gives you more dense representation of the words with low dimension representation.

So, you add an embedding layer at the beginning.

Pretrained embeddings word2vec, glove etc.

Encoding in encoder and decoder.

1. **Deep LSTM’s:**

Three reasons:

2.1 Long term dependencies: Able to remember the long-term contexts as we have multiple layer context vectors which keeps the longer context in memory and that is why it performs better for longer context vectors.

* 1. Layered (Hierarchical) representation: lower layer LSTMs understand the word level context like POS etc. whereas the middle layer LSTMs understand the sentence level contexts and finally the top layer LSTM’s understand the paragraph level context.
  2. More parameters: On general level when you increased the number of parameters in the Neural network the level of understanding to which the NN understands gets increased and NN understands things on deeper level if we avoid overfitting.

1. **Reverse the input:**

Like give the reverse input in the encoder and the original correct one in Decoder,

The basic logic is that when we give the input in reverse order in encoder and with the correct sequence in the decoder the distance between the initial words and final words in the decoder reduces and that sometimes helps in understanding the initial context far better especially for the languages where the initial words hold more importance.

**Transformers:**

**Work on sequence-to-sequence problems like**

1. **Machine Translation**
2. **Question answers**
3. **Text Summarization**

**Transforms has self-attention in them which makes them process the multiple sequential input in parallel and hence can be trained on large data which was not possible in sequence-to-sequence models with attentions.**

**Impact of Transformers:**

1. **Revolutionizing NLP:** NLP has been the main thing central thing in AI. With the use of transformer, it got revolutionized
2. **Democratizing AI:** BERT and GPT models have made open source that propelled lot of enhancement in AI
3. **Multimodal capability:** Not only for NLP but transformer can be used instead of CNN, ANN i.e. image or tabular data.
4. **Acceleration of Gen AI:** instead of GANs transformer are used widely.
5. **Unification of the Deep Learning:** Transformers are used for Gen AI, reinforcement learning, NLP etc.

Like mentioned above sequence to sequence to learning with attention has a big flaw of not able to train [parallelly and hence not able to have transfer learning in NLP. The exact above issue is solved by transformers. It is very stable and scalable architecture.](https://www.merriam-webster.com/dictionary/parallelly" \l ":~:text=%3A%20in%20a%20parallel%20manner)

[Brief history timeline of the DL.](https://www.merriam-webster.com/dictionary/parallelly" \l ":~:text=%3A%20in%20a%20parallel%20manner)

[2000 – 2014 -- RNNs/ LSTMs](https://www.merriam-webster.com/dictionary/parallelly" \l ":~:text=%3A%20in%20a%20parallel%20manner)

[2014 – Attention](https://www.merriam-webster.com/dictionary/parallelly" \l ":~:text=%3A%20in%20a%20parallel%20manner)

[2017 – Transformers](https://www.merriam-webster.com/dictionary/parallelly" \l ":~:text=%3A%20in%20a%20parallel%20manner)

[2018 – BERT, GPT, Transfer learning](https://www.merriam-webster.com/dictionary/parallelly" \l ":~:text=%3A%20in%20a%20parallel%20manner)

[2018 – 2020 – vision transformer / Alpha fold etc.](https://www.merriam-webster.com/dictionary/parallelly" \l ":~:text=%3A%20in%20a%20parallel%20manner)

[2020-2021 – Gen AI](https://www.merriam-webster.com/dictionary/parallelly" \l ":~:text=%3A%20in%20a%20parallel%20manner)

2022 – chat GPT, stable diffusion.

Advantages of the Transformers:

1. Scalability
2. Transfer Learning.
3. Multimodal
4. Flexible architecture:
   1. Only decoder model
   2. Only encoder model
5. Ecosystem: Lot of libraries like hugging face and lot of open-source work is around Transforms.
6. Integrated AI: Transformers with GAN like DALLI, Transformers with reinforcement learning for game playing agents.
7. Vision transformer: CNN + transformer

Disadvantages of the Transformers:

1. Hi computations
2. High data
3. Over fitting
4. Energy consumptions
5. Interpretability
6. BIAS (bias from the data and ethical concerns)

Future:

1. Improvement in efficiency
2. Multimodality
3. Responsibility
4. Domain specific
5. Multilingual
6. Interpretability

Self-Attention:

Normal encoding techniques we have are OHE, BOW and TFIDF and finally we have Embedding’s.

In embedding an unsupervised text data is learned by a model which represents a word in a dense vector with each element represents a magnitude value of certain characteristics.

E.g. word “KING”

[0.6, 0.7, 0.9, 0.4] and

“Queen”

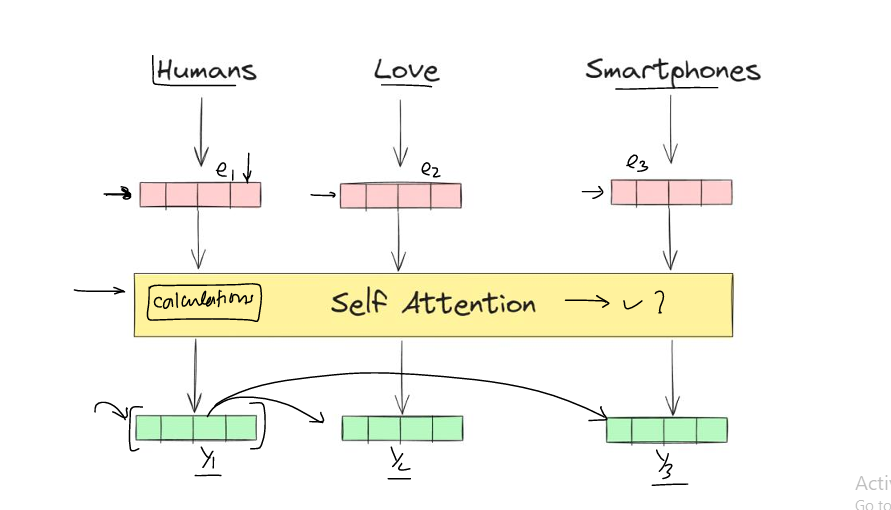
[0.4, 0.3, 0.9, 0.6]

Each element of these two arrays would represents a characteristic of the king and queen and value would be the magnitude of those characteristic.

As king and queen are similar words these two vectors are very similar to each other with very minimal angle between them.

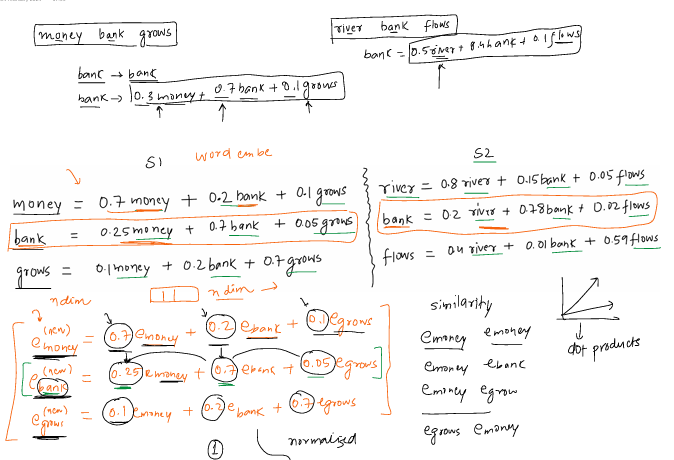
As this embedding’s are formed by training huge unsupervised data once and these static embeddings are used over and over again for many different applications where the context of those words would be different but the embedding representation of the words is not contextual but it is static cause it represent the average meaning the word. I.e. a word is represented by single value of the characteristic for all the contexts instead of different vectors for different contexts. Means you have single vectors for all the different context instead of multiple different vectors for different contexts.

With the self-attention you can represent the word with contextual embedding and ultimately that embedding is process by the transformer down the line.



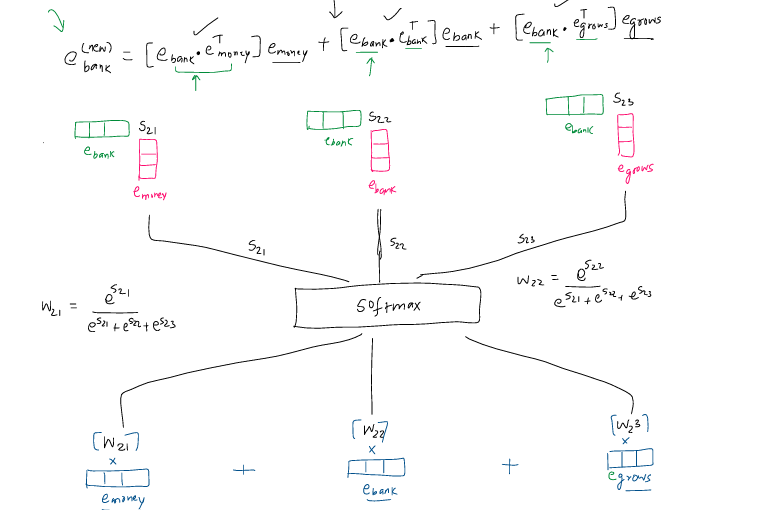
**Self-Attention**

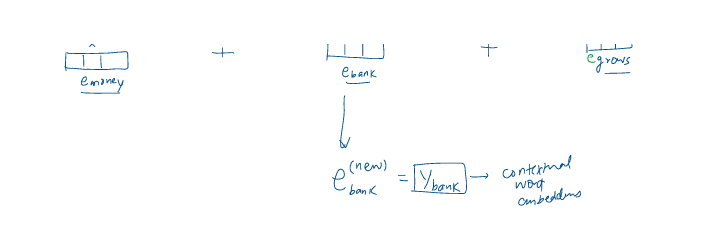
**First principal approach: (for calculating contextual embedding vector)**

****

So basically, the new embedding vector for the money is calculated based on the other words by considering the similarity with each word. The similarity between the word embedding is calculated by dot product between the vectors.

The normalized dot product is taken with SoftMax for similarity score.





This operation is parallel operation, and no parameters are involved in the above embeddings.

**Make sure to upload the file from the local comp.**