DL Notes

Author : Arnab Dana

▼ *Resource*

https://www.udemy.com/course/complete-machine-learning-nlp-bootcamp-mlops-deployment/

CAMPUSX: https://www.youtube.com/playlist?list=PLKnIA16_RmvYuZauWaPlRTC54KxSNLtNn

https://www.youtube.com/playlist?list=PLyqSpQzTE6M9gCgajvQbc68Hk_JKGBAYT,
https://archive.nptel.ac.in/courses/106/106/106106184/

https://www.youtube.com/playlist?list=PLeo1K3hjS3uu7CxAacxVndI4bE_o3BDtO

https://www.youtube.com/@codebasicsHindi/playlists

Papers

https://arxiv.org/abs/2011.02323

Paramanu: A Family of Novel Efficient Generative Foundation
Language Models for Indian Languages: https://arxiv.org/pdf/2401.18034

▼ *Content*

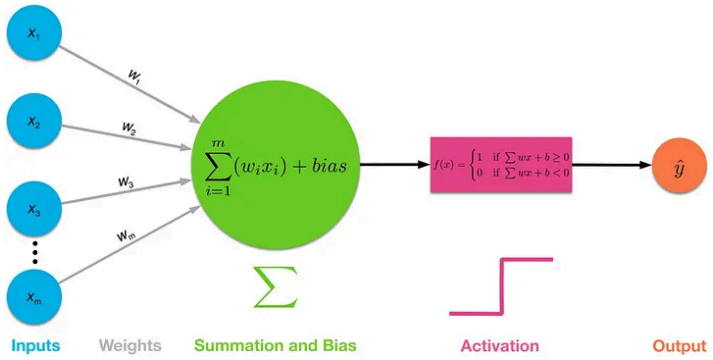
27. ***GPT (Decoder-Only Transformers)***
28. ***T5 (Encoder-Decoder Transformers)***
29.

▼ 1. *Definition*

▼ 2. ***ANN**, **CNN**, and **RNN***

Feature	ANN (Artificial Neural Network)	CNN (Convolutional Neural Network)	RNN (Recurrent Neural Network)
Architecture	Fully connected layers	Convolutional + pooling + fully connected layers	Loops in architecture to handle sequences
Input Type	Fixed-size vector	Grid-like data (e.g., images)	Sequential data (e.g., time series, text, audio)
Best For	Tabular data, general-purpose learning	Image processing, computer vision	Time series, speech, natural language processing
Memory of Past Inputs	No	No	Yes (has memory of previous inputs)
Feature Extraction	Manual or learned via dense layers	Automatic via convolution filters	Temporal pattern learning via recurrence
Parameters	More parameters (dense connections)	Fewer parameters (shared weights in convolution layers)	Moderate; depends on sequence length
Training Complexity	Relatively simple	Computationally intensive but optimized	Complex (especially due to vanishing/exploding gradients)
Handling Long Dependencies	Poor	Not applicable	Difficult for vanilla RNNs (solved by LSTM/GRU variants)
Common Variants	MLP (Multilayer Perceptron)	LeNet, AlexNet, ResNet, VGG	LSTM, GRU, Bi-RNN
Use Cases	Classification, regression	Image classification, object detection	Language modeling, speech recognition, translation

▼ 3. ***Single Layer** Perception **(Feedforward Neural Networks)***



A **Perceptron** is a **binary classification algorithm** that maps input features to an output using a **linear decision boundary**. It's the building block of neural networks.

It is typically used for supervised learning.

Perceptron Algorithm - Step by Step

1. **Initialize Weights and Bias**
   - Initialize weights: w1, w2, ..., wn (small random values)
   - Initialize bias: b
2. **Receive Input**
   - Input vector: X = [x1, x2, ..., xn]
3. **Calculate Weighted Sum (Net Input)**
   - z = w1 * x1 + w2 * x2 + ... + wn * xn + b

- This can be written as: z = dot_product(W, X) + b

4. **Apply Activation Function**
   - If using step function:

     if z >= 0 then

     output = 1

     else

     output = 0
5. **Make Prediction**
   - Use the result of the activation function as predicted class (0 or 1)
6. **Calculate Error**
   - error = y_true - y_pred
7. **Update Weights and Bias**
   - For each weight wi:

     wi = wi + learning_rate * error * xi
   - Update bias:

     b = b + learning_rate * error
8. **Repeat**
   - Repeat steps 2 to 7 for multiple epochs or until error is minimized

Example

x1	x2	y (true label)
0	0	0
0	1	0
1	0	0
1	1	1

Initial Settings

Weights: w1 = 0, w2 = 0 | Bias: b = 0 | Learning Rate: η = 1

Epoch 1

Input (x1, x2)	y (True)	z = w1*x1* + *w2*x2 + b	Prediction	Error (y - pred)	Weight Update	Bias Update
(0, 0)	0	0	1	-1	w1 = 0 + 1 * (-1) * 0 = 0 w2 = 0	b = 0 - 1 = -1
(0, 1)	0	-1	0	0	No change	No change
(1, 0)	0	-1	0	0	No change	No change
(1, 1)	1	-1	0	1	w1 = 0 + 1 * 1 * 1 = 1 w2 = 0 + 1	b = -1 + 1 = 0

After Epoch 1

Updated Weights: w1 = 1, w2 = 1 | Bias: b = 0

Epoch 2

Input (x1, x2)	y (True)	z = w1*x1* + *w2*x2 + b	Prediction	Error (y - pred)	Weight Update	Bias Update

(0, 0)	0	0	1	-1	w1 = 1, w2 = 1 (no change, x=0)	b = 0 - 1 = -1
(0, 1)	0	0	1	-1	w2 = 1 + (-1) * 1 = 0	b = -1 - 1 = -2
(1, 0)	0	-1	0	0	No change	No change
(1, 1)	1	-1	0	1	w1 = 1 + 1 = 2, w2 = 0 + 1 = 1	b = -2 + 1 = -1

Advantages of Single Layer Perceptron:

1. **Simplicity:**
   - The SLP is simple and easy to understand. Its architecture consists of just one layer of weights connecting the input layer to the output layer. This makes it relatively straightforward to implement.
2. **Computational Efficiency:**
   - Since it has only one layer, the SLP is computationally less expensive compared to more complex models like deep neural networks. This makes it faster to train, especially on small datasets.
3. **Linear Classification:**
   - A Single Layer Perceptron can be effective for problems where the classes are linearly separable. For such problems, the SLP can learn a linear boundary (hyperplane) that separates the different classes.
4. **Foundation for Neural Networks:**
   - Despite its simplicity, the Single Layer Perceptron is a foundational concept for understanding neural networks. It introduces key ideas like weights, activation functions, and the learning process (such as gradient descent).

Disadvantages of Single Layer Perceptron:

1. **Limited to Linear Decision Boundaries:**
   - The biggest limitation of a Single Layer Perceptron is that it can only solve linearly separable problems. If the data cannot be separated by a straight line (in higher dimensions, a hyperplane), the SLP will not perform well. It cannot model non-linear decision boundaries.
2. **Limited Expressive Power:**
   - The SLP has limited capacity to represent complex patterns in data. It lacks the depth and non-linearity required for solving more complicated problems, like those found in image recognition or language processing.
3. **No Capability for Feature Learning:**
   - Unlike multi-layer neural networks, a Single Layer Perceptron doesn't have the ability to learn hierarchical features. In deep neural networks, the additional layers allow for more complex feature extraction and abstraction.
4. **Training Limitations:**
   - The training process in SLP is often slow and inefficient, especially for more complex datasets. It may require more sophisticated techniques (like backpropagation) and may not converge properly if the data is not linearly separable.
5. **Inability to Handle Complex Data:**
   - SLPs are not suited for handling complex datasets like images, audio, or videos. More advanced architectures such as Convolutional Neural Networks (CNNs) or Recurrent Neural Networks (RNNs) are typically used for such tasks.

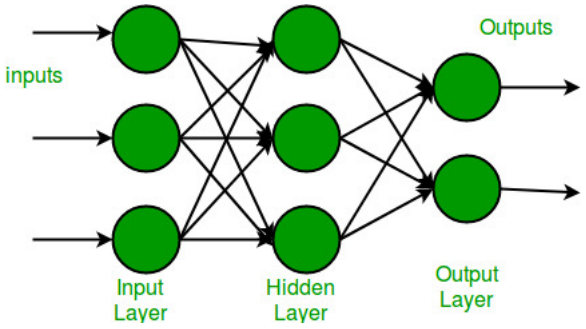
Conclusion:

While the Single Layer Perceptron is useful for basic tasks where the data is linearly separable, it is limited in its ability to handle complex, real-world problems that require non-linearity or feature learning. For more advanced

tasks, multi-layer networks are generally preferred.

▼ *4. ** **Multi-Layer Layer** Perception (ANN / **MLP**)*

Multi-Layer Perceptron Learning in Tensorflow | GeeksforGeeks



Working of Multi-Layer Perceptron

1. Input layer receives data
2. Data is forwarded to hidden layers
3. Weighted sum is calculated
4. Bias is added
5. Activation function is applied
6. Output of hidden layer becomes input to next layer
7. Steps 3–6 are repeated for all layers
8. Final output is computed at output layer
9. Loss is calculated using loss function
10. Backpropagation is performed
11. Gradients are calculated using chain rule
12. Weights and biases are updated using optimizer
13. Steps 1–12 are repeated for multiple epochs until convergence

Advantages of Multi Layer Perceptron

- **Versatility**: MLPs can be applied to a variety of problems, both classification and regression.
- **Non-linearity**: Thanks to activation functions, MLPs can model complex, non-linear relationships in data.
- **Parallel Computation**: With the help of GPUs, MLPs can be trained quickly by taking advantage of parallel computing.

Disadvantages of Multi Layer Perceptron

- **Computationally Expensive**: MLPs can be slow to train, especially on large datasets with many layers.
- **Prone to Overfitting**: Without proper regularization techniques, MLPs can overfit the training data, leading to poor generalization.
- **Sensitivity to Data Scaling**: MLPs require properly normalized or scaled data for optimal performance.

▼ *5. Image Processing Filter*

A filter is a mathematical operation applied to an image to modify its pixels based on a predefined rule or kernel (a small matrix). It processes the image to enhance, suppress, or extract specific features.

```
// Sobel X Kernel (detects vertical edges)
[ [-1,  0,  1],
  [-2,  0,  2],
  [-1,  0,  1] ]

// Sobel Y Kernel (detects horizontal edges)
[ [-1, -2, -1],
  [ 0,  0,  0],
  [ 1,  2,  1] ]
```

Purpose

- **Enhance**: Improve image quality (e.g., sharpening, contrast adjustment).
- **Suppress**: Reduce noise or unwanted details (e.g., smoothing, blurring).
- **Extract**: Detect edges, textures, or patterns (e.g., edge detection).

Types of Filters:

- **Spatial Filters**: Operate directly on pixel values in the image domain.
  - **Linear Filters**: Use convolution with a kernel (e.g., Gaussian blur, Sobel edgeTED for edge detection).
  - **Non-linear Filters**: Apply conditional operations (e.g., median filter for noise reduction).
- **Frequency Filters**: Operate in the frequency domain using Fourier transforms (e.g., low-pass, high-pass filters).

Linear relationships involve additive and multiplicative operations, ensuring predictable scaling and combination of inputs.

Non-linear relationships involve operations like sorting, clipping, or logic-based decisions, making them more flexible for complex tasks (e.g., edge-preserving noise reduction) but less predictable mathematically.

A filter is called *linear* if the output is a linear combination of the input pixel values.
A filter is *non-linear* if the output is not a linear combination of input pixels.

Linear Filter

Name	Purpose
Gaussian Filter	Smoothing/Blurring
Box Filter (Mean)	Smoothing/Averaging
Sobel Filter	Edge Detection (Gradient)
Prewitt Filter	Edge Detection (Gradient)
Roberts Cross Filter	Edge Detection (Gradient)
Laplacian Filter	Edge Detection (Second Derivative)
Scharr Filter	Edge Detection (Improved Sobel)
High-pass Filter	Sharpening/Edge Enhancement
Low-pass Filter	Noise Reduction/Smoothing
Emboss Filter	Highlight edges with direction
Motion Blur Filter	Directional Blurring
Gabor Filter	Texture/Edge Analysis

Non-Linear Filters

Name	Purpose
Median Filter	Noise Reduction (Salt & Pepper)
Max Filter	Bright Region Enhancement
Min Filter	Dark Region Enhancement

Name	Purpose
Mode Filter	Noise Reduction
Alpha-Trimmed Mean	Noise Reduction with control
Adaptive Median Filter	Noise Reduction (Adaptive kernel)
Bilateral Filter	Edge-Preserving Smoothing
Kuwahara Filter	Edge-Preserving Smoothing
Rank Filter	Generalized Pixel Ranking
Weighted Median Filter	Enhanced Median Filtering
Conditional Filter	Custom Filtering Based on Criteria
Anisotropic Diffusion	Edge-Preserving Image Smoothing

Frequency Filters

Name	Purpose
Ideal Low-pass Filter	Removes high-frequency noise
Ideal High-pass Filter	Enhances edges, removes smooth areas
Butterworth Low-pass	Smooth transition, noise reduction
Butterworth High-pass	Edge enhancement, gradual transition
Gaussian Low-pass	Soft blurring, noise reduction
Gaussian High-pass	Smooth edge enhancement
Band-pass Filter	Preserves specific frequency bands
Band-stop Filter	Removes specific frequency bands
Homomorphic Filter	Illumination correction, contrast

▼ *6. ** Feature Extraction*

▼ *7. ** Feature Selection*

▼ *8. **Recurrent Neural Networks (**Simple or Vanilla RNN)*

https://www.geeksforgeeks.org/introduction-to-recurrent-neural-network/

https://www.datacamp.com/tutorial/tutorial-for-recurrent-neural-network

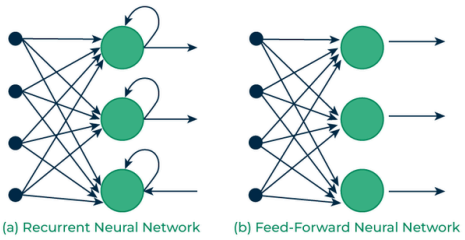
Problem in Feedforward Neural Networks

1. **No Temporal Memory -** FNNs treat each input independently and lack the ability to remember previous inputs.
2. **Fixed Input/Output Size -** FNNs require fixed-size inputs and outputs, making them unsuitable for sequential data like text or time series.
3. **Poor Performance on Sequential Data -** FNNs do not capture order or context, so they struggle with tasks where the position or sequence matters.
4. **Not Suitable for Variable-Length Inputs -** Since FNNs expect fixed-size input vectors, they cannot naturally handle sequences of varying lengths.

How RNNs Solve These Problems:

1. **Maintains a Hidden State (Memory) -** RNNs have loops that allow information to persist, meaning they remember previous inputs through their hidden states.
2. **Handles Sequential Data -** RNNs process sequences one element at a time, maintaining context across the sequence.
3. **Supports Variable-Length Sequences -** RNNs can process inputs of variable length because they operate step-by-step through time.
4. **Context-Aware Predictions -** RNNs consider previous inputs when making predictions, making them ideal for tasks like language modeling, translation, or time series forecasting.

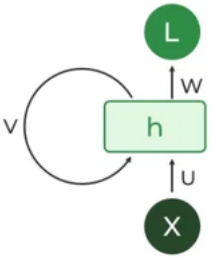
Recurrent Neural Networks (RNNs) solve this **by incorporating loops that allow information from previous steps to be fed back into the network**. This feedback enables RNNs to remember prior inputs making them ideal for tasks where context is important.



Key Components of RNNs

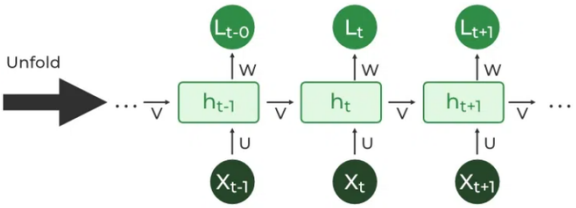
1. **Recurrent Neurons**

The fundamental processing unit in RNN is a **Recurrent Unit.** Recurrent units hold a hidden state that maintains information about previous inputs in a sequence. Recurrent units can "remember" information from prior steps by feeding back their hidden state, allowing them to capture dependencies across time.



2. **RNN Unfolding**

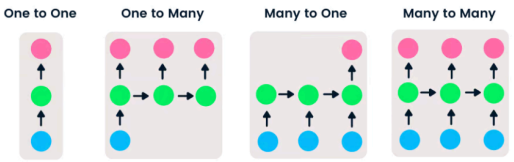
RNN unfolding or unrolling is the process of expanding the recurrent structure over time steps. During unfolding each step of the sequence is represented as a separate layer in a series illustrating how information flows across each time step.



Types of Recurrent Neural Networks Bashed on Input Output

- **One-to-One**
  - Not technically RNN (standard neural network).
  - One input → One output.
  - Example: Image classification.
- **One-to-Many**
  - Single input → Sequence of outputs.
  - Used when one input triggers multiple outputs.
  - **Example**: Image captioning (image → caption).
- **Many-to-One**
  - Sequence input → Single output.

    - Common for classification tasks.
    - **Example**: Sentiment analysis (review text → sentiment label).
- **Many-to-Many (Equal Length)**
    - Input and output sequences have the **same length**.
    - **Example**: POS tagging, where each word gets a corresponding tag.
- **Many-to-Many (Unequal Length / Encoder-Decoder)**
    - Input and output sequences have **different lengths**.
    - **Example**: Machine translation (English sentence → French sentence).



- **Many-to-Many also called Sequence to Sequence**

Main types of RNNs

1. Vanilla RNN (Basic RNN)

- The simplest form of RNN.
- Maintains a hidden state that updates at each time step.
- Struggles with learning long-term dependencies due to vanishing/exploding gradients.

2. LSTM (Long Short-Term Memory)

- Designed to handle long-term dependencies.
- Uses memory cells with three gates: input gate, forget gate, and output gate.
- More complex but effective for longer sequences.

3. GRU (Gated Recurrent Unit)

- A simpler alternative to LSTM.
- Combines the forget and input gates into a single update gate.
- Faster to train, often with comparable performance to LSTM.

4. Bidirectional RNN

- Processes input in both forward and backward directions.
- Useful for tasks where future context improves understanding (e.g., speech recognition, language modeling).

5. Deep RNN (Stacked RNN)

- RNNs with multiple layers stacked on top of each other.
- Captures more abstract and hierarchical features.
- Can model more complex sequential patterns.

Applications of RNNs

RNNs are widely used in fields that require sequence processing:

- **Natural Language Processing (NLP)**: Language modeling, text generation.
- **Speech Recognition**: Transcribing spoken words into text.
- **Machine Translation**: Translating text from one language to another.
- **Time Series Prediction**: Stock market prediction, weather forecasting.
- **Music Generation**: Composing music by predicting sequences of notes.

Problems in RNN

1. **Vanishing Gradient Problem (main)**

- During backpropagation through time (BPTT), gradients can become very small.
- As a result, the network stops learning long-term dependencies because earlier layers receive negligible updates.

2. **Exploding Gradient Problem**

- The opposite of vanishing gradients: gradients can grow very large, leading to unstable training.
- This can cause the model weights to diverge, resulting in NaN values or erratic behavior.

3. **Difficulty in Capturing Long-Term Dependencies**

- Even without gradient issues, vanilla RNNs struggle to retain information over long sequences.
- This limits their performance on tasks requiring memory of earlier input (e.g., language modeling, time-series forecasting).

4. **Training Time and Computational Cost (main)**

- Sequential nature of RNNs means computations cannot be easily parallelized.
- As a result, training can be slow, especially for long sequences.

5. **Lack of Flexibility with Variable-Length Dependencies**

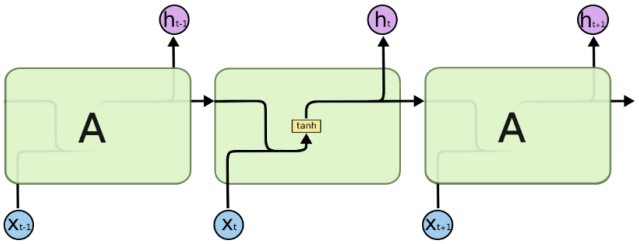
- RNNs often assume a fixed length of dependencies and struggle with varying temporal patterns.
- This makes them less effective for tasks where important information may appear at unpredictable points in a sequence.

6. **Overfitting on Small Datasets**

- RNNs, especially deep or complex ones, have many parameters.
- Without enough data or regularization (e.g., dropout), they tend to overfit.

▼ *9. RNN **Forward Propagation***

https://www.youtube.com/playlist?list=PLGP2q2bIgaNzBBpxxNUf126chLsQ20dfG



1. Review → Sentiment

- movie was good → 1
- movie was bad → 0
- movie was not good → 0

Find out Sentiment

2. Encoding

Vocab → 5 words:

movie, **was**, **good**, **bad**, **not**

Encoding:

- movie → [1 0 0 0 0]

- was → [0 1 0 0 0]
- good → [0 0 1 0 0]
- bad → [0 0 0 1 0]
- not → [0 0 0 0 1]

3. RNN takes input as : (timesteps or word, input features)

   Token Number is act as time stamp t = 1, t = 2, t = 3

   Examples:

   - Review "movie was good"
     - [[1 0 0 0 0], [0 1 0 0 0], [0 0 1 0 0]] → shape (3,5)
     - Here we have 3 time stamp, input features = **Vocab size = 5**
   - Review "movie was not good" → shape (4,5)
   - In Keras input takes : (Batch Size, timesteps, input features)
     - Batch Size mean corpus size here 3 (number of review).

4.

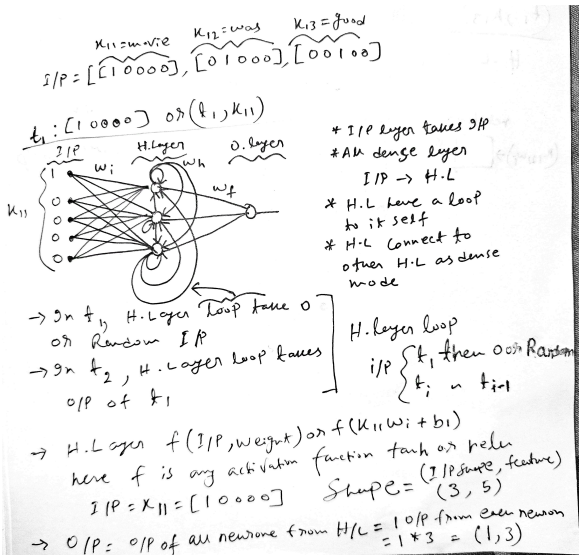
Input Reviews and Sentiments:

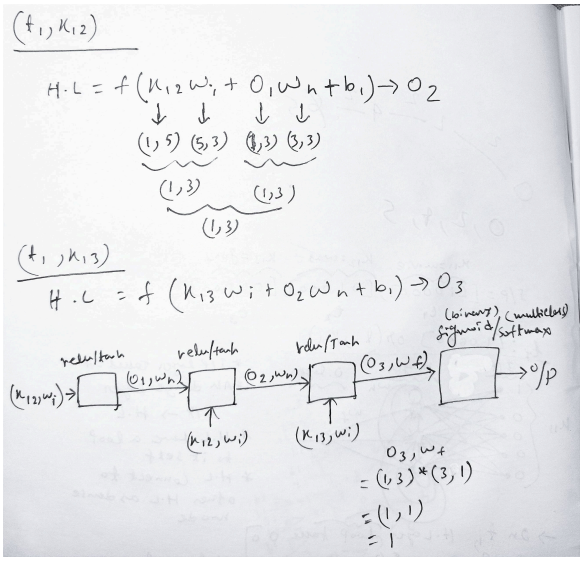
$$x_1 = \text{movie was good} \rightarrow 2$$
$$x_2 = \text{movie was bad} \rightarrow 0$$
$$x_3 = \text{movie was not good} \rightarrow 0$$

Each review is broken down into timesteps:

$$x_{11}, x_{12}, x_{13} = \text{movie, was, good}$$
$$x_{21}, x_{22}, x_{23} = \text{movie, was, bad}$$
$$x_{31}, x_{32}, x_{33}, x_{34} = \text{movie, was, not, good}$$

5.





Weight matrices:

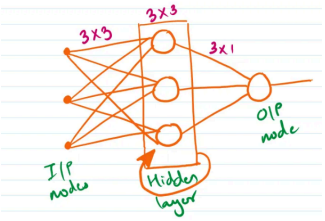
- **w_i**: Input-to-hidden weights
- **w_h**: Hidden-to-hidden (recurrent) weights
- **w_f**: Hidden-to-output weights
- here f is tanh
  - z = W*x+W*h+b
  - h_t = tanh(z) = (e^z - e^-z)/(e^z + e^-z)
  -

▼ *10. RNN **Backpropagation (Backpropagation through time BPTT)***

https://www.youtube.com/playlist?list=PLGP2q2bIgaNzBBpxxNUf126chLsQ20dfG

https://medium.com/@abhishekjainindore24/backpropagation-in-rnn-abd87b940a7d

1. **Architecture**



- **3 input nodes** (IP nodes)
- **1 hidden layer with 3 neurons**
- **1 output node** (O/P node)

Weights Calculation:

1. **From input layer to hidden layer:**

   Each of the 3 input neurons connects to 3 hidden neurons

   → 3×3=9 weights

2. **From hidden layer to output layer:**

   Each of the 3 hidden neurons connects to 1 output neuron

   → 3×1=3 weights

   So, total weights = (3×3)+(3×1)=9+3=12

Biases Calculation:

- **Each neuron (not input) typically has one bias term**
- 3 hidden neurons → 3 biases
- 1 output neuron → 1 bias

So, total biases =3+1=4

2. **Forward Pass**

Recurrent Neural Network (RNN) Forward Pass

Let:

- x_{1i}: input at time step i
- O_i: output at time step i
- W_i: weight matrix for input to hidden
- W_h: weight matrix for hidden to hidden
- W_o: weight for output
- b: bias term
- f: activation function (e.g., tanh, ReLU)

Equations per time step:

$$O_1 = f(x_{11} \cdot W_i + O_0 \cdot W_h + b)$$
$$O_2 = f(x_{12} \cdot W_i + O_1 \cdot W_h + b)$$
$$O_3 = f(x_{13} \cdot W_i + O_2 \cdot W_h + b)$$
$$\hat{y} = f(O_3 \cdot W_o + b)$$

Notes:

- O_0 is typically initialized to zeros or random values.
- This is a single-layer unrolled RNN for 3 time steps.
- The final prediction $\hat{y}$ is derived from the last hidden state.

3. Updating the weights

$$W_i = W_i - \eta \frac{\partial L}{\partial W_i}$$
$$W_h = W_h - \eta \frac{\partial L}{\partial W_h}$$
$$W_o = W_o - \eta \frac{\partial L}{\partial W_o}$$

Wi, Wh, Wo Now we need to calculate these 3 derivatives in order to do backpropagation.

4. Derivatives for Backpropagation

We have to minimize this loss by using gradient descent.

$$L = -y_i \log(\hat{y}_i) - (1 - y_i) \log(1 - \hat{y}_i)$$

Where:

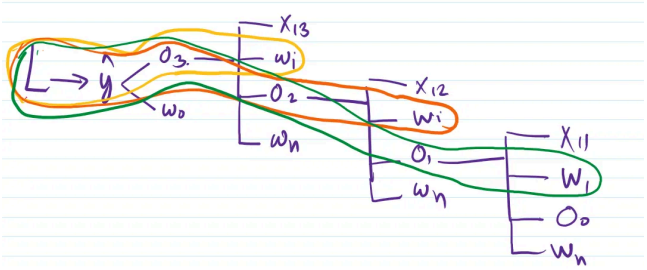
- L is the loss.
- y_i is the true label.
- y′_i is the predicted probability for the positive class.

Output Layer

$$\frac{\partial L}{\partial w_o} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_o}$$

Input Layer

CHAIN RULE OF DIFFERENTIATION



$$\frac{\partial L}{\partial w_i} = \sum_{j=1}^{3} \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial O_j} \cdot \frac{\partial O_j}{\partial w_i}$$

$$\frac{\partial \hat{y}}{\partial O_j} = \frac{\partial \hat{y}}{\partial O_3} \cdot \frac{\partial O_3}{\partial O_2} \cdot \frac{\partial O_2}{\partial O_1} \cdot \frac{\partial O_1}{\partial w_i}$$

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial O_3} \cdot \frac{\partial O_3}{\partial w_i} + \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial O_3} \cdot \frac{\partial O_3}{\partial O_2} \cdot \frac{\partial O_2}{\partial w_i} + \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial O_3} \cdot \frac{\partial O_3}{\partial O_2} \cdot \frac{\partial O_2}{\partial O_1} \cdot \frac{\partial O_1}{\partial w_i}$$

Hidden Layer

$$\frac{\partial L}{\partial w_h} = \sum_{j=1}^{m} \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial O_j} \cdot \frac{\partial O_j}{\partial w_h}$$

▼ *11. Representations of data: **Sparse** and **Dense***

In the context of feature vectors, especially in machine learning and data science, **sparse** and **dense** refer to two different types of representations of data.

Sparse Feature Vector

- **Sparse** vectors contain a large number of zeros. They are efficient for representing data with many missing or zero values.
- Typically used when the data has a lot of features, but only a few are non-zero. For example, in text data (like in TF-IDF), most words might not appear in a particular document, so most values in the vector will be zero.
- **Memory efficient**: Sparse representations save memory by storing only non-zero elements and their positions (for example, using data structures like a sparse matrix).
- Example: If you have 10,000 features (like words in a text corpus), but only 100 words are present in a document, the vector would be sparse with most values as 0.

Example of Sparse Vector:
`[0, 0, 0, 1, 0, 0, 0, 0, 0, 3]`

Here, the values represent the feature vector where only two non-zero values exist (1 and 3), and the rest are zeros.

Dense Feature Vector

- **Dense** vectors contain mostly non-zero values. Every element in the vector holds some information, making them "full" compared to sparse vectors.
- Dense representations are often used when the features have values for each entry or are derived from continuous data (like image pixels, deep learning embeddings, etc.).
- **Memory usage**: Dense vectors require more memory compared to sparse vectors because they store all values, including zeros.
- Example: In machine learning tasks where each feature has a value (like in regression problems or certain embeddings), the feature vector would be dense.

Example of Dense Vector:
`[0.2, 0.5, 0.9, 1.2, 0.4]`

Here, each element has a meaningful value, and there are no zeros in the vector.

Key Differences:

1. **Sparsity**: Sparse vectors contain many zeros, while dense vectors contain mostly non-zero values.
2. **Memory Efficiency**: Sparse vectors are more memory-efficient for high-dimensional data with many zeros, whereas dense vectors take up more memory.
3. **Use Cases**: Sparse vectors are typically used in situations like text data (TF-IDF, bag-of-words), while dense vectors are common in continuous data or embeddings (e.g., Word2Vec, embeddings in deep learning models).

Would you like an example to visualize how these representations are used in machine learning models?

▼ *12. **Embedding Layer***

Single word → Embedding Layer → Dense vector

Single word example: i, we, cat, semantic etc

Dense vector: [0.05 -0.11 0.23] **3-dimensional dense vector of cat**

An embedding layer is a neural network layer that transforms categorical data (like words) into dense, continuous vector spaces where similar items are closer together.

Embeddings are low-dimensional, dense vector representations of data (like words, sentences, or items) that capture their semantic meaning.

They are typically used in natural language processing (NLP) and machine learning to transform high-dimensional, sparse data (like one-hot encoded vectors) into compact, meaningful numerical representations.

For example, **word embeddings** (e.g., Word2Vec, GloVe) map words with similar meanings to vectors that are close together in the vector space.

In short: **Embeddings = Efficient, dense vectors that capture complex relationships in data.**

Advantages of Embedding Layer

- Reduces dimensionality compared to one-hot encoding
- Learns task-specific word representations
- Captures semantic word relationships
- Efficient and scalable for large vocabularies
- Enables use of pretrained embeddings
- Improves model generalization and performance
- Reduces risk of overfitting with dense vectors

Input Parameter

input_dim: (Vocabulary size + padding) or (number of unique token +1).

```
output_dim: Each word is embedded into a ''output_dim'' dimensional dense embedding vector.
input_length: Number of feature or timesteps or X_i.
trainable: Whether the embeddings are trainable or fixed.
embeddings_initializer: Initializer for the embedding matrix (e.g., 'uniform', 'normal').

embedding_layer = Embedding(<<Input Parameter>>)  # Set to True to enable fine-tuning
model.add(embedding_layer)
```

```
# Fine-Tuning Pre-trained Embeddings in Keras
embedding_layer = Embedding(input_dim=vocab_size,
                            output_dim=embedding_dim,
                            weights=[embedding_matrix],
                            input_length=50,
                            trainable=True)  # Set to True to enable fine-tuning

# Add this embedding layer to your model
model = Sequential()
model.add(embedding_layer)
model.summary()
```

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense

input_dim=len(tokenizer.word_index)+1 # Vocabulary size + padding
output_dim=5 # Each word is embedded into a ''output_dim'' dimensional vector. embedding size
input_length=padded_sequences.shape[1] # Number of feature, timesteps
input_shape=(input_length, output_dim) # (timesteps, features)
print("Input Dimension:", input_dim)       # Vocabulary size or number of unique tokens
print("Output Dimension:", output_dim)     # Embedding vector size for each token
print("Input Length:", input_length)       # Length of input sequences (number of tokens)

model = Sequential()
model.add(Embedding(input_dim=input_dim,output_dim=output_dim,input_length=input_length))
model.add(SimpleRNN(units=64, input_shape=input_shape ,return_sequences=True))
model.add(SimpleRNN(units=16, return_sequences=False))
model.add(Dense(1, activation='sigmoid'))

model.build(input_shape=(None, input_length))  # force model to build
model.summary()
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
# Total params  embedding layer=input_dim*output_dim
# Output Shape : (Num of row, input_length, output_dim)
```

Pre-trained embeddings: Word2Vec, GloVe, FastText

These are embeddings that have been pre-trained on massive text corpora like Wikipedia or news articles.

▼ *13. RNN code*

The `oov_token` (short for **Out-Of-Vocabulary token**) is a special token used by the Keras `Tokenizer` to handle words **not seen during training**.
When you use a tokenizer with a vocabulary limit (e.g., `num_words=10000`), any word **outside** that top 10,000 most frequent words is considered "out-of-vocabulary."

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense

input_dim=len(tokenizer.word_index)+1 # Vocabulary size + padding
output_dim=5 # Each word is embedded into a ''output_dim'' dimensional vector. embedding size
input_length=padded_sequences.shape[1] # Number of feature, timesteps
input_shape=(input_length, output_dim) # (timesteps, features)
print("Input Dimension:", input_dim)       # Vocabulary size or number of unique tokens
print("Output Dimension:", output_dim)     # Embedding vector size for each token
print("Input Length:", input_length)       # Length of input sequences (number of tokens)

model = Sequential()
model.add(Embedding(input_dim=input_dim,output_dim=output_dim,input_length=input_length))
model.add(SimpleRNN(units=64, input_shape=input_shape ,return_sequences=True))
# Units: number of neuron in hidden state, number of sigmoid and tanh
model.add(SimpleRNN(units=16, return_sequences=False))
model.add(Dense(1, activation='sigmoid'))

model.build(input_shape=(None, input_length))  # force model to build
model.summary()
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
# Total params  embedding layer=input_dim*output_dim
# Output Shape : (Num of row, input_length, output_dim)
```

▼ *14. Vanishing and Exploding gradient problem*

Vanishing gradient problem

The **vanishing gradient problem** happens during training of deep neural networks, especially **RNNs**, when the **gradients become extremely small** as they are backpropagated through many layers or time steps, making training difficult.

What's Happening?

In training, we use **backpropagation** to update weights:

- It multiplies gradients layer-by-layer (or time-step-by-time-step in RNNs).
- If gradients are **< 1**, repeated multiplication causes them to **shrink toward zero**.
- As a result, **early layers (or earlier time steps) stop learning** → network forgets long-term patterns.

Why It's a Problem:

- **Slows learning** or makes it **impossible** to learn long-range dependencies.
- Common in **RNNs**, especially for long sequences (like full sentences or time-series).
- Makes the model focus only on **recent inputs**, ignoring earlier ones.

Solution:

- **Use LSTM/GRU**: Gated units maintain long-term dependencies better than vanilla RNNs.
- **ReLU or Leaky ReLU Activation**: Helps preserve gradient flow compared to tanh/sigmoid.
- **Proper Weight Initialization**: Use Xavier or He initialization to maintain gradient scale.
- **Batch Normalization**: Normalizes layer inputs to stabilize and maintain gradient flow.
- **Shorter Sequence Lengths**: Reduce time steps during training (e.g., using truncated BPTT).
- **Residual Connections**: Skip connections help gradients flow through deep networks.

Exploding gradient problem

The **exploding gradients** problem in Recurrent Neural Networks (RNNs) refers to a situation during training when the gradients of the loss function with respect to the model parameters become excessively large, often growing exponentially during backpropagation through time (BPTT).

What's Happening?

- RNNs share weights across time steps.
- Gradients are computed by multiplying many Jacobian matrices.
- If matrix norms > 1, gradients grow exponentially.
- Longer sequences worsen the effect during backpropagation.

Why It's a Problem:

- **Unstable Training**: The exploding values can cause the model's parameters to **oscillate wildly** or even become **NaN (Not a Number)**.
- **Loss Divergence**: The loss function may suddenly increase to very large values, making it impossible for the model to converge.
- **Poor Learning**: The model fails to learn meaningful representations or gets stuck in poor local minima.

Solution:

- **Gradient Clipping**: Limit gradient norm (e.g., `clip_norm = 5.0`) to stabilize updates.
- **Weight Regularization**: Use L2 regularization to penalize large weights.
- **Gated Architectures**: Use LSTM or GRU instead of vanilla RNNs.
- **Weight Initialization**: Apply Xavier or He initialization for balanced gradients.
- **Smaller Learning Rates**: Reduce learning rate to avoid large parameter updates.

▼ *15. LSTM **(Long Short-Term Memory)***

https://www.youtube.com/playlist?list=PLGP2q2bIgaNzBBpxxNUf126chLsQ20dfG

https://www.geeksforgeeks.org/deep-learning-introduction-to-long-short-term-memory/

https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/

https://medium.com/@anishnama20/understanding-lstm-architecture-pros-and-cons-and-implementation-3e0cca194094

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

https://dwbi1.wordpress.com/2021/08/07/recurrent-neural-network-rnn-and-lstm/

We need **LSTM (Long Short-Term Memory)** over standard **RNN** because **RNNs struggle with long-term dependencies** due to a problem called the **vanishing gradient**.
Problem with RNN:

- When sequences are long (like paragraphs of text), RNNs **forget early inputs**.
- During training, the gradients shrink and become too small to update earlier weights → **can't learn long-term relationships**.

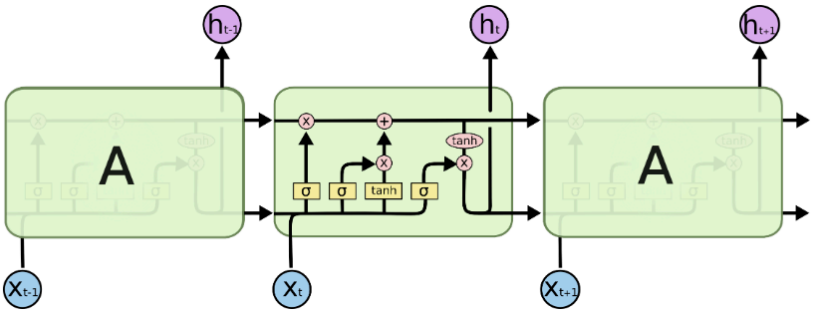
Why LSTM?

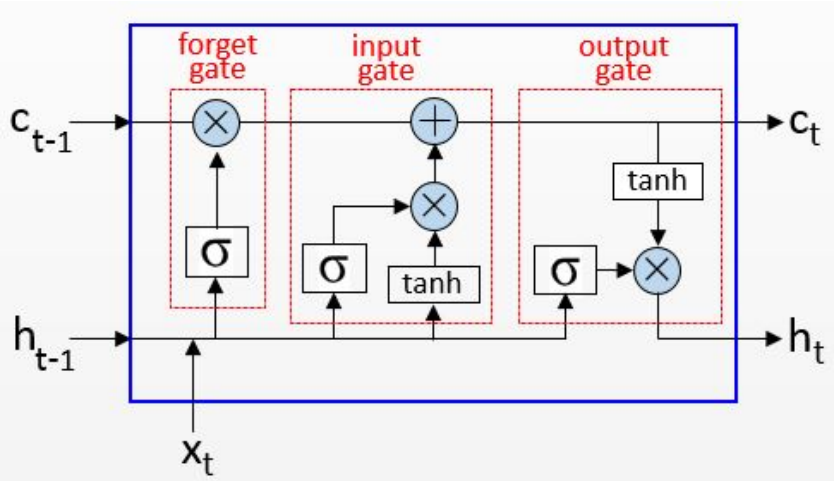
LSTM was designed to **remember information for long periods**.

It introduces:

Component	Role
Cell State	Acts like a conveyor belt — memory flow that persists across time.
Gates	Control how much information to keep, forget, or output.
- Forget Gate	Decides what to discard from memory.
- Input Gate	Decides what new info to store.
- Output Gate	Decides what to pass to the next step.

LSTM Architecture





1. **Neural Network Layer**

   A Layer of **Neural Network each containing same number of neuron with activation** Sigmoid or Tanh.

   Number of neuron we can set as hyperparameter

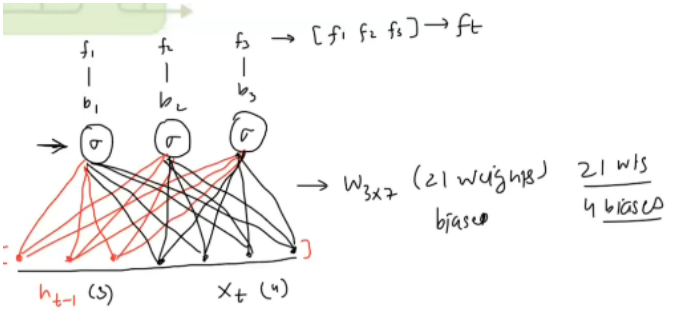
2. **Pointwish Operation**

   Vector 1 : [1 3 6]

   Vector 2 : [2 4 3]

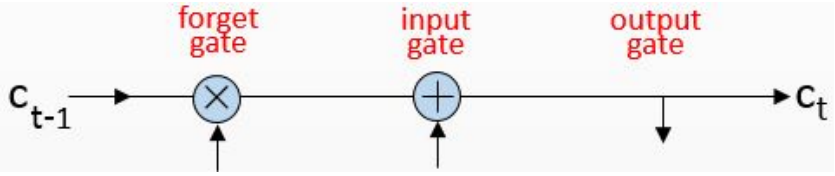
   1. Dot Product : [2 12 18]
   2. Addition Product : [3 7 9]

3. **Vector Transfer** : Flow of data (tensor) from one component to another.
4. **Concatenate** : Merges multiple vectors or tensors along a specified axis.



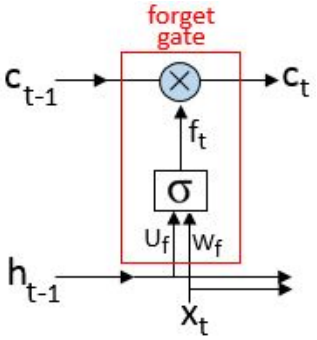
```
lstm_out1 = [0.1, 0.2, 0.3, 0.4]     # shape: (4,)
lstm_out2 = [0.5, 0.6, 0.7, 0.8, 0.9, 1.0]  # shape: (6,)
merged = lstm_out1 + lstm_out2
merged = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
```

5. **Code** : Splits or duplicates the output to be used by multiple subsequent operations.
6. **Cell Memory or Long Term Memory**



The horizontal line at the top (from c_t-1 to c_t) is the cell state.

- In an LSTM (Long Short-Term Memory) diagram, the **horizontal line at the top** (from c_t−1 to c_t) represents the **cell state**.
- This cell state acts as the **short-term memory** of the LSTM.
- Along this line, **three main operations** take place:
  1. **Forget Gate**: The cell state is **multiplied** by the forget gate output to **remove irrelevant information** from the previous state (c_t−1).
  2. **Input Gate**: The cell state is then **updated** (increased or reduced) based on new information via the input gate.
  3. **Output Gate**: Finally, the updated cell state contributes to the **output** of the LSTM through the output gate.
7. **Forget Gate**



$$f_t = \sigma (W_f x_t + U_f h_{t-1} + b_f)$$
$$c_t = c_{t-1} \circledast f_t$$

- The forget gate **removes unwanted information** from the cell state c_t−1.
- The gate outputs a value f_t between **0 and 1** using a **sigmoid function (σ)**.
- This controls **how much of the previous cell state** should be retained.
- where b_f is the bias
- W_f weight matrix for input x_t
- U_f weight matrix for previous hidden state h_t−1
- The blue circle with cross is element wise multiplication.
- t is the current time slot and t-1 is the previous time slot.
- Notice that h and x have their own weights.

Alternate Equivalent Form:

Alternatively, it's also valid to write:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$

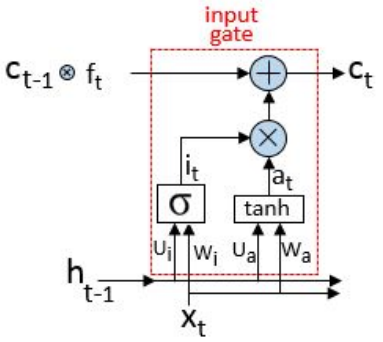
Where:

- $[h_{t-1}, x_t]$: denotes the **concatenation** of h_{t-1} and x_t
- W_f: a **single weight matrix** that combines both W_f and U_f

Interpretation:

- If f_t[i]≈0 The LSTM "forgets" the i-th part of the cell state.
- If f_t[i]≈1 The LSTM retains the i-th part of the cell state.
- Initial `c` and `h` are tensor of zeros with shape `(num_layers, batch_size, hidden_size)`.
- These initial values are used when processing the **first time step** of the sequence. As the LSTM processes input through time steps, it updates both `c` and `h` at each step.
- `c`: Helps carry long-term memory.
- `h`: Carries output information to the next time step or layer. Short term memory

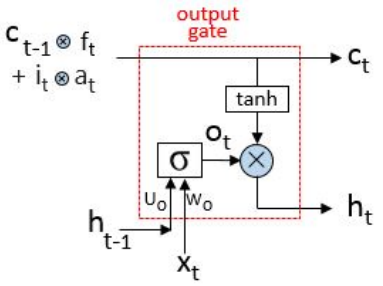
8. **Input Gate**



$$i_t = \sigma\left(W_i x_t + U_i h_{t-1} + b_i\right)$$
$$a_t = \tanh\left(W_a x_t + U_a h_{t-1} + b_a\right)$$
$$c_t = c_{t-1} \otimes f_t + i_t \otimes a_t$$

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

9. **Output Gate**



$$o_t = \sigma\left(W_o x_t + U_o h_{t-1} + b_o\right)$$
$$h_t = \tanh(c_t) \circledast o_t$$

All formulas

1. Forget Gate

Controls what information to discard from the previous cell state.

- **Formula:**

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

- **Cell state update (partial):**

$$c_t = f_t \odot c_{t-1} + (\text{update from input gate})$$

- **Used variables:**
  f_t: forget gate output
  c_t: current cell state

2. Input Gate

Controls what new information to add to the cell state.

- **Gate activation:**

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

- **Candidate (new content):**

$$a_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

- **Cell state update (combined with forget gate):**

$$c_t = f_t \odot c_{t-1} + i_t \odot a_t$$

- **Used variables:**
  i_t: input gate
  a_t: candidate cell content
  c_t: updated cell state

3. Output Gate

Controls what part of the cell state to output as the hidden state.

- **Gate activation:**

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

- **Hidden state (output):**

$$h_t = o_t \odot \tanh(c_t)$$

- **Used variables:**
  o_t: output gate
  h_t: hidden state (output)

Gate	Purpose
Forget Gate	Decides what information from the **previous cell state** (c_t−1) should be **discarded**. Helps the model forget irrelevant or outdated information.
Input Gate	Determines **how much new information** (from current input x_t and previous hidden state h_t−1) should be **added to the cell**

Gate	Purpose
	state.
Output Gate	Controls **how much of the cell state** (c_t) should be **exposed** to the **next layer or time step** via the hidden state h_t.

▼ *16. GRU (**Gated Recurrent Unit**)*

Need of GRU Over LSTM

- GRU has a simpler architecture than LSTM.
- It uses fewer gates (2 vs. 3), reducing complexity.
- Requires fewer parameters, making it faster.
- Trains quicker and is computationally efficient.
- Often performs equally or better on smaller datasets.
- Suitable for real-time and resource-constrained environments.

Cons of GRU (Gated Recurrent Unit):

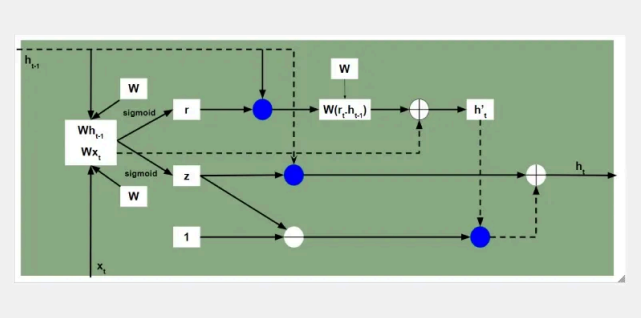
- Less effective at capturing very long-term dependencies.
- May overfit on small datasets.
- Less interpretable due to complex gating.
- Not always better than LSTM across tasks.
- Fewer tuning resources and community support than LSTM.

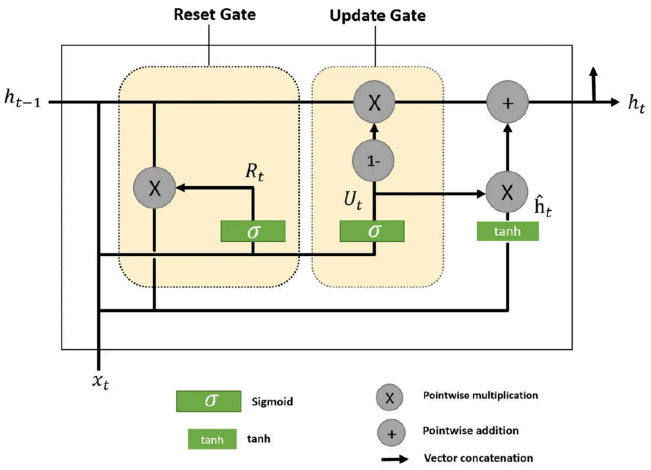
LSTM vs GRU

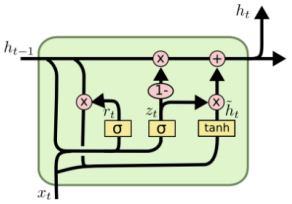
Feature	LSTM (Long Short-Term Memory)	GRU (Gated Recurrent Unit)
Architecture	Complex: 3 gates (input, forget, output)	Simpler: 2 gates (reset, update)
Memory Cell	Has a separate memory cell	Merges hidden and cell state
Training Time	Slower due to more parameters	Faster due to fewer parameters
Performance	Better for complex and longer sequences	Comparable or better on shorter sequences
Overfitting	Higher risk with small datasets	Less prone to overfitting
Parameter Count	More	Fewer
Suitability	Better for tasks requiring long-term memory	Better for real-time and small devices

Characteristics of GRU:

- The number of unit (neurons) is the same in all three neural networks.
- The vectors h_t−1, h_t, x_t, r_t, z_t, and h~_t have the same size as the number of units.



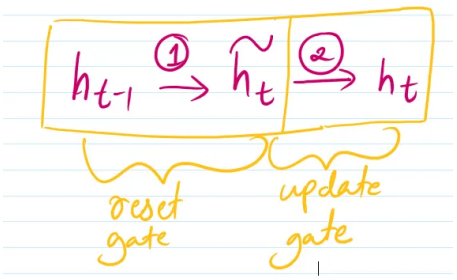




$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

ALL OF BELOW ARE VECTORS

- h_t−1 : hidden state for previous timestamp
- h_t : hidden state for current timestamp
- x_t : current input
- r_t : reset gate
- z_t : update gate
- h'_t : candidate hidden state



Steps to calculate values

1. **Update Gate (z_t)**
   - Decides how much of the past information to keep and how much to update with the new input.
   - A value close to 1 → keep the past information; close to 0 → update with new information.
   - *z_t or u_t=σ* (*W_z*·[*h_t*−1,*x_t*] + b)
2. **Reset Gate (r_t)**
   - Controls how much of the previous hidden state to forget.
   - A value close to 0 → forget the previous hidden state; close to 1 → remember it.
   - *r_t=σ* (*W_r*·[*h_t*−1,*x_t*] + b)
3. **Candidate Activation ($\tilde{h}_t$)**

- This is the "new" hidden state candidate, created using the reset gate.
- If the reset gate is zero, the previous hidden state is ignored.
- $h'_t = \tanh(W_h \cdot [r_t \cdot h_{t-1}, x_t] + b)$

4. **Final Hidden State (h_t)**
   - Combines the previous hidden state and the new candidate using the update gate.
   - This linear interpolation helps in deciding whether to use the old or new state.
   - $h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot h'_t + b$

GRU Architecture

The GRU architecture consists of the following components:

1. **Input layer:** The input layer takes in sequential data, such as a sequence of words or a time series of values, and feeds it into the GRU.
2. **Hidden layer:** The hidden layer is where the recurrent computation occurs. At each time step, the hidden state is updated based on the current input and the previous hidden state. The hidden state is a vector of numbers that represents the network's "memory" of the previous inputs.
3. **Reset gate:** The reset gate determines how much of the previous hidden state to forget. It takes as input the previous hidden state and the current input, and produces a vector of numbers between 0 and 1 that controls the degree to which the previous hidden state is "reset" at the current time step.
4. **Update gate:** The update gate determines how much of the candidate activation vector to incorporate into the new hidden state. It takes as input the previous hidden state and the current input, and produces a vector of numbers between 0 and 1 that controls the degree to which the candidate activation vector is incorporated into the new hidden state.
5. **Candidate activation vector:** The candidate activation vector is a modified version of the previous hidden state that is "reset" by the reset gate and combined with the current input. It is computed using a tanh activation function that squashes its output between -1 and 1.
6. **Output layer:** The output layer takes the final hidden state as input and produces the network's output. This could be a single number, a sequence of numbers, or a probability distribution over classes, depending on the task at hand.

Gated Recurrent Unit | Deep Learning | GRU | CampusX

▼ *17. Deep RNN*

https://www.youtube.com/watch?v=mlDkTrILaio&list=PLKnIA16_RmvYuZauWaPlRTC54KxSNLtNn&index=66&t=12s
https://medium.com/@kramiknakrani100/understanding-deep-recurrent-neural-networks-rnns-with-keras-29138b17e2dc

https://medium.com/@abhishekjainindore24/deep-rnns-stacked-rnns-stacked-lstms-79cf652f451a

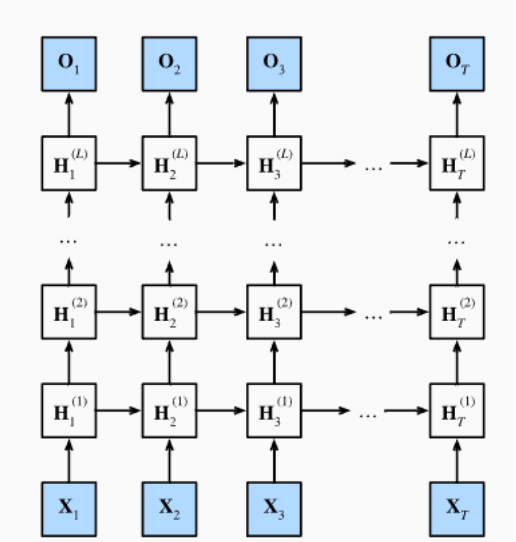


Fig. 10.3.1 Architecture of a deep RNN.

A **Deep Recurrent Neural Network (Deep RNN)** is an advanced form of Recurrent Neural Network (RNN) that consists of **multiple layers of RNN cells**, rather than just a single layer. This depth allows the network to learn more abstract and hierarchical features from sequential data.

Why Use Deep RNNs

- Capture complex sequential dependencies across time steps
- Learn hierarchical temporal features through multiple layers
- Improve performance on tasks like NLP, speech, and time-series analysis
- Provide richer representations of input sequences
- Scale better with large datasets needing higher model capacity

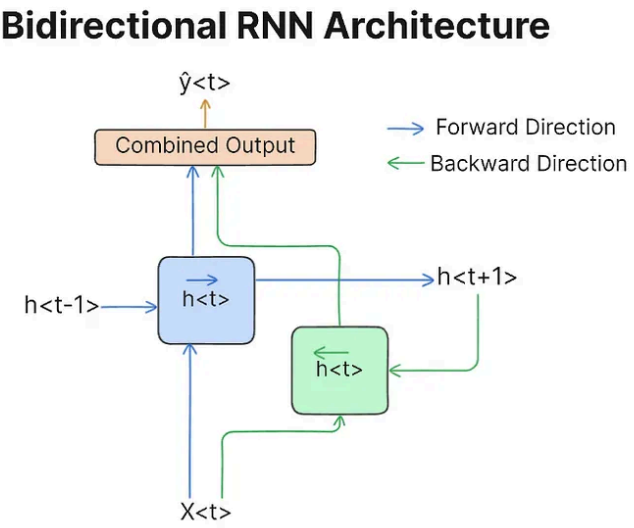
Topic: When to Use Deep RNNs

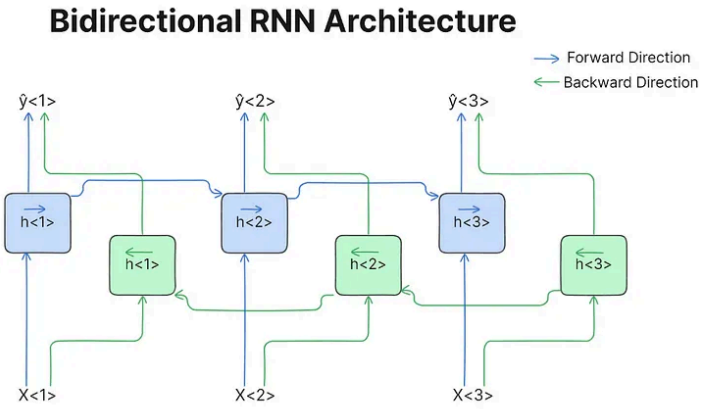
- Task involves complex or long-term sequential dependencies
- A single-layer RNN underfits or lacks performance
- Sufficient training data is available to support deeper models
- Adequate computational resources are present for training
- Goal is to boost accuracy in tasks like NLP, speech recognition, or forecasting

```
model = Sequential()
model.add(Embedding(max_features, 128))
model.add(SimpleRNN(128, return_sequences=True))
# return_sequences=True, must be true
# then only information flow one layer to another layer
model.add(SimpleRNN(128))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
```

▼ *18. **Bidirectional RNN (BiRNN/LSTM/GRU)***

https://www.youtube.com/watch?v=k2NSm3MNdYg&list=PLKnIA16_RmvYuZauWaPlRTC54KxSNLtNn&index=67





A bidirectional recurrent neural network (RNN) is a type of recurrent neural network (RNN) that processes input sequences in both forward and backward directions.

This enables the network to utilize both past and future context when making predictions.

Example:
Apple is my favorite _____, and I am going to buy one.

Forward RNN Input (left to right)	Apple is my favorite _____, and I am going to buy on
Backward RNN Input (right to left)	one buy to going am I and _____ favorite my is Apple

Use Bidirectional RNNs

- Full input sequence is available (not real-time).
- You need context from **both past and future**.
- Tasks like:
  - Text classification
  - Named Entity Recognition (NER)
  - Part-of-Speech (POS) tagging
  - Sentiment analysis
  - Question answering

- Speech recognition (offline)

Advantages:

- **Full Context Understanding**: Access to both past and future inputs improves prediction accuracy.
- **Better Performance**: Often outperforms unidirectional RNNs in sequence labeling tasks.
- **Improved Disambiguation**: Helps resolve ambiguity in meaning (e.g., "Apple" as fruit vs company).
- **Useful for NLP tasks**: Essential in tasks like NER, POS tagging, and sentiment analysis.

Disadvantages:

- **Not suitable for real-time applications**: Requires the full input sequence up front.
- **Higher computation cost**: Two RNNs (forward & backward) double the parameters and processing time.
- **Memory intensive**: Uses more memory compared to unidirectional RNNs.
- **Slower inference**: Less efficient for tasks needing quick responses.

Code

```
from tensorflow.keras.layers import Dense, SimpleRNN, Bidirectional

model = Sequential([
    Bidirectional(SimpleRNN(16), input_shape=input_shape),
    Dense(3, activation='softmax')  # for 3-class classification
])
```

Formulas

Notation:

- $x^{\langle t \rangle}$: input at time step t
- $\overrightarrow{h}^{\langle t \rangle}$: hidden state from forward RNN at time t
- $\overleftarrow{h}^{\langle t \rangle}$: hidden state from backward RNN at time t
- $y^{\langle t \rangle}$: output at time step t
- W, b: weights and biases (different for forward and backward)

Forward RNN equations:

$$\overrightarrow{h}^{\langle t \rangle} = \tanh(W_{xh}^{\rightarrow} x^{\langle t \rangle} + W_{hh}^{\rightarrow} \overrightarrow{h}^{\langle t-1 \rangle} + b_h^{\rightarrow})$$

Backward RNN equations:

$$\overleftarrow{h}^{\langle t \rangle} = \tanh(W_{xh}^{\leftarrow} x^{\langle t \rangle} + W_{hh}^{\leftarrow} \overleftarrow{h}^{\langle t+1 \rangle} + b_h^{\leftarrow})$$

> Note: The backward RNN moves in the reverse direction — it takes the next hidden state from the future.

Concatenated hidden state:

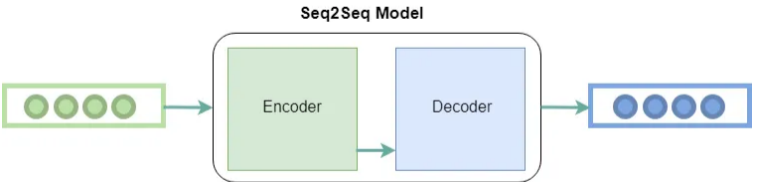
$$h^{\langle t \rangle} = \begin{bmatrix} \overrightarrow{h}^{\langle t \rangle} \\ \overleftarrow{h}^{\langle t \rangle} \end{bmatrix}$$

Output at time step t:

$$y^{\langle t \rangle} = f(W_y h^{\langle t \rangle} + b_y)$$

- f is typically a softmax or sigmoid depending on the task.

▼ *19. Sequence-to-Sequence Architecture (Seq2Seq)*



Data Type or Problem Type	Solution Approach	Example Problem	Input	Output
Tabular Data	ANN, ML Algorithms	Predicting house prices using structured datasets	Structured data (CSV, Excel)	Price (numeric prediction)
Image Data	CNN (Convolutional Neural Net)	Classifying handwritten digits (e.g., MNIST dataset)	Image	Class label (digit 0–9)
Sequence Data	RNN, LSTM	Sentiment analysis of a review	Sentence	Sentiment (Positive/Negative)
Sequence-to-Sequence	Seq2Seq Models, Transformer	Language translation (e.g., English to French)	Sentence in source language	Sentence in target language

Sequence-to-Sequence (Seq2Seq or many-to-many) problems is a special class of Sequence Modelling Problems in which both, the input and the output is a sequence.

Advantages of Seq2Seq Models

- Handles variable-length input and output sequences efficiently.
- Suitable for tasks like machine translation, summarization, and chatbots.
- Learns contextual relationships within sequences using encoder-decoder architecture.
- Can be enhanced with attention mechanisms for better performance.
- Supports end-to-end training with gradient descent.
- Language-agnostic; works across multiple languages with proper training.
- Scalable to large datasets and adaptable to different domains.

Disadvantages of Seq2Seq Models

- **Loss of Long-Term Dependencies**: Struggles with remembering long input sequences, especially in vanilla RNNs.
- **Fixed Context Vector Bottleneck**: Compressing all input into a single vector can limit model performance.
- **High Computational Cost**: Training and inference can be resource-intensive.

- **Data-Hungry**: Requires large labeled datasets for effective performance.
- **Difficult to Interpret**: Acts as a black box without clear reasoning for predictions.
- **Error Propagation**: Mistakes made early in the output sequence can affect later predictions.
- **Limited Generalization**: May not generalize well to out-of-distribution or unseen inputs.

Applications of Seq2Seq Models

- **Machine Translation**: Translating text from one language to another (e.g., English to French).
- **Text Summarization**: Producing concise summaries from longer documents.
- **Chatbots & Conversational Agents**: Generating relevant and coherent responses.
- **Speech Recognition**: Converting spoken language into text.
- **Image Captioning**: Generating text descriptions for input images (with encoder as CNN).
- **Code Generation**: Translating natural language instructions into code.
- **Question Answering**: Producing answers from context in natural language.
- **Grammatical Error Correction**: Generating corrected text from incorrect input.

Evaluation Criteria for Seq2Seq Models

- **BLEU (Bilingual Evaluation Understudy) Score**:

  Measures n-gram overlap between predicted and reference sequences; widely used in machine translation.
- **ROUGE (Recall-Oriented Understudy for Gisting Evaluation)**:

  Common in summarization tasks; evaluates recall and overlap of n-grams, sequences, or word pairs.
- **METEOR (Metric for Evaluation of Translation with Explicit ORdering)**:

  Considers synonymy and stemming; better at capturing semantic meaning than BLEU alone.
- **Perplexity**:

  Evaluates how well a probability model predicts a sample; lower perplexity indicates better performance.
- **Accuracy**:

  Useful in classification-like Seq2Seq tasks (e.g., chatbot intent matching or grammar correction).
- **Edit Distance (Levenshtein Distance)**:

  Measures the number of operations needed to convert the predicted sequence into the target sequence.
- **Human Evaluation**:

  Often used for tasks involving naturalness, fluency, and coherence, especially in dialogue systems.

▼ *20. **Encoder Decode***

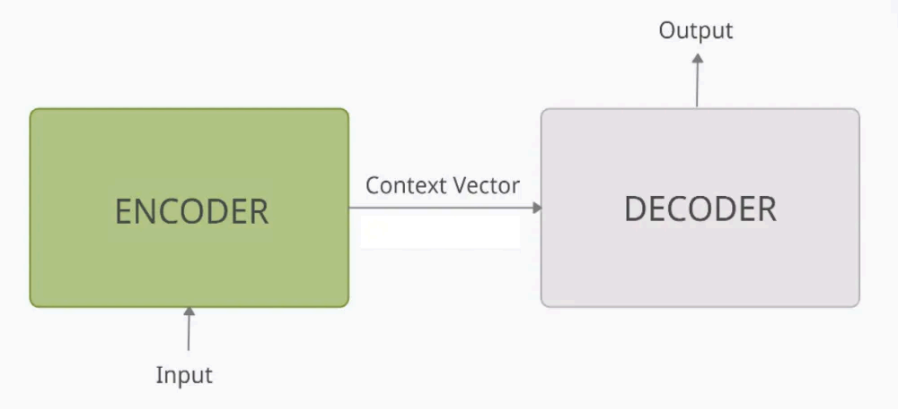
Language translation dataset : https://www.kaggle.com/datasets/preetviradiya/english-hindi-dataset, https://www.kaggle.com/datasets/vaibhavkumar11/hindi-english-parallel-corpus, https://www.kaggle.com/datasets/sayedshaun/english-to-bengali-for-machine-translation

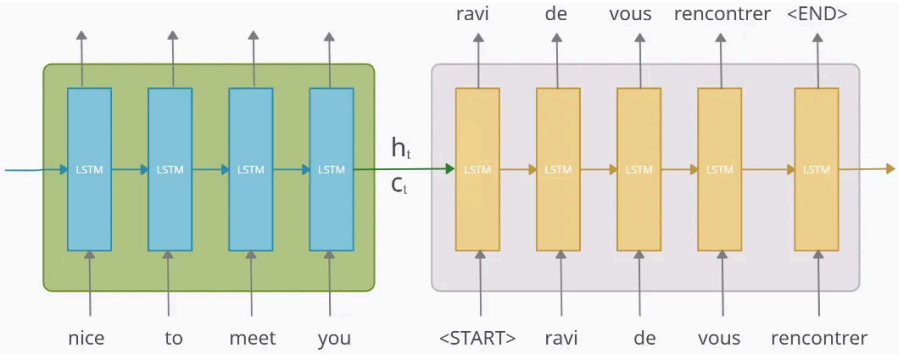
Sequence to Sequence Learning with Neural Networks: https://arxiv.org/pdf/1409.3215

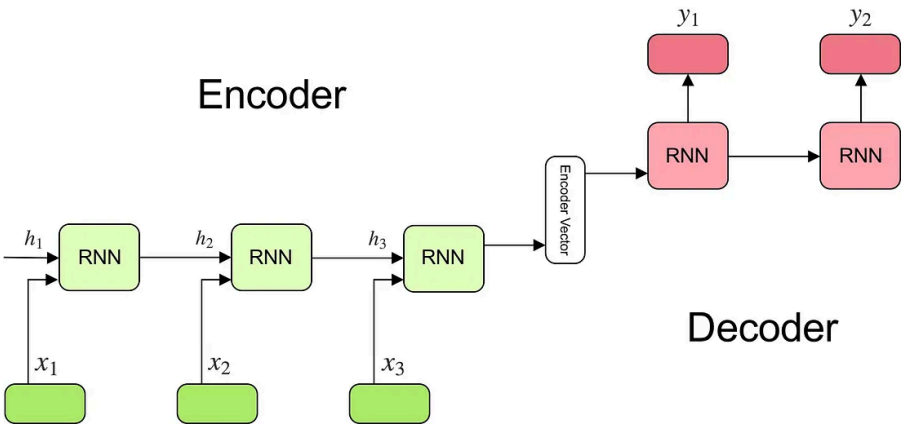
NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE: https://arxiv.org/pdf/1409.0473

https://www.youtube.com/watch?v=KiL74WsgxoA&list=PLKnIA16_RmvYuZauWaPlRTC54KxSNLtNn&index=69&t=27s

https://medium.com/analytics-vidhya/encoder-decoder-seq2seq-models-clearly-explained-c34186fbf49b







Working of Encoder Decode

Input → Encoder → Context Vector (C_t+H_t) → Decoder (with Teacher Forcing) → Output

1. Encoder Block

- Processes the input sequence using RNN or LSTM units.
- Encodes the sequence into internal states:
  - **Hidden state (h_t)**
  - **Cell state (c_t)**
- Captures all relevant information from the input.
- Final internal states are passed to the decoder to assist in generating the output sequence.

2. Context Vector

- Formed after the encoder processes the complete input sequence.
- Represents a compressed summary of the input data.
- Composed of:

- **h_t (final hidden state)**
- **c_t (final cell state)**

- Acts as the memory of the input sequence for the decoder.

3. Decoder Block

- Uses the context vector (h_t, c_t) from the encoder.
- Generates the output sequence step by step.
- Typically built using RNN or LSTM units.
- The generation begins using the initial states provided by the encoder.

4. Teacher Forcing

- A training technique used to enhance model performance.
- During training, the actual target token is fed as the next input.
- Replaces the decoder's own prediction with the ground truth at each time step.
- Helps the model converge faster and produce more accurate results.

Step-by-Step Working of Encoder-Decoder

- **X (Input Sentence)**: "nice to meet you"
- **Y_true (Target Sentence)**: "ravi de vous rencontrer"
- **Goal**: Learn to translate the English sentence into French using an Encoder-Decoder model.

1. Preprocessing

- **Tokenization**:
  - `X_tokenized = ['nice', 'to', 'meet', 'you']`
  - `Y_tokenized = ['<sos>', 'ravi', 'de', 'vous', 'rencontrer', '<eos>']`
    (<sos> = start of sentence, <eos> = end of sentence)
- **Vocabulary Lookup**: Each word is converted into an integer using a word-index mapping (embedding follows).

2. Embedding Layer (Encoder)

- Converts each token in X into a dense vector representation:

  'nice' → x1 : [1 0 0 0]

  'to' → x2 : [0 1 0 0]

  'meet' →x3 : [0 0 1 0]

  'you' → x4 : [0 0 0 1]

3. Encoder (e.g., LSTM or GRU)

- Processes the input sequence one token at a time.
- Produces a sequence of hidden states.
- Only the **last hidden state** is passed to the decoder as **context**:

4. Decoder Initialization

- Takes the **context vector (c)** and uses it to initialize its hidden state.
- Starts decoding using the `<sos>` token.

  Decoder Step-by-Step Generation

Time Step	Hidden State (from previous)	Input Token	Output (Predicted Word)
t=1	context vector (from encoder)	`<sos>`	"ravi"
t=2	decoder hidden state	"ravi"	"de"
t=3	decoder hidden state	"de"	"vous"

Time Step	Hidden State (from previous)	Input Token	Output (Predicted Word)
t=4	decoder hidden state	"vous"	"rencontrer"
t=5	decoder hidden state	"rencontrer"	<eos>

5. Loss Calculation

- Multiclass use cross-entropy loss

7. Backpropagation & Update

- Gradients from loss are backpropagated through both the decoder and encoder.
- Model parameters (embeddings, encoder weights, decoder weights) are updated using optimizers like Adam.

Encoder-Decoder Mathematical Formulas (RNN used)

1. Encoder (RNN/LSTM/GRU)

Given input sequence $X = (x_1, x_2, ..., x_T)$:

- **Hidden State Update** (for RNN):

$$h_t = f(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

where:

- h_t = hidden state at time t
- x_t = input vector at time t
- f = activation function (e.g., tanh)
- W_{xh}, W_{hh} = weight matrices
- b_h = bias

- Final encoder hidden state (context vector):

$$c = h_T$$

2. Decoder (RNN/LSTM/GRU)

The decoder generates output sequence $Y = (y_1, y_2, ..., y_{T'})$ given context vector c:

- **Initial hidden state**:

$$s_0 = c$$

- **Hidden State Update**:

$$s_t = f(W_{ys}y_{t-1} + W_{ss}s_{t-1} + b_s)$$

- **Output Prediction**:

$$\hat{y}_t = \text{softmax}(W_{so}s_t + b_o)$$

3. Loss Function

- **Cross-Entropy Loss** between predicted $\hat{y}_t$ and true output y_t:

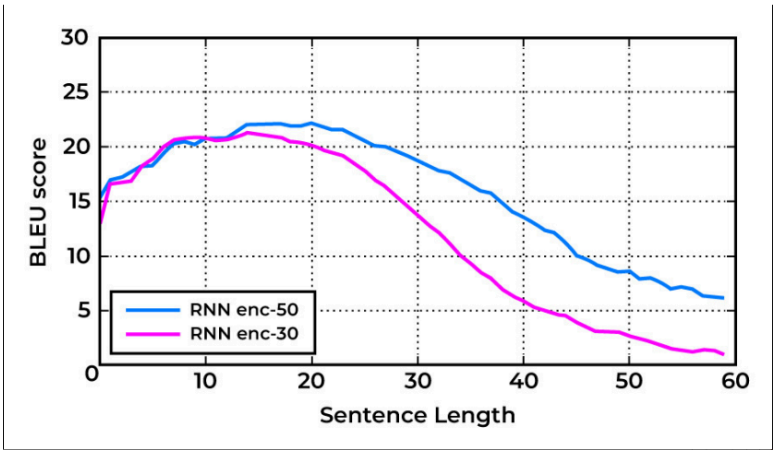
$$\mathcal{L} = -\sum_{t=1}^{T'} y_t \log(\hat{y}_t)$$

Improvements in Encoder-Decoder Models

- **Add Embedding Layer**: Converts sparse word indices into dense vector representations for better semantic understanding.
- **Use Deep Architecture**: Stack multiple LSTM/GRU layers in encoder and decoder to capture complex patterns.
- **Reverse Input Sequence**: Helps model learn alignment between input and output more effectively (especially in early models).
- **Apply Attention Mechanism**: Allows the decoder to focus on relevant parts of the input at each time step, improving translation quality.
- **Use Bidirectional Encoder**: Captures both forward and backward context for richer input representation.
- **Incorporate Pre-trained Embeddings**: Improves performance with semantic knowledge from large corpora (e.g., GloVe, Word2Vec).
- **Use Transformer Architecture**: Replaces RNNs with self-attention for faster training and better long-range dependency modeling.
- **Scheduled Sampling**: Mixes ground-truth and predicted tokens during training to reduce exposure bias.
- **Use Layer Normalization and Dropout**: Enhances model stability and prevents overfitting.

Problems with Encoder-Decoder Models

- **Single Context Vector Bottleneck**: Traditional encoder-decoder models use only the last hidden state of the encoder (a fixed-length context vector) to represent the entire input sequence, which works for short sentences but fails to capture the semantics of longer sentences.



- **Loss of Information in Long Sequences**: As sequence length increases, important contextual information may get compressed or lost, leading to degraded translation or output quality.
- **Gradient Vanishing Issues**: Especially in RNN-based models, gradients may vanish or explode during backpropagation through long sequences.
- **Error Accumulation**: Mistakes in earlier decoder steps often lead to cascading errors in the following outputs.

- **Insufficient Focus on Important Words**: All parts of the input are treated equally, so critical parts of a sentence may be underutilized or ignored.
- **Low BLEU Score for Long Sentences**: BLEU score, a common evaluation metric, often drops significantly when translating long or complex sentences.
- **No Direct Alignment Between Input and Output**: It's difficult to track which input tokens influenced which output tokens.

Solution: **Attention Mechanism**

- Dynamically generates a context vector at each decoding step by weighing encoder outputs.
- Allows the model to "attend" to relevant parts of the input sequence during decoding.
- Improves performance, particularly on longer sequences, and increases interpretability.

▼ *21. **Attention Mechanism***

https://erdem.pl/2021/05/introduction-to-attention-mechanism

Sequence to Sequence Learning with Neural Networks: https://arxiv.org/pdf/1409.3215

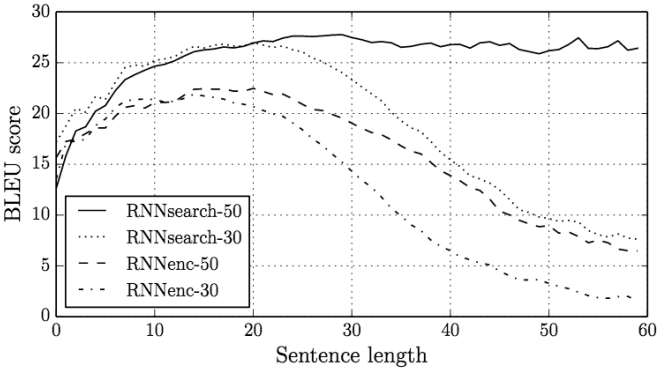
NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE: https://arxiv.org/pdf/1409.0473

Deep Learning(CS7015): Lec 15.3 Attention Mechanism: https://www.youtube.com/watch?v=yInilk6x-OY

What is an Attention Mechanism?

An **Attention Mechanism** is a technique in neural networks that dynamically focuses on the most relevant parts of the input data when making predictions, allowing the model to weigh different parts differently rather than treating them all equally.

Need of Attention Mechanism:



- Seq2Seq models perform poorly for sentences longer than 25 words, as shown by the declining BLEU scores. The Attention Mechanism addresses this by enabling the model to focus on relevant parts of the input, improving performance for long sentences.
- To handle long-range dependencies efficiently.
- To focus on important parts of the input while ignoring irrelevant ones.
- To overcome the limitations of fixed-size context in RNNs and CNNs.
- To improve performance in tasks like machine translation, image captioning, and more.
- To enable parallel processing in models like Transformers.

How Does the Attention Mechanism Work (Bahdanau Attention)

NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE: https://arxiv.org/pdf/1409.0473

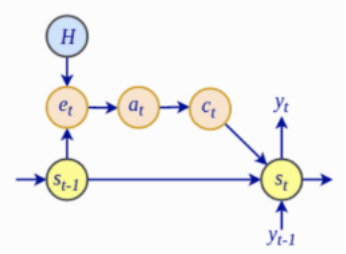
All formulas

Task: Machine Translation

Data: {x_i = source, y_i = target}, Here i = 1 to T

Component: **Bidirectional RNN Unit in Encoder, Normal RNN Unit Decoder**

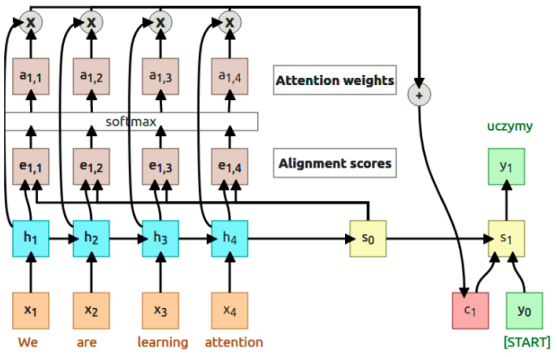
here **RNN Unit refer LSTM/GRU**



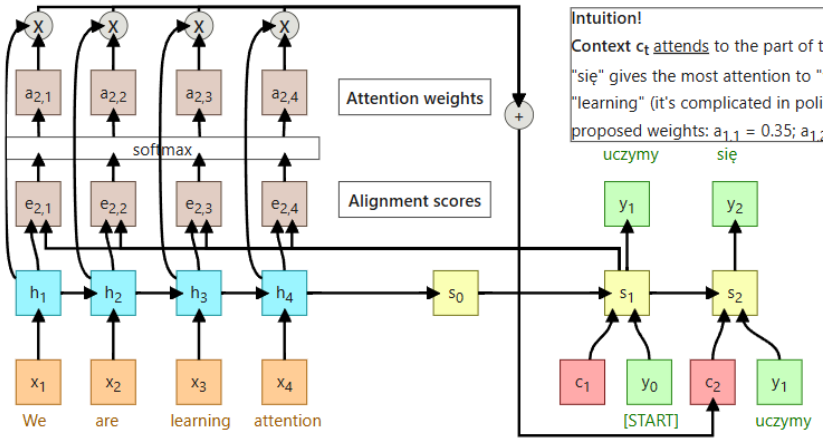
1. X = [we, are, learning, math] = [x_1, x_2, x_3, x_4]
2. Encoder Hidden state
   h_t=RNN (h_t-1, x_t),
3. Final Encoder Hidden state (Context vector)
   s_0 = tanh (W [h_T ; h_1'])
   Here T mean I/P seq (i = 1 to T) completed
   h_1' is backword RNN o/p
4. Alignment score
   e_t,i = ANN (s_t-1, h_i),
   timestamp = t, I/P = x_i
5. Attention weights
   a_t,i = Softmax (e_t,i) = exp(e_t,i) / sum (all alignment score)
6. Context vector
   c_t=sum (a_t,i * h_i)
7. Decoder Compute the first output (Decoder Hidden State)
   s_t = RNN ([y_t-1; c_t], s_t-1) = tanh (W_s * s_t-1 + W_y * y_t-1 + W_c * c_t+b)
8. Predicted word
   l_t = Softmax (V*s_t+b)
   y_t=arg max(l_t)
   V: weight matrix mapping the hidden state to vocabulary logits.
   l_t: a vector where each element l_t,i represents the probability of the i^th word in the vocabulary being the next word in the sequence.
9. Repeat from 4
   Continue decoding until the end-of-sequence (EOS) token is generated.

Architecture

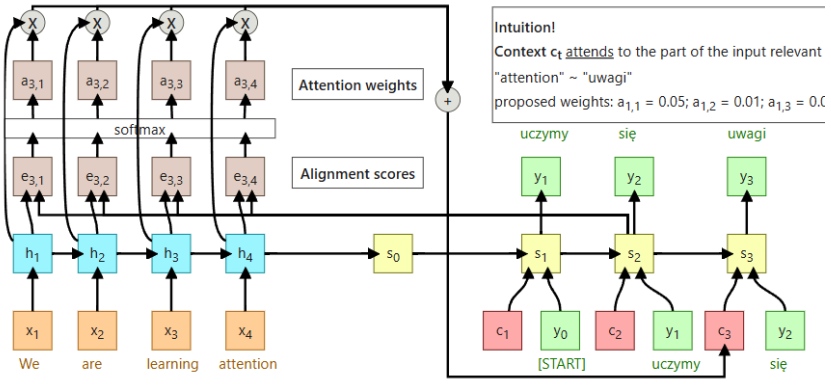
t=1



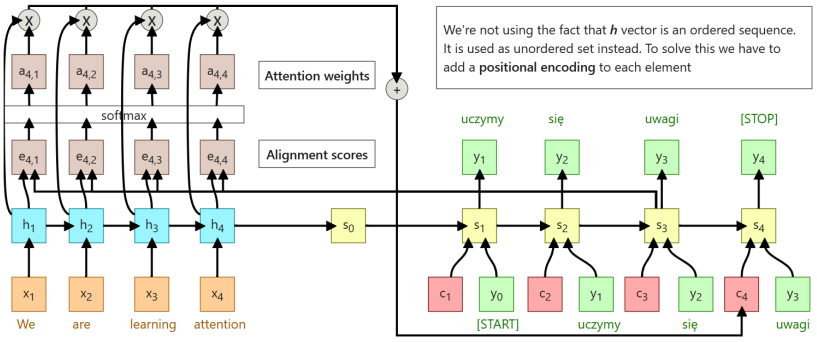
t=2

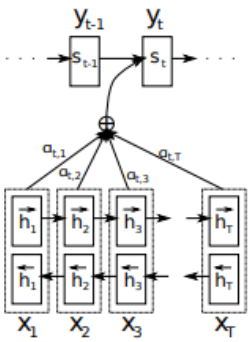

Attention weights
softmax
Alignment scores
Intuition!
Context c_t attends to the part of t
"się" gives the most attention to "
"learning" (it's complicated in poli
proposed weights: $a_{1,1}$ = 0.35; $a_{1,}$
uczymy
się
We
are
learning
attention
[START]
uczymy

t=3


Attention weights
softmax
Alignment scores
Intuition!
Context c_t attends to the part of the input relevant
"attention" ~ "uwagi"
proposed weights: $a_{1,1}$ = 0.05; $a_{1,2}$ = 0.01; $a_{1,3}$ = 0.0
uczymy
się
uwagi
We
are
learning
attention
[START]
uczymy
się

t=4

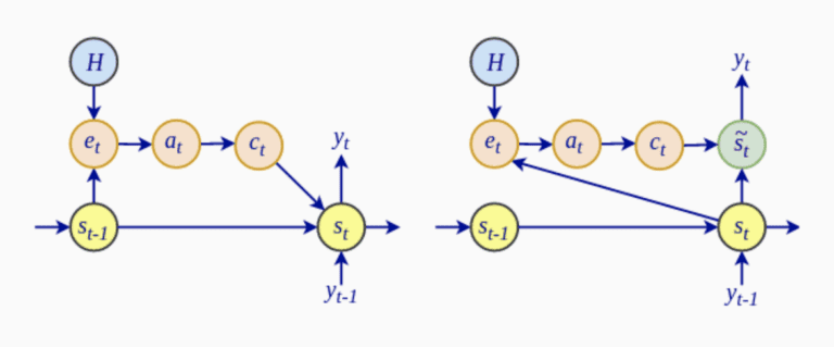

Attention weights
softmax
Alignment scores
We're not using the fact that *h* vector is an ordered sequence. It is used as unordered set instead. To solve this we have to add a **positional encoding** to each element
uczymy
się
uwagi
[STOP]
We
are
learning
attention
[START]
uczymy
się
uwagi



Luong Attention

Effective Approaches to Attention-based Neural Machine Translation: https://arxiv.org/pdf/1508.04025

Luong Attention is faster and simpler, suitable when efficiency is crucial.
Bahdanau Attention is more flexible and powerful but computationally expensive.
Moreover same as **Bahdanau Attention**



Bahdanau architecture (left) vs. the Luong architecture (right)

1. Task: Machine Translation
2. Input Sequence Example X (same as Bahdanau)
3. Encoder Hidden State h_t (same as Bahdanau)
4. Context vector s_0 (same as Bahdanau)
5. Decoder Hidden State Update s_t = RNN ([y_t-1; c_t-1], s_t-1)
6. Alignment Score
   - Dot Product: e_t,i = s_t * h_i
   - General: e_t,i = (final s_t) * W * h_i
   - Concat: e_t,i = ANN (s_t, h_i)
   - when to use

Alignment Score	Complexity	Speed	Flexibility	Best For
Dot Product	Low	Fast	Low (same dimensions)	Real-time tasks, low resources
General	Medium	Medium	Medium (different sizes)	Summarization, multilingual tasks
Concat	High (non-linear)	Slow	High (complex relationships)	Image captioning, story generation

7. Attention Weights: a_t (same as Bahdanau)
8. Context Vector: c_t (same as Bahdanau)
9. Concatenate Context Vector with Decoder State: s'_t = RNN (c_t, s_t)
10. Predicted Word
    - l_t = Softmax (V * s'_t + b)
    - y_t=arg max(l_t)
11. Repeat from Step 5:

- Continue decoding until the end-of-sequence (EOS) token is generated.

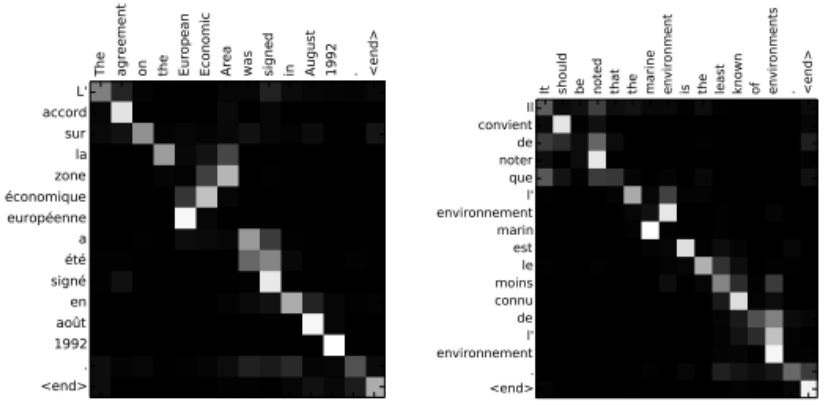
Types of Attention Mechanism

1. **Soft Attention:** Weighted sum of all inputs (differentiable, efficient).
2. **Hard Attention:** Selects one input element (non-differentiable, uses reinforcement learning).

3. **Self-Attention:** Computes relationships within a single sequence (core of Transformers).
4. **Global Attention:** Uses all input elements (suitable for sequence-to-sequence tasks).
5. **Local Attention:** Focuses on a small input window (efficient for long sequences).
6. **Multi-Head Attention:** Uses multiple attention heads for richer representation.
7. **Hierarchical Attention:** Applies attention at multiple levels (e.g., word and sentence).
8. **Spatial Attention:** Focuses on spatial regions (image processing).
9. **Channel Attention:** Emphasizes informative feature channels (vision tasks).
10. **Cross-Attention:** Links different sequences/modalities (e.g., text and image).

Applications of Attention Mechanism

- **Natural Language Processing (NLP)**: Translation, summarization, text classification, and QA.
- **Computer Vision**: Image captioning, object detection, and image classification.
- **Speech Processing**: Recognition, voice conversion, and speaker verification.
- **Healthcare**: Disease prediction, EHR analysis, and explainable medical diagnosis.
- **Recommendation Systems**: Personalized recommendations and content suggestion.
- **Reinforcement Learning**: Policy improvement and environment understanding.
- **Graph Neural Networks (GNNs)**: Node classification and graph representation learning.
- **Multimodal Learning**: Video-audio analysis and cross-modal retrieval.
- **Time Series Analysis**: Financial forecasting and anomaly detection.
- **Explainable AI (XAI)**: Model interpretability and feature attribution.



Advantages of Attention Mechanism:

- Enhances model performance by focusing on important information (**Selective Information Processing)**.
- Handles long-range dependencies efficiently.
- Enables parallel processing in Transformer models.
- Improves interpretability by visualizing attention weights.
- Reduces the problem of vanishing gradients in RNNs.

Disadvantages of Attention Mechanism:

- Computationally expensive for large inputs due to quadratic complexity.
- Requires significant memory during training.
- Can overfit when data is limited or noisy.
- May produce inconsistent attention (**Attention to Noise)** maps for similar inputs.

▼ *22. **Transformers: Self Attention Mechanism***

https://medium.com/analytics-vidhya/a-deep-dive-into-the-self-attention-mechanism-of-transformers-fe943c77e654

https://colab.research.google.com/drive/1hXIQ77A4TYS4y3UthWF-Ci7V7vVUoxmQ#scrollTo=YLAhBxDSScmV

Contextual/Dynamic Meaning: The meaning of a word changes depending on the surrounding words in a sentence.

Example: “bank” in “river bank” vs. “money bank”.

Static Meaning: A fixed representation of a word, regardless of context.

Example: Word2Vec gives the same vector for “bank” in both sentences.

Context: Context refers to the surrounding information or circumstances that help determine the meaning or relevance of something, such as a word, sentence, or event.

Multiple Contexts: Multiple contexts refer to different situations or environments in which the same word, phrase, or event can have different meanings or implications.

Problems in Word Embedding

- **Context Ignorance**: Traditional embeddings (like Word2Vec, GloVe) give the same vector for a word regardless of context.
- **Out-of-Vocabulary (OOV)**: Cannot handle unseen words during training.
- **Static Representation**: Meaning of a word does not change with sentence structure.
- **Lack of Syntax & Semantics**: Limited ability to capture deep syntactic or semantic relationships.
- **Bias Propagation**: May inherit societal biases present in training data.

Need of Self-Attention

- Captures context by relating each word to all others.
- Handles long-range dependencies effectively.
- Enables parallel computation (faster training).
- Assigns dynamic importance to different words.
- Scales well in Transformer-based models.

What is Self Attention(Also called scaled dot-product attention)

Self-attention is a mechanism that allows a model to weigh and relate different parts of the same input sequence to better understand context and meaning.

Self-attention is a technique used in NLP that helps models to understand relationships between words or entities in a sentence, no matter where they appear.

It is called **"self"** attention because the model attends to **different positions within the same input** — i.e., it relates a word to **other words in the same sequence** (including itself) to compute a richer representation.

Components of the Self-Attention Block

https://osakuadeopeyemi.medium.com/query-key-and-value-in-self-attention-34bbe6fabc75

- **Query (Q)**: Represents what the current word is looking for in the other words (a kind of question vector).
  - Imagine you’re at a library. The **query** is what you’re searching for (the topic you’re interested in).
  - Think of the query as a search request. It’s like a question you’re asking: “Which other words (or tokens) in the sentence are important for this word?”
- **Key (K)**: Represents what each word contains (like a label or identity).
  - The **keys** are like labels or tags on the books in the library, describing what they’re about.

    - The key helps determine how relevant each word is to the query. It's like a label on each word that helps match the query with the right information.
- **Value (V)**: Contains the actual information or content of each word.
    - The **values** are the content inside the books.
    - The value is the actual information you want to retrieve once you know which words are important. After matching the query and key, the value tells you what to focus on or pay attention to.

All Q, K, and V vectors are generated by multiplying the input embeddings with different learnable weight matrices.

- Every word in a sentence has its own query, key, and value.
- The transformer calculates how much each word should pay attention to other words in the sentence by comparing the queries and keys, and then it uses the values to adjust the word's representation.

Self Attention(Step by step) Working of single-head Self Attention

1. `Input: sentence`
2. `Tokenization`
3. `Word embeddings : X`
4. `linear transformation: Compute the Query(Q), Key(K), and Value(V) Matrices`
    - `Three different weight matrices W_q , W_k , and W_v , which are learnable parameters.`
        - `Query(Q) = X * W_q`
        - `Key(K) = X * W_k`
        - `Value(V)  = X * W_v`
5. `Calculate Attention Scores: the model calculates how much attention each word should pay to every other word in the sequence.`
   `Attention Scores = Q * transpose(K)`
6. `Scaling the Scores: In high-dimensional spaces, raw dot products can have large variances, potentially destabilizing training.`
   `Scaled Scores = Attention Scores / sqrt{d_k}`
   `d_k is dimensionality of the key vector.`
7. `Apply the Softmax Function`
   `Attention Weights = softmax (Scaled Scores) = softmax (Q * transpose(K)/sqrt{d_k})`
8. `Weight the Values/Final vector: The attention weights are used to compute a weighted sum of the value vectors V.`
   `Output = Attention Weights * Value(V)`

Step-by-step example

1. Input: sentence: **"The cat sat."**
2. Tokenization: **The, cat, sat**
3. Word embeddings : X

```
"The" → [0.1, 0.3]
"cat" → [0.4, 0.5]
"sat" → [0.7, 0.9]
```

4. **Linear Transformations**

   Assume:

   - W_Q = [[1, 0], [0, 1]]
   - W_K = [[0.5, 0], [0, 0.5]]
   - W_V = [[1, 1], [1, 1]]

   Apply these to each embedding: `X * W`

Token	Q (Query)	K (Key)	V (Value)
The	[0.1, 0.3]	[0.05, 0.15]	[0.4, 0.4]
Cat	[0.4, 0.5]	[0.2, 0.25]	[0.9, 0.9]
Sat	[0.7, 0.9]	[0.35, 0.45]	[1.6, 1.6]

5. **Calculate** Attention Scores: Attention Scores = Q * transpose(K)

   Dot product of Q for current word with K of all words.

   For "The":

   - Q•K("The") = 0.1×0.05 + 0.3×0.15 = 0.05
   - Q•K("cat") = 0.1×0.2 + 0.3×0.25 = 0.095
   - Q•K("sat") = 0.1×0.35 + 0.3×0.45 = 0.155

   Same to others

From / To	The (K)	Cat (K)	Sat (K)
The (Q)	0.050	0.095	0.155
Cat (Q)	0.065	0.205	0.335
Sat (Q)	0.120	0.355	0.595

6. **Scaling the Scores:** Attention Scores / sqrt{d_k}

   Given that each Q and K vector has 2 elements, d_k=2

   1/sqrt{d_k}=0.7071

From / To	The (K)	Cat (K)	Sat (K)
The (Q)	0.050 × 0.7071 ≈ **0.0354**	0.095 × 0.7071 ≈ **0.0672**	0.155 × 0.7071 ≈ **0.1096**
Cat (Q)	0.065 × 0.7071 ≈ **0.0450**	0.205 × 0.7071 ≈ **0.1450**	0.335 × 0.7071 ≈ **0.2369**
Sat (Q)	0.120 × 0.7071 ≈ **0.0849**	0.355 × 0.7071 ≈ **0.2511**	0.595 × 0.7071 ≈ **0.4205**

7. **Apply the Softmax Function**

   For "The"
   Values: [0.0354, 0.0672, 0.1096]

Exponentials:	e^0.0354≈1.0360	e^0.0672≈1.0696	e^0.1096≈1.1157

   Sum: 1.0360+1.0696+1.1157=3.2213

Softmax:	1.0360/3.2213 ≈0.3216	1.0696/3.2213 ≈0.3320	1.1157/3.2213 ≈0.3464

   Final Softmax Attention Weights

From / To	The (K)	Cat (K)	Sat (K)
The (Q)	0.3216	0.3320	0.3464
Cat (Q)	0.3015	0.3332	0.3653
Sat (Q)	0.2794	0.3297	0.3910

8. **Weight the Values/ Final vector**

   Output = Attention Weights * Value(V)

   Output for "The"

   Weights: [0.3216, 0.3320, 0.3464]

   =0.3216×[0.4,0.4]+0.3320×[0.9,0.9]+0.3464×[1.6,1.6]

   =[0.1286,0.1286]+[0.2988,0.2988]+[0.5542,0.5542]

   =[0.9816,0.9816]

Token	Attention Output
The	[0.9816, 0.9816]
Cat	[1.0050, 1.0050]
Sat	[1.0341, 1.0341]

Key challenges of single-head self-attention

- Unable to capture multiple perspectives or contexts of a single word. This limitation is addressed in **multi-headed attention**, where each head captures a different perspective. In the Transformer architecture, 8 heads are typically used.
- **Limited Expressiveness:** A single attention head can only focus on one type of relational pattern at a time, restricting the richness of the learned representation.
- **Insufficient Context Capture:** It may miss capturing multiple dependencies in parallel (e.g., long-range vs. short-range), limiting its ability to understand complex structures. It focuses on global dependencies across all tokens but it may not effectively capture local patterns. This can cause inefficiencies when local context is more important than global context.
- **No Subspace Diversity:** All attention computations are confined to a single projection space, reducing the model's ability to extract diverse semantic features.
- **Bottleneck in Learning:** It can act as a learning bottleneck, especially in deep models, due to limited attention capacity per layer.
- **Susceptibility to Overfitting:** With fewer learned parameters and patterns, it may overfit on training data more easily compared to multi-head setups.

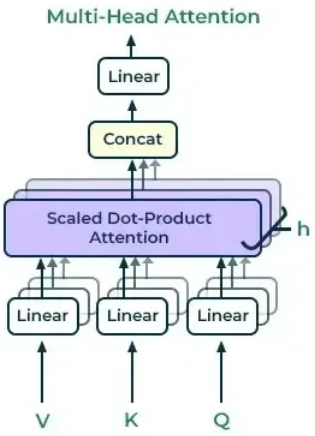
Why Multi-Headed Attention?

Self-attention is a key mechanism in transformers that allows each token to attend to every other token in the sequence, helping the model capture contextual relationships.

Multi-Headed Attention is an *enhanced version* of self-attention that applies self-attention multiple times in parallel (with different learned projections) and combines the outputs.

- **Captures Diverse Relationships:** Each head learns different types of relationships (e.g., syntactic, semantic) by attending to different parts of the sequence.
- **Improved Representation Power:** Multiple heads project the input into different subspaces, enriching the learned representations.
- **Increased Model Capacity:** Enables the model to process multiple patterns simultaneously, improving expressiveness and generalization.
- **Better Gradient Flow:** Having multiple attention heads helps stabilize training and improves gradient flow.
- **Redundancy and Robustness:** Multiple heads provide redundancy, making the system more robust to overfitting or missing information.

Multi-head attention Working



Steps

1. **Input:** $X \in \mathbb{R}^{n \times d_{\text{model}}}$
2. **Linear projections per head:**
   - For head i, compute:

$$Q^i = XW_q^i, \quad K^i = XW_k^i, \quad V^i = XW_v^i$$

3. **Scaled Dot-Product Attention per head:**

$$\text{Attention}^i = \text{softmax}\left(\frac{Q^i(K^i)^T}{\sqrt{d_k}}\right)V^i$$

4. **(Optional) Dropout on attention weights**
5. **Concatenate outputs:**

$$\text{Concat} = \text{concat}(\text{Attention}^1, \ldots, \text{Attention}^h)$$

6. **Final linear projection:**

$$\text{Output} = \text{Concat} \times W_o$$

where

- i=1 to h, and h is the number of attention heads.
- W_o output weight matrix.
- The input and output have the same dimensionality.

Masked Multi-Head Attention

https://medium.com/analytics-vidhya/a-deep-dive-into-the-self-attention-mechanism-of-transformers-fe943c77e654

***Autoregressive*: In the context of deep learning, autoregressive models are a class of models that generate data points in a sequence by conditioning each new point on the previously generated points.**

Example: Predicts future stock prices based on a linear combination of past stock prices.

- **Training Time (Non-Autoregressive):** The model sees the entire output sequence at once, allowing for parallel processing.
- **Inference Time (Autoregressive):** The model generates one word at a time, using previously generated words as input to predict the next one.

The Transformer decoder is autoregressive at inference time and non-autoregressive at training time.

Need of Masking

- **Prevents future token access** in decoder (causal masking).
- **Ignores padding tokens** during attention (padding mask).
- **Maintains sequence integrity** in autoregressive tasks.
- **Ensures training stability** by avoiding irrelevant attention.
- **Supports complex input structures** like multi-segment data.

Types of Masking

Masking Type	Purpose	Where It's Used
Padding Mask	Prevents the model from attending to **padding tokens** in a batch of sequences.	Encoder, Decoder (both)
Look-Ahead Mask (Causal Mask)	Prevents the decoder from **"seeing the future"** by masking out future tokens.	Decoder (during training, in self-attention)
Key Padding Mask	Used to mask padding tokens in the **key** sequence so that attention doesn't focus on them.	In multi-head attention
Attention Mask (custom)	Allows applying **arbitrary masking** (e.g., span masking, selective attention).	Specialized models like BERT, T5, etc.

Working (Look-Ahead Mask)

I/P: ["I", "am", "fine"]

Positions: T1 T2 T3

This prevents future token leakage during training.

Token Position	Can Attend to	Masked Tokens
"I" (T1)	["I"]	["am", "fine"]
"am" (T2)	["I", "am"]	["fine"]
"fine" (T3)	["I", "am", "fine"]	None

Mask Matrix

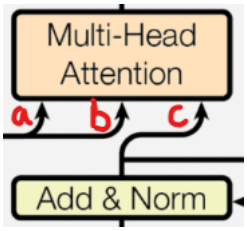
From \ To	I (T1)	am (T2)	fine (T3)
I	✅	❌	❌
am	✅	✅	❌
fine	✅	✅	✅

$$\text{Masked Scores} = \begin{cases} QK^T/\sqrt{d_k}, & \text{if no future tokens} \\ -\infty, & \text{if attending to future tokens} \end{cases}$$

- During self-attention, a **mask matrix** (same size as attention score matrix) is created.
- Elements corresponding to padded or future tokens are set to `-inf` so that their softmax becomes 0.

Cross Attention

It allows a model to focus on different parts of an input sequence when generating an output sequence.
Cross Attention is a mechanism used in the **decoder** of a Transformer model where the decoder **attends to the encoder's output** instead of attending only to itself.



Encoder output

a: key, b: value

Decoder hidden states

c: query

Why It's Important

- **Enables alignment**: The decoder knows *which parts of the input sequence* to focus on when generating each output token.
- **Critical for sequence transduction tasks** like translation, where the output depends heavily on the source.

▼ *23. Transformers: Positional Encoding (PE)*

https://blog.timodenk.com/linear-relationships-in-the-transformers-positional-encoding/

https://www.youtube.com/watch?v=GeoQBNNqIbM&list=PLKnIA16_RmvYuZauWaPlRTC54KxSNLtNn&index=80

Why is Positional Encoding Needed?

Due to parallel processing, the model does not retain the order of words, making it unable to capture the original meaning. For example, *"lion killed goat"* and *"goat killed lion"* are treated the same in self-attention. This issue is resolved using positional encoding.

Types of Positional Encoding

- **Sinusoidal Positional Encoding**

  Uses fixed sine and cosine functions of different frequencies. It allows the model to generalize to sequence lengths not seen during training.

- **Learned Positional Encoding**

  The model learns a set of positional embeddings during training, similar to how word embeddings are learned.

Sinusoidal Function

For a given position pos and embedding dimension i:

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

$$\mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

Where:

- pos: The position of the word in the sequence.
- i: The dimension index.
- d_model: The dimensionality of the embedding (e.g., 512).
- Even indices (2i) use sine, odd indices (2i+1) use cosine.

Working of Sinusoidal Positional Encoding

1. **Embeddings vector**

2. Compute Positional Encodings using **Sinusoidal Function**
3. Final Output: PE-Augmented Embeddings

   Final Output = Embedding + PE

Example working

1. Input Details
   - **Sentence**: "The cat sat."
   - **Tokens**: `"The"`, `"cat"`, `"sat"`
   - **Embeddings** (dimension = 3):
     - `"The"` → [0.1, 0.3, 0.5, 0.6]
     - `"cat"` → [0.4, 0.5, 0.3, 0.2]
     - `"sat"` → [0.7, 0.9, 0.3, 0.7]
2. Compute Positional Encodings using **Sinusoidal Function**

   a. d_model=4 (vector size)

   b. Find Pos

token	The	cat	sat
pos	0	1	2

   c. Find "i" (dimension index) = 0 to d_model-1

   i = 0 to 3

   d. Let's compute the **positional encoding vector**

$$\text{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$
$$\text{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

dimension index: i	Odd/Even Index	2*i/d_model	w_i=10,000^(2*i/d_model)	p=0	p=1	p=2
0	Even	2*0/4=0	1	sin(0/w_i)=0	sin(1/w_i)=0.84	sin(2/w_
1	Odd	2*1/4=0.5	100	cos(0/w_i)=1	cos(1/w_i)=0.99	cos(2/w
2	Even	2*2/4=1	10,000	sin(0/w_i)=0	sin(1/w_i)=0.00	sin(2/w_
3	Odd	2*3/4=1.5	1,000,000	cos(0/w_i)=1	cos(1/w_i)=0.99	cos(2/w

3. PE-Augmented Embeddings for `pos = 0` (token: The)
   Final Output = Embedding + PE

	i=0	i=1	i=2	i=3
"The"	0.1	0.3	0.5	0.6
PE	0	1	0	1
Final Embedding	0.1	1.3	0.5	1.6

   Calculate same for pos=1 and 2

Number of Curves

ach pair of dimensions (one sine, one cosine) corresponds to **one frequency** or "curve." So, if your model has dimensionality d, then:

Number of sinusoidal curves = d/2

Each curve corresponds to a unique wavelength, allowing the model to encode positional information at different granularities.

How the Frequency is Set:

The frequency of each sine/cosine wave is determined by the denominator:

1/10,000^2i/d

So the **frequency increases exponentially with iii**. That means:

- Lower dimensions (smaller i) encode **longer-range patterns** (low frequency).
- Higher dimensions (larger i) encode **fine-grained patterns** (high frequency).

Customizing Frequencies:

If you want to **customize** frequencies rather than use the 10,000 base, you can change it:

PE=sin(pos/base^2i/d)

Where:

- `base` controls the frequency range.
- Larger base → **slower variation**, useful for longer sequences.
- Smaller base → **faster variation**, better for short sequences.

How It Introduces Ordering:

- **Positional Encoding introduces ordering** by injecting information about the **position of each token** in the sequence into its embedding — something that plain Transformers can't do inherently since self-attention is **permutation-invariant**.
- Input to Transformer=Token Embedding+Positional Encoding
- This **augmented embedding** now contains information about **what** the token is and **where** it is in the sequence.
- Because sine and cosine are periodic and differentiable, the model can **learn to distinguish** not just **absolute positions**, but also **relative distances** between tokens.
- For example, certain patterns in the sine/cosine values will correspond to a token being earlier or later in the sequence.

▼ *24. Transformers: Layer Normalization*

https://www.youtube.com/watch?v=qti0QPdaeIg&list=PLKnIA16_RmvYuZauWaPlRTC54KxSNLtNn&index=80

https://www.geeksforgeeks.org/what-is-layer-normalization/

Need of Normalization

1. **Stable and Faster Training**: Normalization (like Layer Normalization) helps stabilize the training of deep neural networks, especially in transformers where deep architectures are common.
2. **Controls Gradient Flow**: It mitigates the issue of vanishing/exploding gradients by keeping activations within a consistent range.
3. **Improves Generalization**: Normalized representations help the model generalize better to unseen data, reducing overfitting.
4. **Essential in Residual Connections**: Transformers rely heavily on residual connections. LayerNorm ensures that outputs of these residual blocks remain well-scaled and meaningful.
5. **Prevents Internal Covariate Shift**: Normalization reduces shifts in the distribution of network activations, allowing each layer to learn more efficiently.
6. **Works Well with Attention Mechanism**: Attention-based architectures benefit from normalization to maintain numerical stability during dot-product computations.

Where to Apply Normalization in Transformer Models

- **Input Data Normalization**:
  - It's common to normalize input data (e.g., scaling pixel values, standardizing text embeddings) to ensure consistent value ranges before feeding into the model.
- **After Activation Functions (Within Layers)**:
  - Normalization can be applied after non-linear activations in intermediate layers to stabilize training.
  - **Batch Normalization**:
    - Common in CNNs but less effective in transformers due to variable sequence lengths and batch size dependencies.
    - Rarely used in transformer architectures.
  - **Layer Normalization** (Preferred in Transformers):
    - Applied **before or after** each sub-layer (e.g., self-attention, feedforward).
    - Independent of batch size and sequence length.
    - Two common placements:
      - **Post-Norm**: After residual addition.
      - **Pre-Norm**: Before sub-layer computation (recent models often prefer this for better gradient flow).

What is Layer Normalization

Layer Normalization is a technique to normalize the inputs across the features of each individual data sample rather than across the batch.

Advantages:

- Works well with variable-length sequences.
- Stable across different batch sizes (including batch size 1).
- Improves convergence and helps gradient flow in deep networks.

Working

1. Input

   $x1$=[3.0, 5.0, 2.0, 8.0]

   x2=[1.0, 3.0, 5.0, 8.0]

   x3=[3.0, 2.0, 7.0, 9.0]

   here x is output of neuron or activation function

2. **Compute Mean and Variance for Each Feature**

$$\mu = \frac{1}{H}\sum_{i=1}^{H} x_i$$

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

   where,

   H is the number of features (neurons) in the layer

   x_i is the input for each feature

   μ is mean

   σ^2 is variance (Population Variance Formula).

   Find out **Mean and Variance for x1**

   1. **Mean (μ1)**: (3.0+5.0+2.0+8.0)/4=18.0/4=4.5

2. **Variance (**σ^2)=[(3.0−4.5)^2+(5.0−4.5)^2+(2.0−4.5)^2+(8.0−4.5)^2]/4=21.0/4=5.25

same do it for others

Values	Mean	Variance
x1=[3.0, 5.0, 2.0, 8.0]	4.5	5.27
x2=[1.0, 3.0, 5.0, 8.0]	4.25	6.69
x3=[3.0, 2.0, 7.0, 9.0]	5.25	8.18

3. **Normalize the Input**

   x_i′ = (x_i - *μ)/sqrt(*σ^2+ ϵ)

   Here *ϵ* is a small constant added for numerical stability.

   small constant ϵ=1e−5

   calculate for x1

   x1 = [3.0, 5.0, 2.0, 8.0], Mean = 4.5, Variance = 5.27

   sqrt(5.27 + 1e-5) ≈ 2.296

   Normalized values:
   x1′= [(3.0 - 4.5)/2.296,

   (5.0 - 4.5)/2.296,

   (2.0 - 4.5)/2.296,

   (8.0 - 4.5)/2.296]

   x1′= [-0.6538, 0.2179, -1.0897, 1.5256]

Values	Mean	Variance	**Normalize values**
x1=[3.0, 5.0, 2.0, 8.0]	4.5	5.27	x1′=[−0.65, 0.21, −1.09, 1.52]
x2=[1.0, 3.0, 5.0, 8.0]	4.25	6.69	x2′=[−1.25, −0.48, 0.29, 1.45]
x3=[3.0, 2.0, 7.0, 9.0]	5.25	8.18	x3′=[-0.78, -1.13, 0.61, 1.31]

4. **Apply Scaling and Shifting(final output)**

   y_i=*γ*x1′ + *β*

   let's assume γ=1.5 and β=0.5 for all neuron, in original every neuron have different values

Normalize values	
x1′=[−0.65, 0.21, −1.09, 1.52]	y1=[−0.48, 0.82, −1.13, 2.79]
x2′=[−1.25, −0.48, 0.29, 1.45]	y2=[−1.38, −0.22, 0.93, 2.67]
x3′=[-0.78, -1.13, 0.61, 1.31]	y3=[−0.67, −1.20, 1.41, 2.46]

Why is Layer Normalization Used in Self-Attention Over Other Normalization Techniques?

1. Batch normalization operates on the mean, but if our data contains multiple zeros due to padding, it becomes difficult to compute an accurate mean value. This issue is addressed by layer normalization. Batch normalization works column-wise, whereas layer normalization works row-wise.

▼ *25. Transformers*

Attention Is All You Need: https://arxiv.org/pdf/1706.03762

A COMPREHENSIVE SURVEY ON APPLICATIONS OF TRANSFORMERS FOR DEEP LEARNING TASKS: https://arxiv.org/pdf/2306.07303

The Illustrated Transformer: https://jalammar.github.io/illustrated-transformer/

> https://github.com/afshinea/stanford-cme-295-transformers-large-language-models

Timeline of significant developments in machine learning and AI from 2000 to 2022. Here's a breakdown:

- **2000 - 2014**: The era of RNNs (Recurrent Neural Networks) and LSTMs (Long Short-Term Memory networks).
- **2014**: Introduction of the Attention mechanism.
- **2017**: The rise of Transformers.
- **2018**: Introduction of BERT (Bidirectional Encoder Representations from Transformers) and GPT (Generative Pre-trained Transformer), marking significant progress in transfer learning.
- **2018 - 2020**: Advancements in Vision Transformers and AI models like AlphaGo-2.
- **2021**: Growth in Generative AI (Gen AI).
- **2022**: Development of models like ChatGPT and Stable Diffusion.

What is Transformers

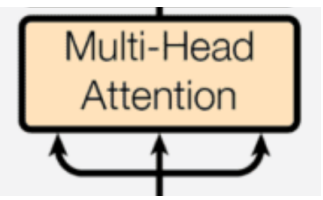
Transformers are a deep learning architecture based on **self-attention** mechanisms that allow models to process entire sequences (like text) in parallel, capturing relationships between all elements efficiently.

Need of Transformers

- Traditional RNNs process one word at a time — **slow and not scalable**.
- Transformers enable **parallel processing** for faster training.
- Handle **long-range dependencies** better via self-attention.
- Support **transfer learning** (pretrain → fine-tune).
- Achieve **state-of-the-art performance** in many NLP tasks.
- **No recurrence or convolution** – simpler and more efficient.
- **Versatile** – works across languages and modalities.

Architecture Components(small topic)

1. Residual Connection(**skip connection)**
   - The output of a sub-layer (like self-attention or feed-forward) is added to its original input.
     - Input + Output A → Residual A
   - Helps preserve gradients and combats vanishing gradients during training.
   - Allows the model to retain original features while learning new ones.
   - Type of connection in neural networks that allow information to bypass one or more intermediate layers.
2. Three Inputs to Multi-Head Attention



   Query (Q), Key (K), Value (V)
3. Feed-Forward Network

- Each token vector (512-dimensional) is passed
- 2 fully connected layers per FFN(MLP)
- I/P 4 Token

  → Number of stack: Num of token

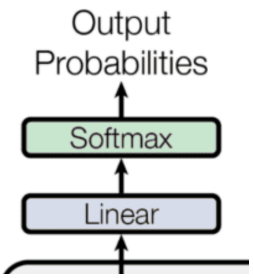
  Each stack contain one token size 512

  (Number of token*Embedding size) Matrix is passed

  → First Layer (2048 neurons)

  → Second Layer (512 neurons)
- ReLU Activation is used.
- Where
  - **d_model:** The **model dimension (size of token embeddings and all Transformer layer inputs/outputs)**
  - d**_ff:** The **feed-forward network dimension (size of the intermediate layer in the feed-forward network (usually 4 times d_model))**

4. Output Part



v: target language vocab count

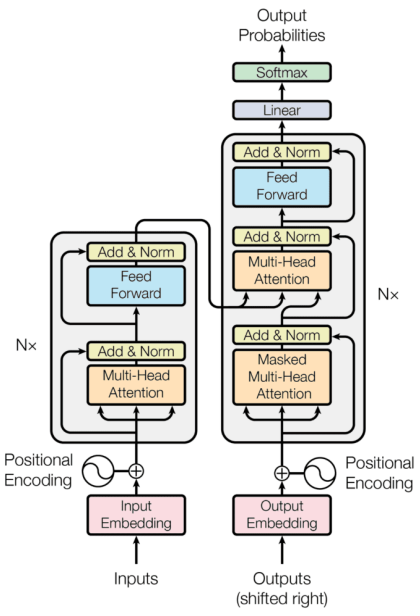
Decoder output size: 512

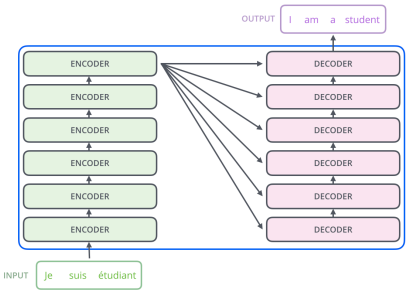
Linear layer contain: v number of neuron

Softmax: normalize the output, sum of all 1, give probability to each word

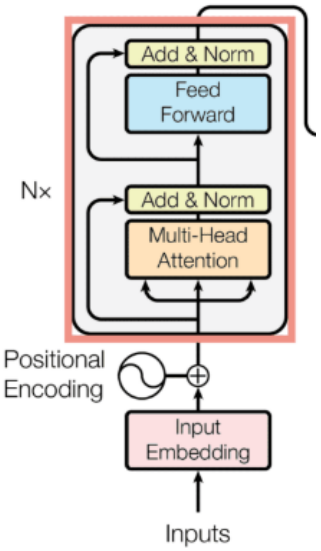
---

Transformers Architecture





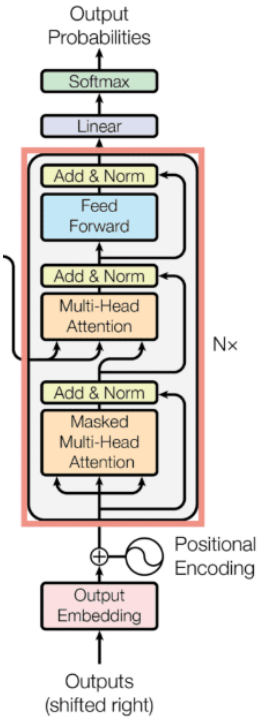
Encoder Part



1. **Input Embedding**
   - Converts input tokens into dense vectors.
   - Adds **positional encoding** to retain word order information.
2. **Encoder Layers (stacked N times)**

   Each layer contains:
   - **Multi-Head Self-Attention**
   - **Add & Layer Normalization**
   - **Feed-Forward Network**
   - **Add & Layer Normalization**
   - **Output to next layer(Context-rich representations passed to the decoder)**

Decoder Part



- **Decoder Stack** (repeated N times)

  Each layer contains:
  - **Masked Multi-Head Self-Attention** – prevents looking ahead.
  - **Add & Norm**
  - **Cross Attention**
  - **Multi-Head Attention** over encoder output.
  - **Add & Norm**
  - **Feed-Forward Network**
  - **Add & Norm**
- **Final Linear + Softmax**
  - Outputs probability distribution over vocabulary for each position.

Where N=6 for official “Attention Is All You Need” research paper

---

Inference (Prediction time)

Inference Time (Autoregressive): The model generates one word at a time, using previously generated words as input to predict the next one.

Advantages

- **Parallel Processing**: Processes sequences in parallel, unlike RNNs.
- **Captures Long-Range Dependencies**: Self-attention relates all words to each other.
- **Transfer Learning Friendly**: Pretrained models (BERT, GPT) can be fine-tuned.
- **Highly Scalable**: Works efficiently on GPUs/TPUs.
- **Language- and Task-Agnostic**: Same architecture works across tasks and languages.
- **State-of-the-Art Performance**: Excels in NLP, vision, and multimodal tasks.
- **No Recurrence**: Simpler and faster than RNN-based models.

Disadvantages

- **Computationally Expensive**: Self-attention has quadratic time/memory complexity.
- **Requires Large Datasets**: Needs a lot of data and compute to train effectively.
- **Harder to Interpret**: Self-attention is less intuitive than traditional models.
- **Not Ideal for Low-Resource Tasks**: Overkill for small datasets or simple tasks.
- **Length Limitations**: Input sequence size is often limited due to memory use.

Types of Transformer architectures

https://huggingface.co/docs/transformers/en/model_summary

1. **Encoder-Only Transformers**

- **Purpose:** Encoding input data (e.g., text classification).
- **Examples:** BERT (Bidirectional Encoder Representations from Transformers), RoBERTa.
- **Characteristics:** Focus on understanding input sequences using self-attention.

2. **Decoder-Only Transformers**

- **Purpose:** Generating text or sequences (e.g., text generation, chatbots).
- **Examples:** GPT (Generative Pre-trained Transformer), GPT-2, GPT-3.
- **Characteristics:** Autoregressive generation, predicting one word at a time.

3. **Encoder-Decoder Transformers (Seq2Seq)**

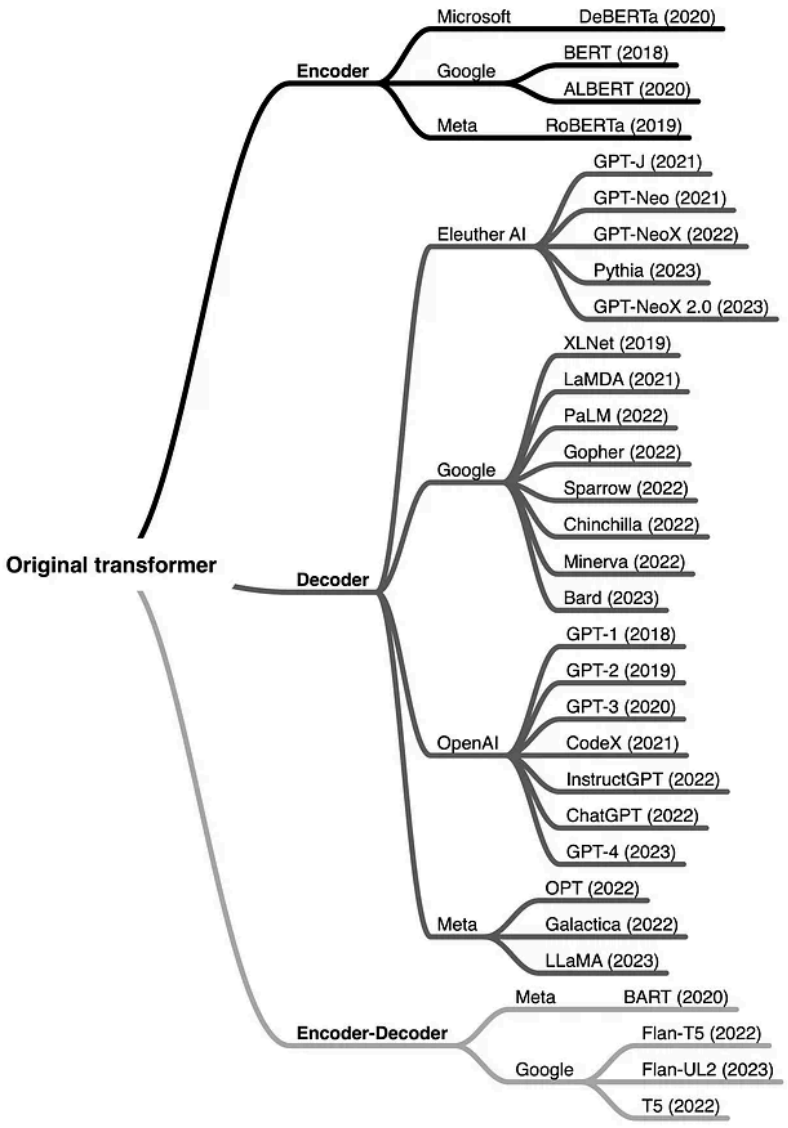
- **Purpose:** Tasks where input and output are both sequences (e.g., translation).
- **Examples:** T5 (Text-to-Text Transfer Transformer), BART.
- **Characteristics:** Uses an encoder to understand the input and a decoder to generate output.

4. **Vision Transformers (ViT)**

- **Purpose:** Image classification and vision tasks.
- **Examples:** ViT, DeiT, Swin Transformer.
- **Characteristics:** Processes image patches like sequences, applying self-attention.

5. **Multimodal Transformers**

- **Purpose:** Combine text, image, audio, or other modalities.
- **Examples:** CLIP (Contrastive Language–Image Pretraining), VisualBERT, Flamingo
- **Characteristics:** Handle multiple data types simultaneously.



▼ *26. **BERT (Encoder-Only Transformers)***

https://jalammar.github.io/illustrated-bert/

https://www.geeksforgeeks.org/explanation-of-bert-model-nlp/

https://medium.com/@shaikhrayyan123/a-comprehensive-guide-to-understanding-bert-from-beginners-to-advanced-2379699e2b51

https://www.youtube.com/playlist?list=PLeo1K3hjS3uu7CxAacxVndI4bE_o3BDtO

https://www.google.com/search?q=question+answering+system+using+bert&sca_esv=0295617911aab7ca&rlz=1C1ONGR_enIN1020IN1020&sxsrf=AHTsxN9_Ky626FmNurCGHW5twg%3A1747415523483&ei=43EnaKemHYeGnesPwrX2wQ8&oq=question+answring+sys AEBmAIBoALoAZgDAOIDBRIBMSBAkgcDMi0xoAfaBrIHAzItMbgH6AE&sclient=gws-wiz-serp

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding: https://arxiv.org/pdf/1810.04805

https://github.com/google-research/bert

BERT : Bidirectional Encoder Representations from Transformers

- BERT is basically a trained Transformer Encoder stack.

- Originating in 2018, this framework was crafted by researchers from Google AI Language.
- **Usage:** Text classification, Named Entity Recognition (NER)
- **Architecture:** Only encoder layers, designed for bidirectional context learning
- **Characteristics:** Good at understanding context within input text
- The decision to use an encoder-only architecture in BERT suggests a primary emphasis on understanding input sequences rather than generating output sequences.

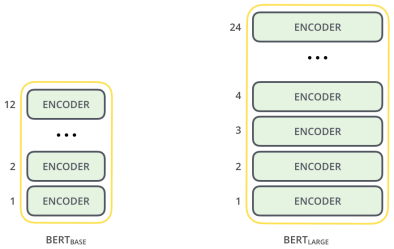
Example: "The bank is situated on the _____ of the river."

In a unidirectional model, the understanding of the blank would heavily depend on the preceding words, and the model might struggle to discern whether "bank" refers to a financial institution or the side of the river.

BERT, being bidirectional, simultaneously considers both the left ("The bank is situated on the") and right context ("of the river"), enabling a more nuanced understanding. It comprehends that the missing word is likely related to the geographical location of the bank, demonstrating the contextual richness that the bidirectional approach brings.

Model Architecture

The paper presents two model sizes for BERT:



Feature	BERT BASE	BERT LARGE
Model Size	Comparable to OpenAI Transformer	Significantly larger (“ridiculously huge”)
Layers (Transformer Blocks)/Number of encoder	12	24
Hidden Size/Embedding dimensions/d_model	768	1024
d_ff (hidden)	3072	4096
Attention Heads	12	16
Total Parameters	110 million	340 million
Performance	Baseline comparison model	Achieved state-of-the-art results in the paper
Use Case	Benchmarking against similar architectures	Maximum performance on NLP benchmarks

---

The BERT model undergoes a two-step process:

1. Pre-training on Large amounts of unlabeled text to learn contextual embeddings.
2. Fine-tuning on labeled data for specific **NLP** tasks.

Pre-Training on Large Data

- BERT is pre-trained on large amount of unlabeled text data. The model learns contextual embeddings, which are the representations of words that take into account their surrounding context in a sentence.
- BERT engages in various unsupervised pre-training tasks. For instance, it might learn to predict missing words in a sentence **(Masked Language Model or MLM task)**, understand the relationship between two sentences, or predict the next sentence in a pair.

Fine-Tuning on Labeled Data

- After the pre-training phase, the BERT model, armed with its contextual embeddings, is then fine-tuned for specific natural language processing (NLP) tasks. This step tailors the model to more targeted applications by adapting its general language understanding to the nuances of the particular task.

- BERT is fine-tuned using labeled data specific to the downstream tasks of interest. These tasks could include sentiment analysis, question-answering, **named entity recognition**, or any other NLP application. The model's parameters are adjusted to optimize its performance for the particular requirements of the task at hand.

What BERT Solves

- Traditional language models usually read text from left to right and try to predict the next word. This limits how well they understand full context.
- **BERT** reads text in **both directions** (left-to-right and right-to-left) at the same time. This helps it understand the **full meaning** of a word based on **all surrounding words**.
- It uses a neural network (Transformer encoder) to turn each word into a **smart vector** that knows the meaning of the word **in context**.
- These **contextual word representations** help BERT perform better on tasks like question answering, sentence classification, and more.

BERT solves the problem of **limited context understanding** by using **bidirectional reading**, which helps it better grasp the **true meaning of words and sentences**.

Working

BERT addresses this challenge with two innovative training strategies:

1. Masked Language Model (MLM): BERT is pretrained using a **Masked Language Modeling (MLM)** objective. This means it learns to predict missing (masked) words in a sentence by understanding the context from **both left and right sides**.
2. Next Sentence Prediction (NSP): NSP helps BERT understand **sentence relationships**, which is essential for tasks like **Question Answering** and **Natural Language Inference**.

Masked Language Model (MLM)

1. Input: "The cat sat on the mat."
2. Tokenization: ["[CLS]", "the", "cat", "sat", "on", "the", "mat", ".", "[SEP]"]
3. Masking: Randomly select some tokens (usually ~15%) to be masked.

   ["[CLS]", "the", "cat", "[MASK]", "on", "the", "mat", ".", "[SEP]"]
4. Input Embeddings

   Each token ID is converted into:

   - Token embedding
   - Segment embedding (for sentence A/B distinction)
   - Position embedding (for word order)

   These are summed and fed into BERT's encoder layers.
5. Transformer Encoding

   BERT uses **multi-head self-attention** to allow each token to gather information from all other tokens. This gives a **context-aware representation** for every token, including the `[MASK]`.
6. **Prediction**

BERT predicts the most likely word at the `[MASK]` position using the final hidden layer representation of that position.

In our example, it might predict:

"[MASK]" → "sat"

7. **Training Objective**

BERT calculates loss between the predicted word and actual word ("sat"). This helps it learn deep language understanding.

Next Sentence Prediction (NSP)

Term	Definition
Positive Pair	Sentence B is the actual next sentence that follows Sentence A in the original text.
Negative Pair	Sentence B is a **random** sentence from the corpus, not the true next sentence after Sentence A.

1. **Input Preparation**

BERT takes **two segments** as input: **Sentence A** and **Sentence B**.

Example 1 (positive pair):

- **A**: "The cat sat on the mat."
- **B**: "It looked very comfortable."

Example 2 (negative pair):

- **A**: "The cat sat on the mat."
- **B**: "I enjoy reading science fiction books."

2. **Construct Input Sequence**

BERT formats the input as:

```
[CLS] Sentence A [SEP] Sentence B [SEP]
```

So, for example 1:

> [CLS] The cat sat on the mat. [SEP] It looked very comfortable. [SEP]

3. **Segment Embeddings**

BERT uses **segment embeddings** to distinguish between Sentence A and Sentence B:

- Tokens from Sentence A get segment ID `0`
- Tokens from Sentence B get segment ID `1`

4. **Encoding**

The full input (tokens + position + segment embeddings) is fed into BERT's encoder.

5. **Prediction**

BERT uses the final hidden state of the `[CLS]` token to predict whether **Sentence B is the actual next sentence** following Sentence A.

- Output: A binary classification — **IsNext** or **NotNext**

6. **Training Objective**

During pretraining:

- 50% of the time Sentence B is the actual next sentence → **IsNext**
- 50% of the time Sentence B is a random sentence → **NotNext**

Sentence A	Sentence B	Label
"The cat sat on the mat."	"It looked very comfortable."	IsNext
"The cat sat on the mat."	"I enjoy reading science fiction."	NotNext

Why Train MLM and NSP Together?

- **MLM** teaches word-level context understanding.
- **NSP** teaches sentence-level relationship understanding.
- Together, they capture both **local (within sentence)** and **global (between sentences)** language structure.
- Improves performance on tasks like **QA, NLI, and summarization**.
- Leads to a **more robust and versatile** language model.

Applications of BERT

- **Text Classification** (e.g., sentiment analysis)
- **Named Entity Recognition (NER)**
- **Question Answering (QA)**
- **Text Summarization**
- **Next Sentence Prediction**
- **Text Similarity & Paraphrase Detection**
- **Language Translation**
- **Information Retrieval & Search Ranking**

Project: **Question Answering (QA)**

▼ *27. GPT (Decoder-Only Transformers)*

https://www.geeksforgeeks.org/introduction-to-generative-pre-trained-transformer-gpt/#architecture-of-generative-pretrained-transformer

1. **GPT-1:** *Title:* Improving Language Understanding by Generative Pre-Training

   Authors: Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever

   Year: 2018

   Summary: Introduced the idea of unsupervised pre-training of a transformer-based language model followed by supervised fine-tuning for specific tasks.

   Link

2. **GPT-2:** *Title:* Language Models are Unsupervised Multitask Learners

   Authors: Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever

   Year: 2019

   Summary: Demonstrated that large-scale transformer models trained on diverse internet text can perform many tasks without fine-tuning (zero-shot).

   Link

3. **GPT-3:** *Title:* Language Models are Few-Shot Learners

   Authors: Tom B. Brown, Benjamin Mann, Nick Ryder, et al.

   Year: 2020

   Summary: Showcased the power of extremely large language models (175B parameters) capable of few-shot, one-shot, and zero-shot learning with impressive performance.

   Link

4. **GPT-4 (Technical Report):** *Title:* GPT-4 Technical Report

   Authors: OpenAI

   Year: 2023

   Summary: Describes the architecture, capabilities, and limitations of GPT-4, highlighting improvements in reasoning, creativity, and multimodal processing.

   Link

GPT (Generative Pre-trained Transformer) – A Comprehensive Review on Enabling Technologies, Potential Applications, Emerging Challenges, and Future Directions: https://arxiv.org/pdf/2305.10435

GPT: Generative Pre-trained Transformer

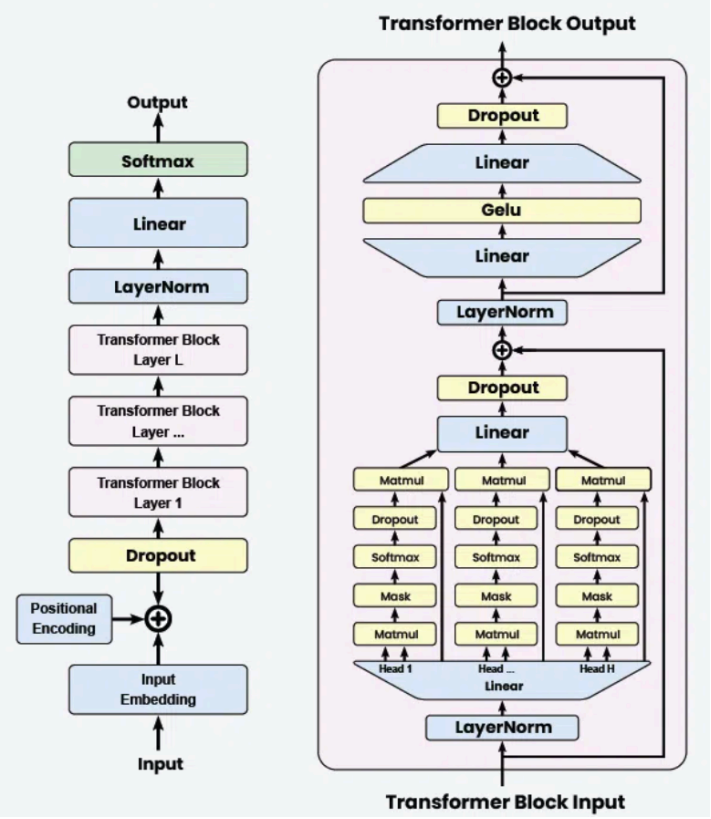
- Developed by Open AI
- To understand and generate human-like text.

- GPT can produce new content.
- Decoder-Only models

Background and Development of GPT:

- **GPT (June 2018):** First model with 12 layers, 768 hidden units, 12 heads, 117M parameters; pre-trained via unsupervised learning.
- **GPT-2 (Feb 2019):** 48 layers, up to 1.5B parameters; capable of generating coherent long-form text; release delayed due to misuse concerns.
- **GPT-3 (June 2020):** 175B parameters; major leap in zero-shot/few-shot learning; handled diverse tasks without fine-tuning.
- **GPT-4 (March 2023):** Larger model (exact size undisclosed); better reasoning, contextual understanding, and performance across domains.

Architecture of GPT



Transformer Blocks

- **LayerNorm**: Each transformer block starts with a layer normalization.
- **Multi-Head Self-Attention**: The core component, where the input passes through multiple attention heads.
- **Add & Norm**: The output of the attention mechanism is added back to the input (residual connection) and normalized again.
- **Feed-Forward Network**: A position-wise feed-forward network is applied, typically consisting of two linear transformations with a GeLU activation in between.
- **Dropout**: Dropout is applied to the feed-forward network output.

Topics

- GeLU: a activation function
- MatMul: Matrix Multiplication.

Comparison of GPT-1, GPT-2, GPT-3, and GPT-4

Feature	GPT-1	GPT-2	GPT-3	GPT-4
Release Year	2018	2019	2020	2023
Model Size	117 million parameters	1.5 billion parameters	175 billion parameters	Estimated 500+ billion parameters
Training Data	BooksCorpus (~7000 books)	40 GB of web text	570 GB from diverse sources	Larger, more diverse and curated data
Capabilities	Basic text generation	Improved fluency and coherence	Strong few-shot learning, versatile	Enhanced reasoning, creativity, and context handling
Public Access	Limited	Open-source	API-based access	API-based access via ChatGPT
Use Cases	Research	Text generation, summarization	Chatbots, coding, translation	Advanced dialogue, multimodal tasks
Limitations	Limited scale and performance	Risk of misinformation, bias	High compute cost, biases	Still lacks true understanding, expensive

Training Process of Generative Pre-trained Transformer (GPT)

- **Unsupervised Learning**: Utilizes large-scale text data corpora.
- **Pre-training Phase**:
  - Focuses on language modeling.
  - Trains the model to predict the next word in a sentence.
  - Uses diverse internet text to generalize across domains.
- **Fine-tuning Phase**:
  - Optional for specific tasks or domains.
  - Enhances performance using task-specific labeled data.
  - Supports better results for specialized applications.

Applications

- **Text Generation**: Produces human-like text for stories, articles, and dialogue.
- **Chatbots & Virtual Assistants**: Powers conversational agents and customer support.
- **Translation**: Translates text between multiple languages.
- **Summarization**: Creates concise summaries of large texts.
- **Sentiment Analysis**: Detects emotion or sentiment in written content.
- **Code Generation**: Assists in writing and debugging code.
- **Content Personalization**: Tailors recommendations and content based on user data.
- **Question Answering**: Provides accurate responses based on input queries.

Advantages:

- **Human-like Text Generation**: Produces coherent and context-aware content.
- **Versatility**: Applicable across diverse tasks (e.g., translation, summarization, coding).
- **Few-shot Learning**: Performs well with minimal task-specific data.
- **Time-saving**: Automates content creation and data processing.

Disadvantages:

- **Bias and Inaccuracy**: May generate biased or factually incorrect outputs.
- **High Resource Demand**: Requires significant computational power and memory.
- **Lack of True Understanding**: Operates statistically, not through true comprehension.
- **Data Privacy Concerns**: May inadvertently reproduce sensitive or copyrighted content.

GPT with Zero-shot, One-shot, and Few-shot Learning

GPT (Generative Pre-trained Transformer) is a powerful language model that can perform various tasks by understanding and generating human-like text. It excels at **transfer learning**, where it leverages its large-scale pretraining to handle new tasks with little to no task-specific training data. This capability is demonstrated in:

1. **Zero-shot Learning**
   - **Definition:** The model performs a task without any example or task-specific training. It relies solely on the instructions or prompts given to it.
   - **GPT Application:** When asked a question or given a prompt that describes a task, GPT can generate relevant responses based on its pre-trained knowledge, even if it has never seen that exact task before.
   - **Example:** Asking GPT to translate a sentence into another language without providing any examples.
2. **One-shot Learning**
   - **Definition:** The model learns to perform a task after seeing just one example.
   - **GPT Application:** By providing GPT with a single example input-output pair as part of the prompt, it can infer the pattern and generate the correct output for a new input.
   - **Example:** Giving GPT one example of a math problem and its solution, then asking it to solve a similar new problem.
3. **Few-shot Learning**
   - **Definition:** The model learns from a few examples (usually 2 to 10) provided in the prompt before performing the task.
   - **GPT Application:** GPT uses a handful of example pairs to better understand the task format and improve the quality and accuracy of its output.
   - **Example:** Providing multiple examples of sentiment-labeled movie reviews, then asking GPT to classify a new review's sentiment.

▼ *28. ***T5 (**Encoder-Decoder Transformers)*

▼ 29. ***Normalization

Batch Normalization is one of several **normalization techniques** used in machine learning and deep learning. Here's a structured explanation of **Batch Normalization** and other **types/techniques of normalization** used in various contexts:

✅ 1. Batch Normalization (BN)

- **Context**: Deep learning (neural networks)
- **Where**: Applied to intermediate layers (usually before or after activation functions)
- **Formula**:

  x^(k)=x(k)−μBσB2+ϵ,y(k)=γx^(k)+β\hat{x}^{(k)} = \frac{x^{(k)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \quad y^{(k)} = \gamma \hat{x}^{(k)} + \beta

  - μB,σB2\mu_B, \sigma_B^2: Mean and variance for the batch
  - γ,β\gamma, \beta: Learnable parameters for scaling and shifting
- **Purpose**:
  - Stabilizes and accelerates training
  - Reduces internal covariate shift
- **Limitation**: Depends on batch size; less effective with small batches

---

✅ 2. Layer Normalization (LN)

- **Context**: RNNs, Transformers

- **Where**: Applied across the features of a single sample (not the batch)
- **Formula**: Similar to BN, but compute mean/variance across features, not batch
- **Advantages**:
  - Works well with variable sequence lengths
  - Independent of batch size

✅ 3. Instance Normalization (IN)

- **Context**: Style Transfer, GANs
- **Where**: Applied across each individual channel of each image
- **Characteristics**:
  - Normalizes per-instance, per-channel
  - Removes instance-specific contrast information
- **Use Case**: Image style transfer

✅ 4. Group Normalization (GN)

- **Context**: Deep learning with small batch sizes
- **Where**: Channels are divided into groups; normalization is performed within each group
- **Advantage**: Independent of batch size; balances between layer and instance normalization
- **Formula**: Similar to BN, but mean/variance is computed per group

---

✅ 5. Weight Normalization

- **Context**: Deep learning optimization
- **Concept**: Reparameterizes weights as:

  $w = \frac{v}{\|v\|} \cdot g$

  - Helps with convergence by decoupling direction and magnitude of weights

---

✅ 6. Feature Normalization (Traditional ML)

Used before feeding data into algorithms:

- **Min-Max Scaling**
- **Z-score (Standard) Normalization**
- **Max Abs Scaling**
- **Robust Scaling (Median & IQR)**
- **L2 Normalization (Vector norm = 1)**

🔍 Summary Table:

Technique	Best For	Batch Dependent	Key Use Case
Batch Norm	CNNs, MLPs	✅ Yes	Faster training, stable updates
Layer Norm	RNNs, Transformers	❌ No	NLP, sequence models
Instance Norm	Style transfer, GANs	❌ No	Image style transfer
Group Norm	Small-batch CNNs	❌ No	Consistent across batch sizes
Weight Norm	Optimization	❌ No	Improved gradient flow
Feature Norms	Traditional ML	❌ No	Preprocessing input features

---

Would you like implementation examples in TensorFlow or PyTorch for any of these?

▼ **30.**
▼ **31.**
▼ **32.**
▼ **33.**

Encoder Decoder : language translation

Working with sound