UNIT-5

Sequence ~~Learning~~ Modeling:

~~Sequence Modelling~~

~~Recurrent and~~

Recurrent and Recursive Nets

Unfolding Computational Graphs, Recurrent NNs, Bidirectional RNNs, Encoder-Decoder Sequence-to-Sequence Architectures, Deep Recurrent Networks, Recursive NN, Echo state Networks, LSTM, Gated RNNs, Optimization for Long-Term Dependencies, Auto encoders, Deep Generative Models.

Recurrent Neural Networks or RNNs are a family of neural networks for processing sequential data.
CNN that is specialized for processing a grid of values such as an image.
CNNs can rapidly scale to images with large width and height, and some convolutional networks can process images of variable size, RNN can scale to much longer sequences than would be practical for networks without sequence based specialization. Most RNNs can also process sequences of variable length.

Parameter sharing makes it possible to extend and apply the model to examples of different forms and generalize across them.

Such sharing is particularly important when a specific piece of information can occur at multiple positions within the sequence.

RNNs may also be applied in two dimensions across spatial data such as images.

Sequence modeling is a crucial aspect of DL that involves understanding and predicting patterns within sequential data.

This type of data includes time series, natural language, audio signals, and more.

RNNs and Recursive NNs are types of NNs designed specifically for sequence modeling.

RNNs:

RNNs are a type of NN architecture that is well-suited for sequential data. They have loops to allow information persistence across different time steps, making them capable of capturing temporal dependencies in the data.

Architecture: The basic RNN unit has a hidden state that is updated at each time step based on the current input and the previous hidden state.

This recurrent structure enables the network to maintain memory of past inputs.

Limitation: Standard RNNs suffer from the vanishing gradient problem.

LSTM:

LSTM - Long Short Term Memory are a type of RNN

designed to overcome the vanishing gradient problem.
They include specialized memory cells and gating mechanisms to better capture long-range dependencies in the data.

Architecture: LSTMs have a more complex structure with input, forget, and output gates that control the flow of information in and out of the memory cells, allowing for better gradient flow during training.

GRU:

GRU-Gated Recurrent Unit are similar to LSTMs but have a simpler architecture with 2 gates (update and reset gates).
They are computationally more efficient than LSTMs.

ReNNs:

ReNNs are designed to operate on hierarchical or tree-structured data. Instead of processing sequences, they process recursively defined structures, such as parse trees in NLP.

In summary, both RNNs, and ReNNs are crucial in handling sequence data, but they are applied to different types of sequences.
RNNs including variants like LSTMs, and GRUs are effective for linear sequences, while ReNN are designed for hierarchical ~~ons~~ ~~tree~~ tree structured data.

Unfolding Computational Graphs :-

A computational graph is a way to formalize the structure of a set of computations, such as those involved in mapping inputs and parameters to outputs and loss.

Unfolding the graph results in the sharing of parameters across a deep ~~neural~~ network structure.

RNNs can be built in many different ways. Much as almost any function can be considered a feedforward NN, essentially any function involving recurrence can be considered a RNN.

One way to draw the RNN is with a diagram containing one node for every component that might exist in a physical implementation of the model, such as a biological NN.

The other way to draw the RNN is as an unfolded computational graph, in which each component is represented by many different variables, with one variable per time step, representing the state of the component at that ~~the~~ point in time.

We call unfolding is the operation that maps a circuit as in the left side of the figure to a computational graph with repeated pieces as in the right side.

The unfolding process introduces 2 major advantages:

→ Regardless of the sequence length, the learned model always has the same i/p size, because it is specified in terms of transition from one state to another state, rather than specified in terms of a variable length history of states.

→ It is possible to use the same transition function f with the same parameters at every time step

The unfolded graph provides an explicit description of which computations to perform.

The unfolded graph also helps to illustrate the idea of information flow forward in time (computing o/ps and losses) and backward in time (computing gradients) by explicitly showing the path along which this information flows.

Computational graphs provide a visual and mathematical way to understand how data is processed through the network over time.

Basic Computational Graph:

In NN, the computation can be represented as a computational graph, where nodes represent operations, and edges represent the flow of data b/w these operations.

Each layer in a NN corresponds to a set of operations (eg. matrix multiplications, non-linear activations).

Time Unfolding in Sequence Models:

In sequence models, like RNNs, the same set of operations is applied at each time step to process sequential data.

Instead of representing the entire sequence in a single graph, the computation is often unfolded over time, creating a series of connected graphs.

Unrolling an RNN:-
Consider a simple RNN cell with an i/p, hidden state, and output. Unfolding the computation over time involves representing the RNN cell at each time step.
The o/p at time step t becomes the input for the next time step.

Training and Backpropagation Through Time (BPTT):
During training, the unfolded computational graph is used to calculate gradients and update model parameters.
BPTT is variant of backpropagation used for training RNN.
Vanishing and Exploding Gradients:
Unfolding the computational graph over many time step can lead to challenges like vanishing (or exploding gradients, especially in traditional RNNs.
Unfolding computational graphs is a valuable visualization and conceptual tool in DL, especially for understanding the dynamics of sequence models.

Recurrent Neural Networks:-
Recurrent networks that produce an output at each time step and have recurrent connections b/w hidden units.

The RNN, when used as a Turing machine, takes a binary sequence as input and its outputs must be discretized to provide a binary output.

The back-propagation algorithm applied to the unrolled graph with O(T) cost is called BPTT.

Models that have recurrent connections from their outputs leading back into the model may be trained with teacher forcing. It is a procedure that emerges from the maximum likelihood criterion.

Teacher Forcing:

We originally motivated teacher forcing as allowing us to avoid back-propagation through time in models that lack hidden-to-hidden connections.

The disadvantage of strict forcing arises if the network is going to be later used in an open-loop mode, with the network outputs fed back as input.

One way to mitigate this problem is to train with both teacher-forced inputs and with free-running inputs.

Computing the Gradient in a RNN:

Computing the gradient through a RNN is straightforward. One simply applies the generalized back propagation algorithm to the unrolled computational graph.

The use of back-propagation on the ~~unrol~~ unrolled graph is called the BPTT. algorithm.

Gradients obtained by back-propagation may then be used with any general-purpose gradient-based techniques to train an RNN.

Once the gradients on the internal nodes of the computational graph are obtained, we can obtain the gradients on the parameter nodes.

Some common ways of providing an extra input to an RNN is

→ as an extra i/p at each time step

→ As the initial state $h^{(0)}$

Unlike traditional feed forward NN, RNNs have connections that form directed cycles, allowing them to maintain a hidden state that captures information about previous inputs.

Recurrent Structures:

The key feature of RNN is the presence of recurrent connections. These connections allow information to persist across different time steps in a sequence.

Hidden state:

This hidden state is updated at each time step and serves as a form of memory.

Applications:

→ NLP : Language modeling, machine translation, sentiment analysis.

→ Time series Analysis : Predicting stock Prices, weather forecasting..

→ Speech Recognition.

→ Video Analysis : Recognizing and generating video Captions.

Bidirectional RNNs:

~~For speech~~ In many applications we want to output a prediction of $y^{(t)}$ which may depend on the whole input sequence.

Ex: In speech recognition, the correct interpretation of the current sound as a phoneme may depend on the next few phonemes because of co-articulation and potentially may even depend on the next few words because of the linguistic dependencies b/w nearby words.

This is also true of handwriting recognition and many other sequence-to-sequence learning tasks.

Bidirectional RNN have applications ~~like~~ such as speech recognition, handwriting recognition and bio informatics.

Bidirectional RNN combine an RNN that moves forward through time beginning from the start of the sequence with another RNN that moves backward through time beginning from the end of the sequence.

This idea can be naturally extended to 2-D input, such as images, by having 4 RNNs, each one going in one of the 4-directions: up, down, left, right.

Bi-RNNs are an extension of traditional RNN that process i/p sequences in both forward and backward directions.

The key idea behind bidirectional ~~p~~processing is to capture information from the past as well as the future at each time step, which can be beneficial for tasks requiring a more comprehensive understanding of the content.

Forward and Backward Processing:

In Bi-RNN, the input sequence is processed in 2 passes: one in the forward direction, and another in the backward direction.

This is achieved by having two separate of hidden states: one for the forward pass and one for the backward pass.

Hidden State Update:

At each time step, the hidden state for the forward pass is updated on the current i/p and the previous forward hidden state.

Similarly, the hidden state for the backward pass is updated based on the current i/p and the previous backward hidden state.

Concatenation of Hidden States:

The final hidden state at each time step is obtained by concatenating the forward and backward hidden states. $h_t = [h_{forward} ; h_{backward}]$.

Application:

NLP: NER, POS tagging, and sentiment analysis.

Encoder - Decoder • Sequence-to-Sequence Architectures:-

RNN can map an input sequence to a fixed-size vector

RNN can map a fixed-size vector to a sequence.

RNN can map an input sequence to an output sequence of the same length.

This comes up in many applications, such as speech recognition, machine translation or question answering, where the i/p and o/p sequences in the training set are generally not of the same length.

The idea behind encoder-decoder sequence to sequence architectures is

→ An encoder or reader or input RNN processes the input sequence.

→ A decoder or writer or output RNN is conditioned on the fixed-length vector. to generate the o/p sequence.

In a sequence-to-sequence architecture, the 2 RNNs are trained jointly to maximize the avg of log over all the pairs of x and y sequences in the training set.

There are at least two ways for a vector-to-sequence RNN to receive input. The i/p can be provided as the initial state of the RNN, or the i/p can be connected to the hidden units at each time step. These two ways can also be combined.

There is no constraint that the encoder must have the same size of hidden layer as the decoder.

Encoder-Decoder architectures are a type of seq2seq model commonly used in NLP tasks, machine translation being a notable example.

These architectures consists of 2 main components:

→ an encoder

→ a decoder.

Encoder:

The encoder takes an i/p sequence and transforms it into a fixed-size content or hidden representation.

Processing: Each element of the i/p sequence is processed one at a time, and the hidden state of the encoder is updated at each step.

Common Architectures: The encoder is often implemented using RNN, LSTM (or) GRUs

Decoder:

The decoder takes the content vector produced by the encoder and generates an o/p sequence.

Processing: The decoder has its own internal hidden state. At each decoding step, it generates an o/p element and updates its hidden state. The hidden state is then used in the next decoding step.

Common Architectures: Like the encoder, the decoder can be implemented using RNNs, LSTMs, GRUs (or) transformer-based architectures.

Applications:

- Machine translation.
- Text summarization.
- Speech-to-text.
- Conversational Agents.

Attention Mechanism:

Attention mechanisms, particularly in the decoder, allow the model to selective focus on different parts of the i/p sequence, improving its ability to handle long range dependencies.

Deep Recurrent Networks:-

The computation in most RNNS can be decomposed into three blocks of parameters and associated transformations:

→ From the i/p to the hidden state.

→ From the previous hidden state to the next hidden state and

→ From the hidden state to the output.

We can think of the lower layers in the hierarchy as playing a role in transforming the raw i/p into a representation that is more appropriate, at the higher levels of the hidden state.

In general, it is easier to optimize shallower architectures, and adding the extra depth makes the shortest path from a variable in time step 't' to a variable in time step t+1 become longer.

Ex: An MLP with a single hidden layer is used for the state-to-state transition.

Deep Recurrent networks refer to the extension of traditional RNNs by incorporating multiple layers in the network architecture.

These deep architectures aim to capture more complex and abstract representations of sequential data by introducing depth into the recurrent layers.

Multiple Recurrent Layers:

In a deep recurrent network, multiple recurrent layers are stacked on top of each other. Each layer processes the i/p sequence sequentially, and the hidden state from one layer serves as o/p to the next layer.

Hierarchical Representation:

The layer-by-layer processing in deep recurrent networks allows the model to learn hierarchical representations of sequential data.

Lower layers capture local dependencies, while higher layers can capture more abstract and global patterns.

Applications:

- NLP: language modeling, machine translation
- Speech recognition.
- Time series prediction.

Recursive Neural Networks:-

Recursive NNs represent yet another generalization of recurrent networks, with a different kind of computational graph, which is structured as a deep tree, rather than the chain-like structure of RNNs.

Recursive networks have been successfully applied to processing data structures as i/p to neural nets in NLP as well as in computer vision.

One clear advantage of recursive nets over recurrent nets is that for a sequence of the same length T, the depth can be drastically reduced from T to O(log T), which might help deal with long-term dependencies.

When processing natural language sentences, the tree structure for the recursive network can be fixed to the structure of the parse tree of the sentence provided by a natural language parser.

The computation performed by each node does not have to be the traditional artificial neuron computation.

Recursive NNs are type of NN architecture designed to operate on recursively defined structures, such as trees or hierarchies.

Recursive structures are characterized by relationships and dependencies b/w elements that are not strictly sequential but hierarchical.

Structure and Processing:

RecNNs operate on tree-structured data, which can be syntactic parse trees in NLP, hierarchical data structures (or other recursively defined representations.

Composition Function:

At each node of the recursive structure, a composition function combines information from child nodes to produce a representation for the current node.

This function is applied recursively until a representation for the root of the tree is obtained.

Tree Traversal:

Recursive NNs traverse the tree structure in a top-down (or bottom-up manner, depending on the specific task and the nature of the data.

Top-down traversal starts from the root, and bottom-traversal starts from the leaves.

Applications:

NLP: RecNNs have been applied to syntactic parsing, sentiment analysis, and other tasks where the i/p data has a hierarchical structure.

Hierarchical Data processing: RecNN are suitable for tasks involving hierarchical data, such as processing organizational structures, family trees etc.

Recursive Neural Tensor Network (RNTN):

A specific type of RecNN is the RNTN, which uses tensor based operations to capture compositional relationships in tree-structured data. RNTN commonly used in NLP tasks.

---

whenever the model is able to represent long term dependencies, the gradient of a long term interaction has exponentially smaller magnitude than the gradient of a short term interaction.

Echo State Networks:-

Both ESN - Echo state Networks and liquid state machines are termed reservoir computing to denote the fact that the hidden units form of reservoir of temporal features which may capture different aspects of the history of inputs.

These reservoir computing recurrent networks map an arbitrary length sequence into a fixed-length vector on which a linear predictor can be applied to solve the problem of interest.

The original idea was to make the eigenvalues of the Jacobian of the state-to-state transition function be close to 1.

An important characteristic of a recurrent network is the eigenvalue spectrum of the Jacobians $J^{(t)} = \frac{\partial s^{(t)}}{\partial s^{(t-1)}}$.

The spectral radius of $J^{(t)}$ is defined to be the max of the absolute values of its eigenvalues.

An eigenvalue with magnitude greater than one corresponds to magnification or shrinking.

The strategy of echo state networks is simply to fix the weights to have some spectral radius such as 3, where information is carried forward through time but does not explode due to the stabilizing effect of saturation non-linearities like tanh.

It has been shown that the techniques used to set the weights in ESNs could be used to initialize the weights in a fully trainable recurrent network, helping to learn long-term dependencies.

Leaky Units and Multiple Time Scales:

One way to deal with long term dependencies is to design a model that operates at multiple time scales, so that some parts of the model operate at fine-grained time scales, and while other parts operate at coarse time scales and transfer information from the distant past to the present more efficiently.

Various strategies for building both fine and coarse time scales are possible.

→ Adding of skip connections through time.

→ Leaky units

→ Removal of some of the connections

ESNs were introduced to address some of the challenges associated with training traditional RNNs, such as the vanishing gradient problem and difficulty in capturing long-term dependencies.

ESNs are particularly known for their simplicity, fast training, and ability to perform well on tasks involving temporal data.

Reservoir Computing Paradigm:

ESNs belong to the family of reservoir computing ~~paradigm~~ models, where the core idea is to use a fixed and randomly initialized recurrent layer called the "reservoir" to capture temporal dependencies in the i/p data.

Reservoir:

The reservoir is an untrained recurrent layer with randomly assigned weights.

This layer acts as a dynamic system that transforms the i/p data into a high-dimensional representation, allowing the network to learn complex temporal patterns

Training Approach:

Only the o/p of the ESN is trained, while the weights within the reservoir remain fixed.

ESNs are typically trained using simple linear regression (or) ridge regression.

Applications:

→ Time-series prediction.
→ Signal processing.
→ Pattern recognition.

The echo state property allows ESNs to have memory-like capabilities, capturing information from the recent past.

Echo state networks provide a unique approach to harnessing the power of recurrent dynamics for temporal data processing while simplifying the training process.

LSTM:-

The most effective sequence models used in practical applications are called gated RNNs. These include the long-short-term memory and networks based on the gated recurrent unit.

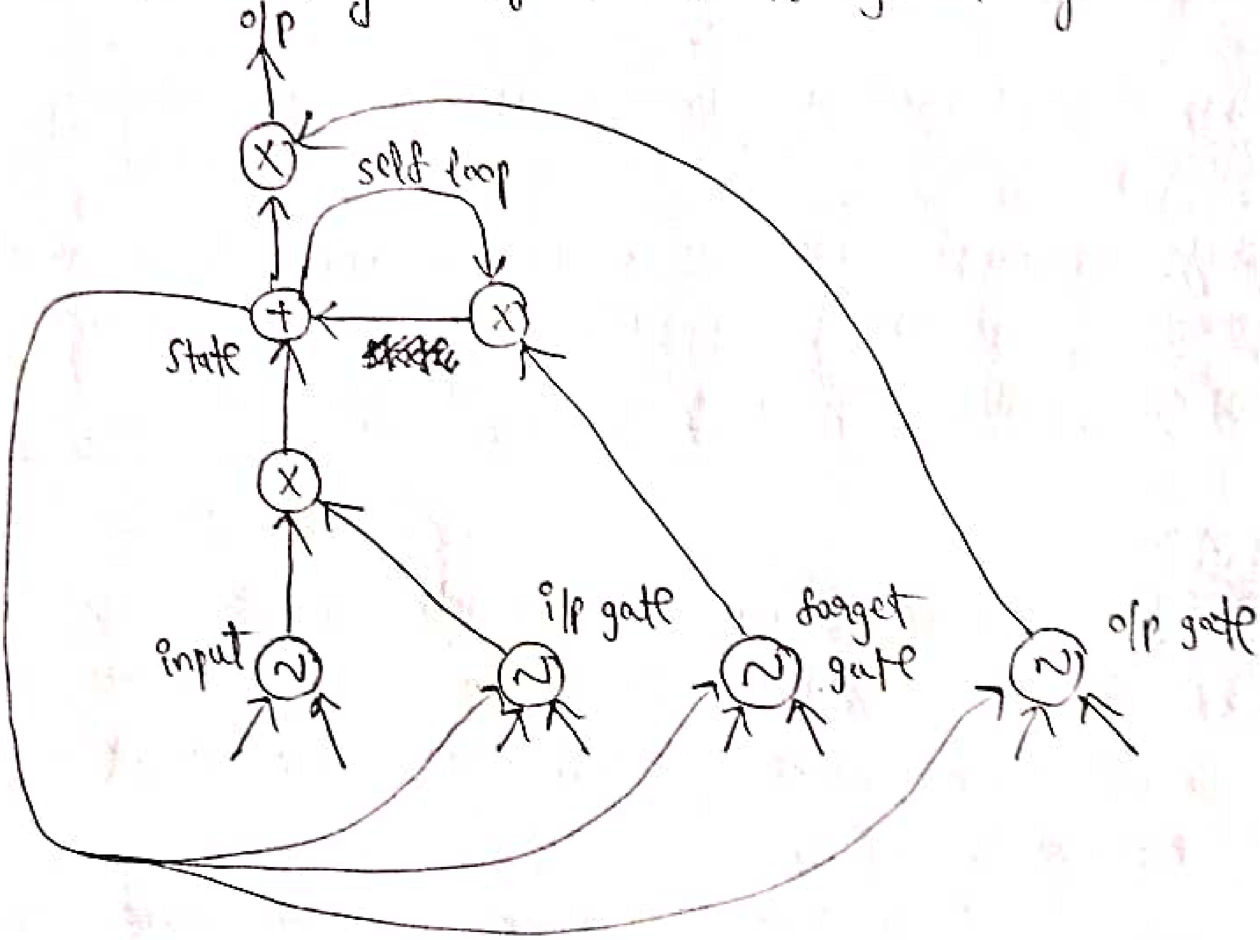
Leaky units allow the network to accumulate information over a long duration.

Once that information has been used, it might be useful for the NN to forget the old state, to forget the old state by setting it to zero. This is what gated RNNs do.

The clear idea of introducing self loops to produce paths where the gradient can flow for long durations is a core contribution of the initial LSTM model.

The LSTM has been found extremely successful in many applications such as unconstrained handwriting recognition, speech recognition, hand writing generation, machine translation, image captioning, and parsing.

The block diagram of LSTM is given by



LSTM is a type of RNN architecture designed to overcome some of the limitations of traditional RNN, such as difficulties in learning long-term dependencies.

LSTMs were introduced by Hochreiter and Schmidhuber in 1997 and have become a fundamental building block in various ~~a~~ DL applications, particularly for tasks involving sequential data.

Memory cells:

LSTM introduce a memory cell that allows the network to store and access information over long sequences.

The memory cell is equipped with mechanisms to regulate the flow of information, preventing the vanishing gradient problem.

Gates:

LSTM uses 3 gates to control the flow of information into and out of the memory cell.

Forget Gate: Decides which information from the previous state should be discarded.

Input Gate: Determines which new information should be stored in the memory cell.

Output Gate: Controls the information flow from the memory cell to the o/p.

Bidirectional LSTM:

LSTMs can be used in ~~[illegible]~~ bidirectional manner, processing sequences both forward and backward.

Bidirectional LSTM capture information from both past and future contents.

Stacked LSTM:

Multiple LSTM layers can be stacked on top of each other to create deep LSTM architectures.

Gated RNN:

The main difference with the LSTM is that ~~a~~ a single gating unit simultaneously controls the forgetting factor and the decision to update the state unit.

The reset gates control which parts of the state get used to compute the next target state, introducing an additional nonlinear effect in the relationship b/w past state and future state.

The reset gat (or forget gate o/p could be shared across multiple hidden units.

The product of a global gate and local gate could be used to combine global control and local control.

Gated Recurrent Unit (GRU) is another type of recurrent NN architecture, similar to LSTM.

Both GRU and LSTM are designed to address the vanishing gradient problem in traditional RNNs and to better capture long-term dependencies in sequential data. GRU were introduced by Cho et al in 20..

Gating Mechanism:

GRUs, like LSTM use a gating mechanism to control the flow of information within the network.

Gates:

GRUs have 2 gates:

Update Gate (z): Determines how much of the past information to carry into the future.

Reset Gate (r): Decides how much of the past information to forget.

Hidden State:

The hidden state is a combination of the current i/p and the past hidden state.

Applications:

→ NLP : Language modeling, Machine Translation.

→ Speech Recognition :

→ time series Analysis : Predicting stock prices, weather forecasting

The choice b/w GRUs and LSTMs often depends on the specific characteristics of the data and the requirements of the task.

Optimization for Long-Term Dependencies :-

second-order optimization algorithms may roughly be understood as dividing the first derivate by the second derivative.

Second-order methods have many drawbacks, including high computational cost, the need for a large minibatch and a tendency to be attracted to saddle points.

Clipping Gradients:

The objective function has a "landscape" in which one finds "cliffs": wide and rather flat regions seperated by tiny regions where the objective function changes quickly, forming a kind of cliff.

Regularizing to Encourage Information Flow:

Gradient clipping helps to deal with exploding gradients, but it does not help with vanishing gradients.

To address vanishing gradients and better capture long-term dependencies, the idea of creating paths in the computational graph of the unfolded recurrent

architectures along which is the product of gradients associated with arcs is near 1.
Another idea is to regularize or constrain the parameters so as to encourage "information flow".

Optimizing NN for long term dependencies, especially in the context of sequence models like RNNs, is crucial for capturing relationships and information over extended periods.
Long-term dependencies often lead to challenges such as vanishing or exploding gradients during training.

Gradient Clipping:
Implement gradient clipping to prevent exploding gradients during backpropagation.
This involves setting a threshold, and if the gradient surpasses that threshold, it is scaled down to a more manageable value.

Weight Initialization:
Proper weight initialization is crucial for training deep networks. Using techniques such as Xavier/Glorot initialization or He initialization can help mitigate the vanishing/ ~~gradient~~ exploding gradient problem.

Batch Normalization:
Batch normalization normalizes the inputs of a layer, which helps in reducing internal covariate shift.

Skip Connections (or) Residual Networks:

Skip connections as seen in Residual Networks, provide shortcuts for the gradient flow during training.

Gradient Descent Variants:

Adam Optimizer: Adaptive Moment Estimation (Adam) is an optimization algorithm that combines ideas from momentum and RMS Prop.

Learning Rate Scheduler: Dynamic adjustment of learning rates during training can be beneficial.

Techniques ~~like~~ such as learning rate annealing (or) cyclical learning rates can help the optimization process.

Auto Encoders:-

Auto encoders are a type of NN architecture used in USL to learn efficient representations of data, particularly in the context of dimensionality reduction.

They consist of an encoder and a decoder, and the network is trained to ~~reconstruct~~ reconstruct the i/p data from its encoded representation.

Autoencoders have various applications, including data compression, denoising, anomaly detection, and feature engineering.

Encoder:

The encoder takes the i/p data and maps it to a lower-dimensional representing, often referred to as a "code" (or) "latent space".

The goal is to capture essential features of the input data in this reduced representation.

Decoder:
The decoder takes the encoded representation and attempts to reconstruct the original i/p data.

Objective Function:
The training objective is typically based on a reconstruction loss, such as mean squared error (or binary cross-entropy.
The autoencoder aims to minimize the difference b/w the i/p data and the reconstructed o/p.

Bottleneck Layer:
The bottleneck layer represents the compressed code in the latent space.
It is the layer in the middle of the autoencoder where the i/p is transformed into a lower-dimensional representation.

Variants of Autoencoders:

Denoising Autoencoder: Trained to reconstruct clean data from noisy input, helping in learning robust representations.

Sparse Autoencoder: Introduces sparsity constraints in the encoded representation encouraging the model to use only a subset of features.

Variational Autoencoder: Combines autoencoders with probabilistic modeling, allowing for the generation of new data samples.

Applications :

→ Dimensionality Reduction.
→ Image Compression.
→ Anomaly Detection.

Training Strategies:

Autoencoders are trained using backpropagation and optimization algorithm such as SGD.

Limitations :

Autoencoders may struggle with capturing highly complex or non-linear relationships in data.

Autoencoders have proven to be versatile tools in various domains, providing a way to learn compact and meaningful representations from data.

Deep Generative Models :-

Deep generative models are a class of NN architectures designed to generate new data samples that resemble a given dataset.

These models learn to capture the underlying distribution of the data and can be used for tasks such as image generation, text generation and more.

Two prominent types of deep generative models are GANs - Generative Adversarial Networks and VAEs - Variational AutoEncoders.

GANs:

Generator and Discriminator: GANs consist of a generator and discriminator. The generator creates new samples, and the discriminator evaluates whether a given sample is real or fake.

Training objective: GANs are trained through min-max game. The generator aims to generate realistic samples to fool the discriminator, while the discriminator aims to correctly distinguish real from fake samples.

Applications: GANs have been successfully applied in image synthesis, style transfer, image-to-image translation and generating realistic images from random noise.

VAEs:

Encoder and Decoder: The encoder maps input data to probabilistic distribution in the latent space, and the decoder generates samples from this distribution.

Training objective: VAEs are trained to maximize the ELBO - Evidence Lower Bound, which encourages the learned latent space to capture meaningful and continuous representations of the input data.

Applications: VAEs are used in image generation, style transfer, data generation, and more.

Adversarial Autoencoders : AAEs

AAEs combine elements of GANs and VAEs, incorporating an adversarial loss. This helps in generating samples that are not only realistic but also diverse.

Flow based Models:

Flow based models, such as Real NVP- Non Volume Preserving and Glow, learn invertible mappings b/w data and latent spaces.

These models are used for density estimation and sample generation.

Applications:

→ Image Generation
→ Style Transfer
→ Data Augmentation.
→ Anamoly Detection.

challenges:

Training deep generative models can be challenging due to issues like mode collapse, the trade-off b/w reconstruction and regularization, and the choice of architecture and hyperparameters.

07/12/2023
[signature]