<<<<<<< Notes on Deep learning by Suraj >>>>>>>>>>>>

Deep learning is a subfield of machine learning that focuses on modeling complex patterns and representations in large amounts of data using artificial neural networks. This is called "deep" learning because it involves training a multilayer neural network (also known as a deep neural network) to learn a hierarchical representation of data.

Let's have a look at various key concepts in deep learning

00. Shallow neural networks.
01. Deep neural networks.
02. Loss functions.
03. Training models.
04. Activation function.
05. Gradient initialization.
06. Weights initialization.
07. Hyperparameter tuning.
08. Measuring performance.
09. Regularization.
10_a. CNNS
10_b. RNNS
11. LSTM and GRUS.
12. Capsule networks
13. Attention mechanisms.
14. Residual networks.
15. Transformers.
16. Graph neural networks.
17. Generative adversarial networks.
18. Normalizing flows.
19. Transfer learning.

20. Variational auto encoders.
21. Deep learning for time series.
22. Diffusion models.
23. Deep reinforcement learning.
24. Domain adaptation.
25. Federated learning.
26. Meta learning.
27. Model compression and quantization.
28. Neural architecture search.
29. One shot and few shot learning.
30. Knowledge distillation
31. Quantum machine learning.
32. Interpretability and explainability.

00. Shallow neural networks

Shallow neural networks, also known as single-layer neural networks, consist of a single neural layer connected to an input layer and an output layer. The output of the network is obtained by applying an activation function to the weighted sum of the inputs. Mathematically, the output of a shallow neural network can be expressed as:

$Z = W^*X + b$
$A = g(Z)$

where X is the input vector, W is the weight matrix, b is the bias vector, Z is the weighted sum of the inputs, g() is the activation function, and A is the output of the network.

01 Deep neural networks

A deep neural network (DNN) is a neural network with multiple hidden layers between the input and output layers. Each hidden layer contains a set of neurons that transforms the input into a higher-level representation. The output of one hidden layer serves as the input to the next hidden layer, allowing more complex representations to be learned.

Mathematically, the output of a deep neural network can be expressed as:

$Z[1] = W[1]*X + b[1]$
$A[1] = g[1](Z[1])$
$Z[2] = W[2]*A[1] + b[2]$
$A[2] = g[2](Z[2])$
$Z[3] = W[3]*A[2] + b[3]$
$A[3] = g[3](Z[3])$
... $Z[L] = W[L]*A[L-1] + b[L]$
$A[L] = g[L](Z[L])$

where X is the input vector, W is the weight matrix, b is the bias vector, Z is the weighted sum of the inputs, g() is the activation function, and L is the number of layers in the network.

The output A[L] from the last hidden layer is used as input to the output layer. The choice of the activation function g[L] depends on the task at hand. For example, if the task is binary classification, we can use sigmoid activation function, while for multi-class classification, we can use softmax activation function.

02 Loss functions

A loss function, also known as an objective function or cost function, is a mathematical function that measures the difference between the predicted output and the actual target output of a machine learning model. They play a key role in training deep neural networks because they are used to guide the optimization process by quantifying the error or loss in model predictions. The goal of training is to minimize the value of the loss function, thereby improving the performance of the model.

Different types of loss functions
are used for different types of problems, and the choice of an appropriate loss function depends on the particular problem being solved.

1. Mean Square Error (MSE):

Math formula:

MSE(y_true, y_pred) = 1/n * sum((y_true - y_pred)^2)

Explanation:

MSE calculates the mean squared difference between the predicted value (y_pred) and the true label (y_true). It is often used in regression problems that aim to predict continuous values.

Binary Cross Entropy (BCE):

Mathematical formula:

BCE(y_true, y_pred) = -[y_true * log(y_pred) + (1 - y_true) * log(1 - y_pred)]

Explanation:
BCE measures the cross-entropy between predicted probabilities (y_pred) and binary ground truth features (y_true). It is commonly used in binary classification problems that aim to predict a binary outcome (eg true/false, yes/no, etc.).

3. Categorical Cross Entropy (CCE):

Math formula:
CCE(y_true, y_pred) = -sum(y_true * log(y_pred))

Explanation:

CCE measures the mutual entropy between predicted probabilities (y_pred) and categorical ground truth labels (y_true) encoded with a single heat. It is often used in multi-class classification problems that aim to predict multiple mutually exclusive classes.

4. Hinge loss:

Math formula: hinge loss(y_true, y_pred) = max(0, 1 - y_true * y_pred)

Explanation:

The hinge loss measures the difference between the predicted value (y_pred) and the true label (y_true) in a binary classification task. It penalizes misclassified samples with margin

less than 1, encouraging the model to use larger differences between different classes.

5. Dice Loss:

Math formula:

dice loss(y_true, y_pred) = 1 - (2 * sum(y_true * y_pred) + epsilon) / (sum(y_true) + sum(y_pred) + epsilon)

Explanation:

Dice loss is often used in image segmentation problems. It measures the similarity between the predicted segmentation mask (y_pred) and the ground truth mask (y_true) using the cube coefficient formula. It penalizes the discrepancy between model predictions and the true mask with values between 0 (no overlap) and 1 (complete overlap).

6. Kullback-Leibler deviation (KL difference):

Math formula:

KL difference(y_true, y_pred) = sum(y_true * log(y_true / y_pred))

Explanation:

KL Divergence measures the deviation between the predicted

probability distribution (y_pred) and the ground truth distribution (y_true). It is often used in probabilistic models such as variational autoencoders to measure the difference between the predicted distribution and the ground truth distribution.

03 Training models

Model training in deep learning involves the process of optimizing model parameters (weights and biases) to minimize a loss or cost function. This is usually done using an optimization algorithm, such as gradient descent, which iteratively adjusts the model parameters according to the gradient of the loss function with respect to the parameters.

The general steps for training a deep learning model can be summarized as follows:

1. Data preparation: Prepare training data by performing pre-processing such as resizing images, normalizing pixel values, and splitting the data into training and validation sets.

2. Model architecture: Define the architecture of the deep learning model, including the number of layers, the types of layers (such as convolutional layers, pooling layers, fully connected layers), and their configurations (such as the number of filters, filter size, and activation). functions).

3. Initialization: Initialize the model parameters (weights and biases) with small random values or use special initialization methods (such as Xavier or He initialization) to ensure that the model starts with a good set of initial values.

5. Forward Propagation: Implements the forward propagation step, which involves passing the input data through the layers of the model to obtain the expected results. This includes applying activation functions, convolutions, pooling, and other operations based on the model architecture.

5. Compute Loss: Compute a loss or cost function that measures the difference between the predicted output and the ground truth labels. This is usually done using one of the loss functions discussed above.

6. Backpropagation: implements a backpropagation step that involves computing the gradient of the loss function with respect to the model parameters. This is done by using a computational chain rule to propagate the gradients from the output layer to the input layer.

7. Update parameters: Use an optimization algorithm such as gradient descent or its variants to update the model parameters by subtracting the product of the gradient from the learning rate. It adjusts the parameters in a direction that minimizes the loss function.

8. Iterate: Repeat forward propagation, loss calculation, backpropagation, and parameter update iteratively for multiple epochs (iterating over the entire training dataset) or until a stopping criterion is met, such as convergence of the loss function.

9. Validation: Regularly evaluate the performance of the model on the validation set to monitor its generalizability and prevent overfitting. This may include the calculation of various

performance measures such as precision, accuracy, recall and F1 scores.

10. Testing: Finally, the trained model is evaluated with a separate set of tests to obtain its performance on unseen data and evaluate its effectiveness in the real world.

04 Activation functions

Activation functions are an important part of deep neural networks because they introduce non-linearity into the model, allowing it to learn complex and non-linear data patterns. Here we discuss some of the more commonly used activation functions, their mathematical formulations, and provide code examples for each function.

1. Sigmoid Activation Function

sigmoid(x) = 1 / (1 + exp(-x))

It maps input values in the range 0 to 1, making it suitable for binary classification problems. However, it suffers from the vanishing gradient problem, where the gradient can become very small for large input values, resulting in slow convergence during training.

2. ReLU (Rectified Linear Unit) Activation Function

ReLU(x) = max(0, x)

It sets all negative input values to zero while sending positive input values unchanged. ReLU is a popular choice of activation function due to its computational efficiency and ability to alleviate the vanishing gradient problem.

3. Leaky ReLU (LReLU) activation function

The Leaky ReLU activation function is a variant of the ReLU function that addresses the "dying ReLU" problem, where some ReLU neurons may become inactive during training and never recover.

$LReLU(x) = max(\alpha * x, x)$, where α is a small positive constant (typically around 0.01)

It introduces a slight slope for negative input values, allowing some gradient flow even for negative inputs.

4. Softmax activation function

The Softmax activation function is often used in multi-class classification problems because it produces a probability distribution over multiple classes.

$softmax(x_i) = exp(x_i) / sum(exp(x_j))$, for all $i \{1, 2, ..., K\}$

where K is the number of categories and x_i is the input value of the ith category. This ensures that the output values are 1, making it suitable for multi-class classification problems.

5. Parametric ReLU (PReLU) activation function

The activation function of Parametric ReLU (PReLU) is similar to Leaky ReLU, but instead of using a fixed slope for negative inputs, it allows the slope to be learned during training. The PReLU function is defined by the following mathematical formula:

PReLU(x) = max(α * x, x), where α is a learnable parameter

This makes the PReLU function more flexible than the Leaky ReLU function because it can adjust the negative input slope according to the data.

6. GELU (Gaussian Error Linear Unit) activation function

The GELU activation function is a smooth approximation of the rectifier function designed to combine the best properties of the ReLU and sigmoid functions. The GELU function is defined by the following mathematical formula:

GELU(x) = 0.5 * x * (1 + tanh(sqrt(2 / π) * (x + 0.044715 * x^3)))

The GELU function is smooth and differentiable and does not have the gradient vanishing problem like the sigmoid function.

05 Gradient initialization

Gradient initialization is an important step in deep neural network training because it determines the initial values of model parameters, which have a significant impact on model convergence

and performance during training. Here is an explanation of some common gradient initialization methods used in deep learning:

1 Zero initialization

With zero initialization, all model parameters (weights and biases) are initialized to zeros. Mathematically, this can be expressed as:

W[l] = 0, where W[l] is the weight matrix of layer l
b[l] = 0, where b[l] is the bias vector of layer l

But initializing all parameters to zero can cause a "symmetry problem", i.e. all neurons in a single layer will have the same output and gradient and will be constantly updated with the same values during training, resulting in poor model performance. bad.

2 Sample initialization

Random initialization involves initializing the parameters with random values drawn from a given distribution. This helps eliminate symmetry issues and introduces some randomness to the pattern. Common distributions used for random initialization are the Gaussian (or normal) distribution and the Xavier/Glorot initialization.

2.1. Gaussian (normal) initialization

In Gaussian initialization, parameters are initialized with random values drawn from a Gaussian distribution with mean 0 and specified standard deviation (sigma). Mathematically, this can be expressed as:

W[l] = np.random.randn(lag_dims[l], lag_dims[l-1]) * sigma,
where W[l] is the weight matrix of layer l,
Sigma is the standard deviation,
np.random.randn generates random samples from a standard normal distribution
b[l] = np.zeros((layers_dims[l], 1))

2.2 Initialization of Xavier/Glorot

Xavier/Glorot initialization is a popular method for parameter initialization of deep neural networks. Its purpose is to make the activations and gradient differences of different layers roughly the same during training. Mathematically, Xavier's initialization can be expressed as:

W[l] = np.random.randn(layers_dims[l], layers_dims[l-1]) * np.sqrt(1 / layers_dims[l-1]),
where W[l] is the weight matrix of layer l,
np.sqrt(1 / layers_dims[l-1]) is the scaling factor

3 He initialization
Initialization is another popular parameter initialization method for deep neural networks, especially for ReLU activation functions. Its purpose is to make the differences in activations and gradients of different layers roughly equal during training. Mathematically, the initialization of He can be expressed as:

W[l] = np.random.randn(layers_dims[l], layers_dims[l-1]) *
np.sqrt(2 / layers_dims[l-1]),
where W[l] is the weight matrix of layer l,
np.sqrt(2 / layers_dims[l-1]) is the scaling factor

Gradient initialization is a critical step in training deep neural networks, and the choice of initialization method can significantly affect model convergence and performance. It is important to test different initialization methods and choose the one that works best for your particular model and task.

06 Weight Initialization

Weight initialization refers to the process of setting initial values for the neural network weights before training. The choice of initial weights can significantly affect the convergence and performance of the network during training.

On the other hand, gradient initialization is the process of initializing gradients of network parameters (including weights) to appropriate values before training begins.

This is usually done to avoid problems such as vanishing or exploding gradients that can hinder the optimization process during training.

Weight initialization and gradient initialization are related in the sense that both aim to set appropriate initial values for neural network parameters to ensure effective training.

But they differ in specific goals and techniques. Weight

initialization focuses on setting initial values of weights in a way that helps improve network convergence and performance. This can include methods such as setting the weights to zero, initializing with random values, or using specialized methods such as Xavier or He initialization that take into account the network architecture and activation functions.

Gradient initialization, on the other hand, focuses on setting the initial values of the parameters (including weights) of the gradient to appropriate values before training begins. This can include things like setting gradients to zero or initializing them with small values to prevent exploding gradients, or using techniques like Scaled Exponential Linear Unit (SELU) to stabilize gradients during training.

To summarize, Although weight initialization and gradient initialization involve setting initial values to neural network parameters, they differ in their specific goals and methods. The purpose of weight initialization is to set appropriate initial values of weights to improve the convergence and performance of the network, while gradient initialization focuses on setting appropriate initial values for the parameter gradient to ensure stable and efficient optimization during training. Both are important steps in the training process that can significantly affect the performance of a neural network.

07 Hyperparameter tuning

Hyperparameter tuning in deep learning refers to the process of selecting optimal hyperparameters for a neural network model to achieve the best performance on a given task.

Hyperparameters are variables that define the architecture, size, and training parameters of a neural network, such as the number of layers, size of each layer, learning rate, batch size, and regularization strength.

There are several approaches to tuning hyperparameters in deep learning, including grid search, random search, Keras receivers, and Bayesian optimization.

8 Measuring performance

Measuring the performance of deep learning networks involves the ability of a trained model to make accurate predictions on unseen data. This is usually done using various performance measures that quantify the ability of the model to correctly classify or predict the target variable. Types of metrics are -

1. Accuracy: Accuracy is the simplest performance measure, defined as the ratio of correctly classified samples to the total number of samples. It provides an overall measure of model performance in terms of correct predictions.

Accuracy = (number of samples correctly classified) / (total number of samples)

2. Loss: Loss is a measure of how well a model is able to reduce the difference between predicted and true values during training. It is often used as an optimization objective during model training. Lower loss values indicate better performance. Depending on the type of problem, different types of loss functions can be used, such as mean squared error (MSE) for regression

problems and cross-entropy loss for classification problems.

loss = f(y_pred, y_true)

3. Precision, recall and F1 score: Precision, recall and F1 score are common performance measures for binary or multi-class classification problems. These metrics provide insight into the trade-off between precision (the ability to correctly identify positive samples), recall (the ability to correctly identify all positive samples), and the harmonic mean of precision and recall (F1 score).

Precision = (true positives) / (true positives + false positives)

Recall = (true positives) / (true positives + false negatives)

F1 score = 2 * (precision * recall) / (precision + recall)

4. Confusion Matrix: A confusion matrix is a table often used to describe the performance of a classification model for a set of data whose true values are known. It shows the number of correctly and incorrectly predicted samples for each class. The diagonal of the confusion matrix represents true positive (TP) and true negative (TN) values, while the off-diagonal elements represent false positive (FP) and false negative (FN) values. The confusion matrix provides a detailed breakdown of the model's performance for each class, which is useful for identifying specific areas where the model may have problems.

5. Receiver operating characteristic (ROC) curve: The ROC curve is a graphical plot that shows the true positive rate (TPR) versus the false positive rate (FPR) for different classification

thresholds. It is often used in binary classification problems to evaluate a model's ability to correctly classify positive and negative samples. The area under the ROC curve (AUC) is a widely used measure of performance, and a higher AUC indicates better performance. Mathematically, the ROC curve and AUC can be calculated as follows:

$TPR = TP / (TP + FN)$
$FPR = FP / (FP + TN)$

In summary, measuring the performance of a deep learning network involves evaluating various performance metrics such as precision, loss, precision, recall, F1 score, confusion matrix, and ROC curve. These metrics provide insight into the model's ability to correctly classify or predict the target variable and help evaluate overall performance and identify any areas of the model that require improvement.

09 Regularization

Regularization is a technique used in deep training to prevent overfitting, which occurs when a model learns to perform well on training data but does not generalize very well to unseen data. Regularization methods add penalty terms to the loss function during training to limit the model weights and biases, preventing them from growing too large and reducing the model's ability to overfit. The penalty system encourages the model to learn simpler, smoother, and more general representations, which often results in better performance on unseen data.

regularization loss = loss + λ * regularization term

Where loss is the original loss function without regularization, λ is the regularization parameter that controls the strength of the regularization, and the Regularization term is the penalty term added to the loss function. Advanced education uses several types of validation techniques, including:

1. L1 regularization (Lasso regularization): L1 regularization adds a penalty term to the loss function that is proportional to the absolute value of the model weights. This forces the model to learn sparse representations by pushing some weights exactly to zero.

Regularization term = λ * ||W||1

Where W is the weight matrix of the model, ||W||1 is the L1 norm of the weight matrix (the sum of the absolute values of the weights), and λ is the regularization parameter.

2. L2 regularization (Ridge regularization): L2 regularization is to add a penalty to the loss function that is proportional to the square of the model weight. This encourages the model to learn smaller weights, preventing one weight from dominating the learning process.

Regularization term = λ * ||W||2^2

where W is the weight matrix of the model, ||W||2 is the L2 norm of the weight matrix (the square root of the sum of the squared values of the weights), and λ is the regularization parameter.

3. Elastic net regularization: Elastic net regularization is a

combination of L1 and L2 regularization. It adds a penalty term to the loss function, which is the weighted sum of the L1 and L2 regularization terms. This provides a balance between sparseness-induced L1 regularization and smoothness-induced L2 regularization.

Regularization term = $\lambda 1 * \|W\|1 + \lambda 2 * \|W\|2^2$

where $\lambda 1$ and $\lambda 2$ are the regularization parameters of L1 and L2 regularization, respectively, and W is the weight matrix of the model.

10a Convolutional neural networks

A Convolutional Neural Network (CNN) is a deep learning architecture that is highly efficient for processing visual data such as images. They aim to automatically learn the appropriate features from the input data using convolution and pooling layers, followed by fully connected layers for classification.

CNN Math:

1. Convolution operation: Convolution operation is the basic element of CNN. It involves applying convolutional filters/kernels to input data, typically images or feature maps from previous layers, to extract local patterns or features. The convolution operation is mathematically defined as follows:

Let's consider a two-dimensional input (image) represented as an H x W (height x width) matrix and a convolutional filter/kernel of size K x K, where K is usually an odd number. The convolution operation involves a dot product between the input data and a

filter/kernel at each possible location (i, j) of the input data, resulting in a feature map or output activation in dimension (H - K + 1) x Graph. (W - K + 1).

The convolution operation can be expressed mathematically as:

output[i, j] = sum(sum(input[i:i+K, j:j+K] * filter))
where Input[i:i+K, j:j+K] represents the local region of the input data to be convoluted with the filter and Filter is the convolution filter/kernel.

2. Pooling operation: The Pooling operation is used to reduce the spatial dimension of the feature map while preserving important features. The most commonly used pool operation is max pooling, which selects the maximum value from a local region of the feature map. The pooling operation is usually applied after the convolution operation, and its mathematical definition is as follows:

output[i, j] = max (input[i:i+K, j:j+K])
Among them, Input[i:i+K, j:j+K] represents the local area of the feature map, and max is the maximum pool operation.

3. Fully Connected Layers: After one or more convolution and pooling layers, the feature map is flattened into a one-dimensional vector and passed through one or more fully connected layers. These layers are similar to those in a traditional feedforward neural network, where each neuron is connected to all neurons in the previous and subsequent layers. The output of the last

fully connected layer is used for classification.

10b Recurrent Neural Networks

A Recurrent Neural Network (RNN) is a deep learning architecture designed to model sequential data where the order of the data points is important. RNNs are able to capture temporal dependencies and learn patterns in data sequences. They are commonly used in tasks such as natural language processing, speech recognition, time series analysis, etc.

At each time step of an RNN, the input data for that time step is processed by an RNN unit that has a hidden state that captures information from previous time steps. The hidden state at time step t, denoted by h(t), is computed using the input data at time step t, denoted by x(t), and the hidden state at time step t-1, denoted by h(t-1), then the formula :

$$h(t) = \text{activation\_function}(W_{hh} * h(t-1) + W_{xh} * x(t) + b_h)$$

$$y(t) = \text{activation\_function}(W_{hy} * h(t) + b_y)$$

where:

h(t) is the hidden state at time step t
x(t) is the input data at time step t
W_hh is the weight matrix for the hidden state
W_xh is the weight matrix for the input data
b_h is the bias term for the hidden state
y(t) is the output at time step t
W_hy is the weight matrix for the output
b_y is the bias term for the output

activation_function is an activation function applied element-wise

An RNN cell updates its hidden state at each time step, incorporates information from the current input data and previous hidden states, and produces an output at each time step. The hidden state acts as a memory capture context from previous time steps, allowing the RNN to capture sequential data patterns and dependencies.

LSTM and GRU

Long-Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are two recurrent neural network (RNN) architectures commonly used for temporal computing tasks such as weather forecasting, natural language processing, and speech recognition.

They are designed to overcome the limitations of traditional RNNs, such as the vanishing gradient problem that can occur when training deep neural networks.

The LSTM and GRU architecture implement a gating mechanism that allows the network to selectively update and store long-sequence information, enabling better learning and modeling of long-term dependencies in sequence data.

1. LSTM (Long Short Term Memory):

An LSTM cell has three gates: an input gate, a forget gate, and an output gate. The input gate controls how much new information is added to the cell state, the forget gate controls how much information is forgotten from the cell state, and the output gate controls how much information is sent to the

output.

f_t = sigmoid(W_f . x_t + U_f . h_t-1 + b_f) # forget gate
i_t = sigmoid(W_i . x_t + U_i . h_t-1 + b_i) # input gate
o_t = sigmoid(W_o . x_t + U_o . h_t-1 + b_o) # output gate
c_t = f_t .* c_t-1 + i_t .* tanh(W_c . x_t + U_c . h_t-1 + b_c) # cell state
h_t = o_t .* tanh(c_t) # hidden state/output

Where:

x_t is the input at time step t
h_t is the hidden state/output at time step t
c_t is the cell state at time step t
W_f, W_i, W_o, W_c are the weight matrices for the input x_t
U_f, U_i, U_o, U_c are the weight matrices for the previous hidden state h_t-1
b_f, b_i, b_o, b_c are the bias vectors
sigmoid is the sigmoid activation function
tanh is the hyperbolic tangent activation function
- .* represents element-wise multiplication

2. GRU (Restricted Periodic Unit):
The GRU device has two gates: a update gate and a reset gate. The update gate determines how much of the previously hidden state is mixed with the new candidate hidden state, and the reset gate determines how much of the previously hidden state is forgotten.

The key difference between GRU and LSTM is that GRU has fewer parameters than LSTM and is therefore faster to train. GRU also combines the forget and input gates of LSTM into a single update gate and merges the cell state and hidden state into a single hidden state

Given an input sequence of length $T = (x_1, x_2, ..., x_T)$, and hidden state h_t at time step t, GRU computes the update gate z_t, reset gate r_t, and new candidate hidden state $\tilde{h}_t$ as follows:

$$z_t = \text{sigmoid}(W_z x_t + U_z h_{t-1} + b_z)$$
$$r_t = \text{sigmoid}(W_r x_t + U_r h_{t-1} + b_r)$$
$$\tilde{h}_t = \tanh(W x_t + r_t (U * h_{t-1}) + b)$$

where W, U, and b are the weight matrix, hidden state matrix, and bias vector, respectively, for the corresponding gate or hidden state. sigmoid is the sigmoid activation function, and tanh is the hyperbolic tangent activation function.

Next, the current hidden state h_t is computed as a weighted sum of the previous hidden state h_{t-1} and the candidate hidden state $\tilde{h}_t$, with the weights determined by the update gate z_t:

$$h_t = (1 - z_t) h_{t-1} + z_t \tilde{h}_t$$

12 Capsule networks

Capsule networks, also known as CapsNets, are a neural network architecture introduced by Jeffrey Hinton in 2017. CapsNets are designed to model hierarchical relationships

between visual elements in images or other data types, and are particularly effective for tasks that require understanding spatial relationships and estimating object position.

The main idea behind CapsNets is to represent entities called capsules as vectors instead of the scalar values used in traditional neural networks. Each capsule represents a specific visual object or part of an object and encodes information such as its presence, pose, and instantiation parameters. These capsules are organized in a hierarchy, where higher-level capsules represent more complex functions that build on lower-level capsules.

Blocks of capsnet are

1. Primary capsules: The input data is first processed by a set of convolutional layers to extract local features. These local features are then grouped into capsules, each representing a specific visual feature. The result of this operation is a set of primary capsules denoted as u_{ij}, where i represents the index of the primary capsule and j represents the index of the local element.

2. Capsule activation: The Primary capsule is then activated using a non-linear activation function, typically a squeeze function, that maps the length of the capsule vector to a value between 0 and 1 that represents the probability that a particular visual feature is present. The Flatten function is defined as:

$$v_{ij} = \text{squash}(u_{ij}) = (\|u_{ij}\|^2 / (1 + \|u_{ij}\|^2)) * (u_{ij} / \|u_{ij}\|)$$

where v_{ij} is the output of the compression function and $\|u_{ij}\|$

is the Euclidean norm of the primary capsule vector.

3. Dynamic Routing: The activated master capsules are then used to compute the output of higher-level capsules using a process called dynamic routing. Dynamic routing involves iteratively updating the connection coefficients of the primary and top-level capsules based on the correspondence between their predicted and actual results. The link coefficients are updated using a protocol routing algorithm that aims to route more active capsules to higher protocol capsules and vice versa. This allows CapsNets to model spatial relationships and estimate object poses in a hierarchical manner.

4. Final prediction: The result of the dynamic routing process is the CapsNet final prediction, which can be used for various tasks such as classification, object detection, or pose estimation.

13 Attention mechanism

Attention is a mechanism used in neural networks to selectively focus on different parts of the input data, allowing the model to learn to weigh the importance of different elements of the input when making predictions or generating output. Attention mechanisms have been widely used in natural language processing (NLP) tasks such as machine translation, sentiment analysis, and text summarization. Mathematically, the mechanism of attention can be described as follows:

1. Input Embedding: The input data (such as a sequence of words in NLP) is first embedded into a continuous vector representation using an embedding layer. It converts discrete input data into a continuous representation that can be processed by a neural

network.

2. Query, key, and value: The embedded input data is then used to generate three sets of vectors-query, key, and value. These vectors are linear transformations of the input embeddings, which are typically computed using a learnable weight matrix. The query vector represents the information the model wants to retrieve or focus on, the key vector represents the information in the input data, and the value vector represents the actual value or representation of the input data.

3. Attention Weights: The next step is to calculate attention weights, which determine how much attention or importance each input data element should have. Note that the weights are calculated by measuring the similarity between the query vectors and the key vectors. Common methods of measuring similarity include dot product, cosine similarity, or scaled dot product.

4. Softmax activation: The attention weights are then passed through a softmax activation function, which normalizes the weights to be between 0 and 1 and ensures that the weights sum to 1. This softmax activation allows the model to allocate attention among the input elements in a probabilistic manner. , giving more weight to more relevant elements and less weight to less relevant elements.

5. Weighted Sum: Attention weights are used to calculate a weighted sum of value vectors, where values with higher attention weights contribute more to the final representation. This weighted sum represents the context or participation representation of the input data, which is a weighted combination

of input values based on their importance.

6. Output: The participant representations or context vectors are then used as input to subsequent neural network layers for further processing or prediction. The attention mechanism allows the model to selectively focus on different parts of the input data according to the relevance of the input data, which improves the model's ability to capture important information and make accurate predictions.

Attention mechanisms can be implemented in different ways, such as self-attention or scaled dot-product attention, often used in models based on transformers, or additive attention or multiplicative attention, which are early versions of attention mechanisms.

14 Residual networks

Residual networks, also known as ResNets, are a deep neural network architecture that solves the vanishing gradient problem that can occur when training deep networks. ResNets implements "skip connections" which allows the network to skip certain layers, allowing the gradients to flow and easing the vanishing gradient problem. Mathematically, let's consider a deep neural network with L layers, denoted by H_l, where l = 1, 2, ..., L denote the layer indices. Each layer takes an input x_l and produces an output y_l using a set of learned parameters W_l. In a traditional neural network, the output of layer l is calculated as follows:

$$y_l = f(W_l * x_l),$$

where f() is the activation function and * represents the matrix multiplication operation.

ResNet introduced skip connections to allow the network to bypass certain layers. The remaining block is the basic element of ResNet, defined as:

$Y_l = f(W_l * x_l + x_l)$,

Where x_l is the input of layer l and Y_l is the output of layer l. A skip connection allows the input to flow directly to the output and then adds it to the output of a traditional neural network layer ($W_l * x_l$) to produce Y_l. . ResNets also call this "identity mapping" or "shortcut connection".

ResNets are based on the intuition that adding skip connections allows the network to learn the residual map, the difference between the input and output of a layer. This makes it easier for the network to learn the approximate identity mapping, which is often easier than learning the entire mapping from scratch. In addition, hop connections allow gradients to more easily propagate through the network, easing the vanishing gradient problem and allowing deeper networks to be trained.

15 Transformers

Transformer is a neural network architecture that has been widely used in various natural language processing (NLP) tasks such as machine translation, text classification, and language generation. They are based on a self-awareness mechanism that allows models to consider different parts of the input sequence with different degrees of attention, making them very efficient at capturing long-range dependencies in sequence data. The main idea behind Transformers is a self-awareness mechanism that allows the model to weigh the meaning of different words in the input sequence when making predictions. The self-attention mechanism calculates an attention score for each word in the input sequence based on the ratio of all pairs of words in the sequence. These attention scores are then used to compute weighted representations of the input sequences, which are used as input to subsequent layers of the network.

Let's take the example of the string "I love this movie, it's great!" and use this example to understand the flow of the Transformer model for text classification.

1. Input Embeds: The input text sequence "I love this movie, it's great!" is first transformed into a sequence of word embeddings. Assume that the word embedding for each word in the sequence is $[x_1, x_2, ..., x_n]$, where x_i is the embedding vector of the ith word.

2. Vectors of queries, keys and values: The transformer model uses three sets of learnable weight matrices to compute the query, key, and value vectors for each word in the input sequence. These vectors are represented as $Q = [q_1, q_2, ...,$

q_n], K = [k_1, k_2, ..., k_n] and V = [v_1, v_2, ..., v_n]. Suppose these vectors are for the input sequence "I love this movie, it's great!".

3. Attention Score: The attention score for each word in the input sequence is calculated as the score product between the query and key vectors. The dot product between the i-th query vector (q_i) and the j-th key vector (k_j) is expressed as $a_{ij} = q_i * k_j$. These attention scores reflect the similarity or relatedness of different words in the input sequence.

4. Softmax and Weighted Representation: Attention scores are normalized using a softmax function that transforms them into a probability distribution over all words in the input sequence. Apply the softmax function to the rows of the matrix of attention indicators, obtaining a new matrix of the same size. This matrix is then used to compute a weighted representation of the input sequence, where each word is weighted by an attention score. Let's assume a weighted representation of the input sequence "I loved this movie, it was great!". Expressed as Y = [y_1, y_2, ..., y_n].

5. Multi-Head Attention: To capture different input sequence patterns, the self-attn mechanism is typically invoked multiple times in parallel using different sets of query, key, and value vectors. These parallel mechanisms of self-attention are called "heads". The output of each head is concatenated along the last dimension to obtain a final weighted representation of the input sequence.

6. Position Transfer Networks: After obtaining the weighted representations of the input sequences, they are passed

through position transfer networks, which are fully connected layers that are applied independently to each position of the sequence. These feedforward networks help the model capture local dependencies in the input sequence.

7. Layer Normalization and Residual Connections: Layer normalization is applied after each self-mechanism and positional feed network to improve model stability and convergence. Additionally, residual connections are used to bypass each self-awareness mechanism and position the feed-forward networks, allowing the model to capture both local and global dependencies in the input sequence.

8. Output: The final output of the transformer model is usually obtained by subjecting the weighted representation of the input sequence to a linear projection followed by a softmax activation function that provides a probability distribution over different mood classes (positive or negative). The category with the highest probability is considered as the expected sentiment feature for the input text.

16 GNNs

Graph Neural Networks (GNNs) are a type of neural network that can process data represented as graphs, which are a collection of nodes (also called vertices) connected by edges (also called edges or links). GNNs have become popular in various domains, such as social networks, bioinformatics, recommendation systems, and knowledge graphs.

17 Generative adversarial networks

A generative adversarial network (GAN) is a deep learning model that consists of two neural networks, a generator and a discriminator, that are trained together in competition. GANS are used to generate realistic and high-quality data such as images, music, and text.

The generator network takes as input a random noise vector, usually drawn from a normal distribution, and produces synthetic data samples. A discriminator network takes as input samples of real data (such as real images) from a target distribution and samples of synthetic data generated by a generator, and aims to distinguish between the two. The goal of the generator is to generate synthetic samples that are indistinguishable from real samples, while the goal of the discriminator is to correctly classify whether the samples are real or fake.

GAN training can be summarized in the following steps:

1. Generator network:
- Input: random noise vector of shape z (batch_size, latent_dim)
- output: synthetic samples of the form x' (batch_size, data_dim)

2. Network of discrimination:
- Input: real samples of shape x(batch_size, data_dim) and synthetic samples of shape x'(batch_size, data_dim)
- Output: binary classification scores in the form d and d' (batch size, 1), where d is the discriminator prediction for the real sample x and d' is the discriminator prediction for the synthetic sample x'.

3. Loss function:

- Generator loss: The generator is designed to generate synthetic samples that can "fool" the discriminator so that its loss is based on the discriminator's predictions for the synthetic samples.
- Discriminator Loss: The goal of a discriminator is to correctly classify true and false samples, so the loss is based on the difference between its predictions for true and false samples.

4. Training process:
- The generator and discriminator are trained together in a adversarial manner using variable optimization. During each iteration, the generator is trained to minimize the loss by updating the weights, while the discriminator is trained to minimize the loss by updating the weights.
- The training process will continue for a specified number of iterations or until convergence is reached. The mathematical equations for the generator and discriminator losses can be defined as:

Generator losses:

$$L\_G = -1 * \text{mean}(\log(d'))$$

where d' is the discriminator prediction for the synthetic sample x' produced by the generator.

Loss of discriminator:

$$L\_D = -1 * (\text{mean}(\log(d)) + \text{mean}(\log(1 - d')))$$

where d is the discriminator prediction for a real sample x, and d' is the discriminator prediction for the synthetic sample x'

generated by the generator.

During training, the generator and discriminator update their weights using gradient descent or other optimization algorithms to minimize the respective losses. The generator and discriminator are usually trained at different learning rates and updated alternately to achieve a balance between the two networks.

This process repeats until the generator produces real samples that can "fool" the discriminator, and the discriminator is no longer able to distinguish the real samples from the fake ones. At this time hopefully, the GAN is converged and the generator can be used to generate synthetic data samples from the target distribution that resemble real data samples.

18 Normalizing flows

Normalized flow is a class of generative models in deep learning that aims to learn bijective mappings between simple distributions (such as Gaussian distributions) and complex distributions (such as high-dimensional datasets). The main idea is to use a series of reversible transformations or "flows" to transform samples from simple to complex distributions.

Mathematically, the normalization flow consists of a sequence of reversible transformations denoted T1, T2, ..., Tn, where each transformation maps a data point x in the input space to a new data point y in the transformed space. The forward transformation is expressed as $y = T(x)$, and the inverse transformation is expressed as $x = T^{(-1)}(y)$, where $T^{(-1)}$ is the inverse of T.

Each transformation T is usually designed to be computationally efficient and must have a tractable Jacobian that defines the locally linearized Jacobian describing the transformation. This is important because it efficiently computes the probability density function (PDF) of the transformed data points, which is required for generative modeling.

During training, a normalized flow is trained to minimize the difference between the data distribution and the model distribution. This is usually done using maximum likelihood estimation (MLE) or variational methods, where the model is optimized to maximize the likelihood of the training data or minimize the difference between the model distribution and the data distribution.

Normalized flows have several advantages, including the ability to generate high-quality samples from complex distributions, the ability to perform accurate probability calculations, and the ability to make inferential inferences (such as computing posterior probabilities). However, they also face some challenges, such as the need for reversible and computationally efficient transformations and the possibility of overfitting due to the large number of parameters of the depth flow.

19. Transfer learning

Transfer learning is a technique within deep learning in which a pre-trained neural network model is used as a starting point for solving different but related problems. Instead of training a new model from scratch, transfer learning allows us to use knowledge gained from a large dataset for different tasks to

improve the performance of smaller datasets on a target task.

The main idea behind transfer learning is that neural networks learn general features from large data sets that can be used for other tasks. By using pre-trained models, we can exploit these learned features, reducing the amount of data and time required for training, while potentially improving performance on the target task. The learning transfer process usually includes the following steps:

1. Pre-training model selection: Choose a pre-training model that is trained on a large dataset and is appropriate for the target task. Models such as VGG, ResNet, Inception, and MobileNet that are pre-trained on large image datasets are commonly offered.

2. Adapting the model: Remove the original output layers from the pre-built model and replace them with new output layers specific to the target task. These new output layers are initialized randomly and will be trained from scratch during refinement.

3. Refinement: train the modified pre-trained model on the target data set, usually using a lower learning rate to avoid overfitting and adapt the model to the specific characteristics of the target data set. Pre-trained layer weights can be frozen to prevent further updates or to allow refinement depending on the size of the target dataset and the amount of data available.

Transfer learning can have a number of benefits, including:

Performance improvement, Faster training and Reduced data requirements

20. Variational Auto encoders

A variational autoencoder (VAE) is a generative model that learns to generate new data points by encoding and decoding data points in a low-dimensional latent space. They are commonly used in tasks such as image generation, text generation and data generation in various fields.

The main idea behind VAE is to introduce a probabilistic framework into traditional autocoding architectures. In a standard autoencoder, the encoder maps the input data into a fixed low-dimensional latent representation, and the decoder maps the latent representation back to the input data. In contrast, the VAE encoder maps the input data to a probability distribution in latent space, while the decoder maps samples back to the input data in latent space.

This probabilistic encoding and decoding process allows the VAE to generate different samples from the latent space, resulting in more realistic and diverse generated data. The mathematical formulation of VAE includes the following components:

1. Encoder ($Q(z|x)$): The encoder maps the input data (x) to a probability distribution in the latent space (z). It is usually expressed as a Gaussian distribution with mean (μ) and standard deviation (σ) parameters. The encoder is trained to learn the parameters of this Gaussian distribution given the input data.

2. Latent space (z): The latent space is a low-dimensional representation that encodes the input data. It is often assumed to follow a Gaussian distribution (ie, $N(0, I)$) so that the latent space is uniform and regular.

3. Reparameterization trick: During training, to enable backpropagation and gradient-based optimization, VAE uses a reparameterization trick to sample the learned Gaussian distribution in latent space. This trick involves generating random samples (ε) from a standard Gaussian distribution and then transforming them using the mean (μ) and standard deviation (σ) parameters learned in the encoder to obtain samples from a Gaussian distribution in latent space. It separates the randomness in latent space from the deterministic computation of the encoder and provides a smooth gradient-based optimization.

4. Decoder ($p(x|z)$): Decoder maps samples from latent space (z) back to input data (x). It generates reconstructed data (x') from latent representations using a neural network trained to minimize the reconstruction error between the reconstructed data and the original input data.

5. Loss function: VAEs are trained using a combination of two loss concepts: the reconstruction loss, which measures the difference between the original input data and the reconstructed data, and the regularization loss, which forces the latent space to follow a Gaussian distribution. . Reconstruction loss is usually measured using an appropriate distance metric such as mean squared error (MSE) or binary cross entropy (BCE) loss.

During training, VAE aims to update encoder and decoder parameters by minimizing the combined loss (reconstruction loss + regularization loss) using gradient-based optimization methods such as stochastic gradient descent (SGD) or Adam. Once the decoder is trained, it can be used to generate new data samples

by sampling from the latent space, thus producing different data points with different characteristics.

21 Deep Learning for mathematics

Deep learning for time series data is a machine learning technique that uses artificial neural networks with multiple hidden layers to model and predict time series data. Time series data is data collected over time where the order of the data points is important, such as stock prices, weather data, or sensor readings. Mathematically, deep learning of time series involves the use of neural networks to learn the underlying patterns and structures of time series data.

A typical deep learning architecture for time series data consists of an input layer, one or more hidden layers, and an output layer. Each layer consists of interconnected nodes (also known as neurons) that process input data and pass it through an activation function to generate output values. During training, the neural weights and biases are learned to optimize the model and make accurate predictions.

A common type of neural network for time series data is the recurrent neural network (RNN), which is designed to capture the temporal dependencies of the data. RNNs have recurrent connections that allow information to persist over time, making them suitable for sequential data. Another popular RNN variant is the Long Short-Term Memory (LSTM) network, which is capable of capturing long-term dependencies and is particularly efficient at processing long sequences of data.

In addition to RNNs, other types of deep learning models such as Convolutional Neural Networks (CNNs) and transforms can be applied to time series data, depending on the type of data and the specific problem to be solved. The process of training a deep learning model on time series data involves minimizing a loss function that measures the difference between the predicted output and the actual output. The optimizer is used to update the neural weights and biases to minimize losses during training. Once the model is trained, it can be used to predict new, unseen time series data.

Deep learning of time series data has shown promise in various applications such as stock market forecasting, weather forecasting, anomaly detection, and speech recognition. However, it also presents challenges, such as the need for appropriately labeled data, careful handling of overfitting, and proper model selection and hyperparameter tuning.

2.2 Diffusion models

Diffusion models, also known as diffusion processes or diffusion equations, are mathematical models that describe the diffusion of a quantity or diffusion in a medium over time. They are widely used in various fields, including physics, chemistry, biology, finance, and image processing. The basic idea behind diffusion models is that the quantity being modeled (such as heat, concentration, or information) tends to diffuse from areas of high concentration to areas of low concentration. The rate of diffusion depends on the concentration gradient, the difference in concentration between adjacent areas. The higher the concentration gradient, the faster the diffusion.

A commonly used diffusion model is the diffusion equation, which is a partial differential equation (PDE) that describes the time evolution of the diffusive quantity. Diffusion equations are usually expressed in the following form:

$\partial u/\partial t = D\nabla^2 u$

where u is the diffusive quantity, t is time, D is the diffusion coefficient, and $\nabla^2 u$ is the Laplace operator applied to u representing the spatial gradient of u. The diffusion equation describes how the concentration "u" changes with time due to diffusion and the diffusion coefficient "D" determines the rate of diffusion. Diffusion models can be used to model and analyze a variety of diffusion phenomena, such as heat conduction, chemical diffusion, and the spread of information in social networks.

They can also be used in image processing tasks such as image segmentation and image smoothing, where a diffusion process is used to spread information between adjacent pixels to improve image quality.

23 Deep reinforcement learning

Deep reinforcement learning (DRL) is a type of machine learning that combines reinforcement learning (RL) with deep neural networks (DNN). It involves training the agent to make decisions in the environment by acting on observed states with the goal of maximizing the cumulative reward signal. Mathematically, DRL can be described using the following components:

1. Markov Decision Process (MDP): MDP is a mathematical model

that defines the interaction between an agent and its environment. It is defined by the string (S, A, P, R), where:

- S is the state space representing all the possible states the environment can be in.
- A is the action space representing all the possible actions an agent can take. - "P" is a transition probability function that determines the probability of going from one state to another when an action is performed.
- R is the reward function that defines the immediate reward that the agent receives after performing a certain action in a certain state.

2. Policy: A policy is a strategy used by an agent to decide what action to take in a given state. It can be deterministic or stochastic and is usually expressed as a function that maps states to actions.

3. Value Function: A value function estimates the expected cumulative reward that an agent can obtain from a given state or state-action pair under a given policy. It is used to guide the agent's decision-making process by evaluating the long-term desirability of various states or actions.

4. Q-value function: The Q-value function, also known as the action-value function, is similar to the value function, but it considers the specific action performed in addition to the state. It calculates the expected cumulative reward that an agent can obtain from a given state-action pair under a given policy.

5. Bellman Equation: The Bellman equation is a key equation in reinforcement learning that expresses the relationship between

the value of a state or state-action pair and the value of an adjacent state or state-action pair. It is used to update the value and Q value function during the learning process.

6. Deep Neural Network (DNN): DNN is used to approximate a policy, value function or Q-value function in DRL. These are typically multilayer neural networks with multiple hidden layers that can learn complex representations from raw state or state action inputs.

7. Replay Buffer: The replay buffer is a DRL data structure used to store and sample the agent's previous experience. This helps to break the temporal relationship between successive samples and improve the stability of the learning process.

8. Exploration vs. Exploitation: Exploration refers to trying out different behaviors to discover their impact on the environment, while exploitation refers to current policies. Achieving a balance between exploration and exploitation is a key challenge in DRL to ensure that the agent does enough exploration to learn an optimal policy without getting stuck in a suboptimal one.

9. Learning Algorithms: DRL algorithms typically use a combination of RL techniques (such as Q-learning, SARSA, or Actor-Critic) and DNNs to learn a policy, value function, or Q-value function. These algorithms update model parameters based on observed experience and the Bellman equation to optimize strategies or value estimates.

24 Domain Adaptation

Domain matching is a subfield of machine learning and domain

transfer that aims to train a model on a source domain and use it to make accurate predictions about the target domain, even if the source and target domains have different distributions. In other words, domain adaptation aims to overcome the challenge of domain transfer, when a model trained in one domain (e.g. source domain) does not perform well in another domain (e.g. target domain) due to differences in data distribution. going on.

Domain matching is particularly useful in situations where it is expensive, time-consuming, or impractical to collect tagged data from the target domain. It is commonly used for tasks such as image recognition, natural language processing, speech recognition and many other practical applications.

Mathematically, domain matching can be formulated as follows: given a labeled source domain dataset Ds = $\{(x_i, y_i)\}$ from the source domain Ds and an unlabeled target domain dataset Dt = $\{x_i\}$ from the target domain Dt, the goal is to use the knowledge obtained from the source domain data to learn a model f(x) that can accurately predict the target domain label y_t for a sample x_t in the target domain.

Domain matching techniques typically involve using labeled source domain data to learn a model that can accommodate differences in data distributions between source and target domains. Some common techniques used in domain matching include domain matching techniques such as domain matching neural networks (DANNs), conflicting domain matching, and domain matching with domain-specific features or representations.

25. Federated learning

Federated learning is a machine learning approach that allows multiple distributed entities (such as devices or servers) to collaborate to train a shared machine learning model without sharing their raw data with a central server. Instead, each device trains a model locally based on its own data and only shares model updates or gradients with a central server that aggregates these updates to update the global model. It allows you to train machine learning models on multiple devices while keeping your data decentralized and secure.

The main idea behind federated learning is to use available data from different entities (often called customers or employees) to train a global model that uses the collective knowledge of all entities without transmitting or revealing the raw data. Blended learning is particularly useful when data is distributed across multiple devices (e.g. edge devices (e.g. smartphones, IoT devices)) or between different organizations (e.g. hospitals, banks) that may have privacy concerns or legal restrictions on data sharing, which is important.

The blended learning process usually includes the following activities:

1. Initialization: The global machine learning model is initialized on a central server.
2. Client Update: Each client device locally trains the global model using its own data and updates the model by computing gradients.
3. Model Aggregation: A central server aggregates client gradients to update the global model.
4. Model Deployment: The updated global model is then distributed to client entities that use it for local inference or

further training.
5. Iteration: 2-4. the operation is repeated iteratively for several rounds until the global model converges to an acceptable level of performance.

Federated training offers several benefits, including data protection because raw data is never left on the local device, reducing the risk of data breaches or privacy violations. It also enables efficient and scalable training of machine learning models on distributed devices, enabling faster model updates and reducing communication overhead. But federated learning also presents challenges such as data heterogeneity between clients, communication and synchronization overhead, and potential bias due to non-IID. (independent and identically distributed) data. Successful implementation of federated learning in real-world scenarios requires careful consideration of these issues, as well as appropriate model development and optimization techniques.

26. Meta learning

Meta-training, also known as "learning to learn", is a machine learning approach that aims to enable a model to learn to adapt and generalize to different tasks or domains for which data from the target task or domain is limited or unavailable. Field. In other words, meta-training focuses on training a model so that it learns to learn efficiently by improving its performance in new, unseen tasks or domains by using previous experience with related tasks or domains.

Metalearning is often used in scenarios where data availability is limited and it may not be possible to collect large amounts of

labeled data for each target task or domain. Meta-learning algorithms typically consist of two levels of learning: the meta level and the task level.

1. Meta-level learning: In the meta-level, the model learns to adapt to different tasks or domains by observing and learning from a set of related tasks or domains in the meta-training phase. This process helps the modeler develop a general understanding of the underlying structures or patterns that are common across tasks or domains.

2. Task-level learning: At the task level, the model uses learned meta-knowledge to quickly adapt to new, unseen tasks or domains with limited data during the metatest or inference phase. It involves using learned prior knowledge to make predictions or decisions about new tasks or domains with minimal additional training data.

Meta-learning has been used successfully in a variety of fields, including computer vision, natural language processing, and robotics. It shows promising data-scarce scenarios and requires rapid model adaptation and generalization to new tasks or domains with limited data.

27 Model compression and quantization

Model compression and quantization are techniques used to reduce the size or complexity of machine learning models, making them more efficient in terms of storage, memory usage, and computation.

1. Model compression: Model compression techniques aim to reduce the size of a model by reducing its parameter count or memory footprint while maintaining its performance or accuracy. Some common methods of model compression include:

- Pruning: Pruning involves removing redundant or unnecessary weights or neurons from the model. This can be done during training (e.g. by setting small weights to zero) or after training (e.g. by removing neurons with negligible activation values).

- Quantization: Quantization involves reducing the precision of model weights and/or activations from floating-point numbers (eg 32-bit) to a smaller bit representation (eg 8-bit or less). This reduces the memory footprint of the model and can speed up inference on hardware where precision support is limited.

- Distillation of Knowledge. Knowledge distillation involves training a smaller "learner" model to mimic the predictions of a larger "teacher" model. The student model learns to approximate the output of the teacher model, which can often be a more compact representation of the knowledge in the original model.

2. Model quantization. Model quantization techniques aim to represent model weights and/or activations in low-bit representation (e.g., 8-bit or smaller) rather than floating-point numbers (e.g., 32-bit). This reduces the memory footprint of the model and can speed up inference on hardware where accuracy support is limited. Some common methods of pattern quantization include:

- Post-training quantization: Post-training quantization involves post-training quantization of the weights and/or activations of a

trained model. This can be done using techniques such as uniform quantization, where values are quantized to a fixed set of levels, or uneven quantization, where quantization levels are adaptive based on the distribution of the data.

- Quantization Awareness Training: Quantization training involves training a model with the goal of optimizing its performance using quantized weights and/or activations. This may include techniques such as quantization-aware backpropagation, which takes quantization effects into account during gradient computation and weight update, or the use of specialized quantization-aware optimization algorithms.

Model compression and quantization techniques are often used in cases where model size, memory consumption, or computational efficiency are critical, such as deployment on resource-constrained devices such as mobile devices, embedded systems, or edge devices with limited computing power or memory. However, it is important to note that model compression and quantization techniques may change the level of model accuracy or performance in favor of reduced size or complexity, the effectiveness of which depends on the specific use case and application requirements.

28 Neural Architecture Search

Neural Architecture Search (NAS) is the process of using machine learning techniques to automatically design the best neural network architecture for a given task, such as image classification, object recognition, or speech recognition. NAS aims to automate the process of designing neural network

architectures, which is usually done by human experts through trial and error.

The goal of NAS is to find the optimal neural network architecture that can achieve high performance in the target task while reducing the computational cost and model size. NAS algorithms typically search a large number of possible network architectures and use various techniques such as reinforcement learning, genetic algorithms, or evolutionary algorithms to automatically discover promising architectures.

Following are the general steps of the NAS process.

1. Define the search domain. This step specifies the set of possible neural network architectures that the NAS algorithm will search for. A search space typically includes different types of layers, such as convolutional, pooling, and fully connected layers, and their hyperparameters, such as filter size, number of filters, and activation functions.

2. Architecture Search: The NAS algorithm searches a given search area to find promising neural network architectures. This can be done using various methods such as random search, grid search or more advanced methods such as reinforcement learning, genetic algorithms or evolutionary algorithms.

3. Evaluate architectures: After generating a set of candidate architectures, they must be evaluated on the target task to determine their performance. This is typically done by training and evaluating the dataset architecture for the target task, such as training datasets for image classification or speech datasets for speech recognition.

4. Update search strategy: Based on the evaluation results, the NAS algorithm updates its search strategy to generate a new set of candidate architectures. This may involve exploiting promising architectures by generating their variants or exploring new regions of the search space to discover potentially better architectures.

5. Iterative process: The NAS process is usually iterative, where the algorithm searches, evaluates, and updates its search strategy several times until a satisfactory architecture is found or a predefined stopping criterion is met.

Once the NAS process is complete, the resulting discovered architecture can be used to train a neural network model, which can then be used to make inferences about the target task.

29.. One shot and few shot learning.

Single-shot learning and few-frame learning are machine learning paradigms that address the challenge of learning from limited or limited labeled data.

1. One time learning. For one-shot training, the model is trained to recognize new categories or tasks with very limited or even one labeled example at a time. in the category. The idea is to use prior knowledge from other tasks or domains to quickly adapt to new tasks with minimally labeled data. A common approach to one-shot training is to learn similarity measures or similarity functions that can compare new examples with available labeled examples and make predictions based on similarity. Siamese

networks, memory-enhanced neural networks, and prototype-based methods are some of the methods used for one-shot training.

2. Shot learning. Shot learning is a variant of one-shot learning in which a model is trained to recognize new classes or tasks using a small number of labeled examples for each instance. class, usually from a few to dozens of examples. The goal is to learn a model that generalizes well to new classes or tasks, even with limited labeled data. Meta learning, or learning to learn, is a common approach in few-frame learning, where a model is trained to learn to quickly adapt to a new task using a small number of labeled examples. This is typically done by training metamodels on different tasks or domains, then using the learned knowledge to adapt to new tasks with minimally labeled data.

Both one-shot and few-frame learning are challenging tasks because, unlike traditional machine learning methods that rely on large amounts of labeled data, the model needs to learn from limited labeled data.

30 Knowledge Distillation

Knowledge distillation is a technique used in machine learning to transfer knowledge from a complex or large model (often called a "teacher" model) to a simpler or smaller model (often called a "learner" model). The goal of knowledge distillation is to train the student model to mimic the behavior of the teacher model, thus benefiting from the knowledge of the teacher model while reducing the memory footprint and potentially faster inference time.

The basic idea behind knowledge distillation is to use teacher model outputs (eg, predicted probabilities or logistic) as "soft targets" during training, rather than the hard labels (eg, one-time coded labels) typically used in default supervision. Soft targets are more informative than hard targets because they encode the confidence or uncertainty of the teacher model in its predictions. The learner model is then trained to minimize the difference between its predictions and the soft measures produced by the teacher model. This allows the learner model to learn not only from the ground truth labels, but also from the knowledge and insights that the teacher model captures during training. The knowledge distillation process typically involves the following steps:

1. Train teacher models: Train complex or large models on large labeled data sets to achieve high accuracy or performance. This model is used as a source of knowledge to transfer to the student model.

2. Gathering soft measures: Use a trained teacher model to generate soft measures (such as expected probabilities or logistic) for a set of unlabeled or labeled data samples that the learner model will use during training.

3. Training the trained model: Use the labeled dataset and the soft objects generated by the teacher model to train simpler or smaller models (eg, with fewer parameters or layers). A learner model is typically trained to minimize the difference between predictions and soft targets using an appropriate loss function.

4. Refine the learner model. The trained model can optionally be further refined using labeled datasets and ground truth labels

(ie, hard labels) to improve its performance.

3| Quantum Machine learning

Quantum machine learning is an interdisciplinary field that combines concepts from quantum mechanics and machine learning to develop algorithms and models that exploit the properties of quantum systems to solve machine learning tasks. The mathematical foundation of quantum machine learning includes several key concepts, including quantum states, quantum gates, and quantum measurements. Here are some mathematical concepts commonly used in quantum machine learning:

1. Quantum State: In quantum mechanics, a quantum state is a mathematical description that represents the state of a quantum system. It is usually represented as a vector in a complex vector space called Hilbert space. Quantum states are represented as ket vectors, usually denoted by $|\psi\rangle$, where ψ is a vector in Hilbert space.

2. Quantum Gates: Quantum gates are mathematical operators that act on quantum states to change them. They look like classical gates in classical computing, but they operate on quantum states in a way governed by the principles of quantum mechanics. Quantum gates are usually represented as identity matrices, which are square matrices that preserve the norms of the quantum states.

3. Quantum measurements. Quantum measurements are mathematical operations that extract information from quantum states. Unlike classical measurements, quantum measurements are probabilistic, and measurement results are

determined by probabilities associated with different possible measurement outcomes. The results of quantum measurements are often expressed as classical probability distributions.

4. Quantum Circuits: A quantum circuit is a series of quantum gates and measurements used to perform quantum computations. They are similar to classical circuits in classical computing, but they operate on quantum states and use quantum gates for computation.

5. Quantum entanglement: Quantum entanglement is a phenomenon in quantum mechanics where the state of one quantum system is associated with the state of another quantum system even though they are spatially separated. Mathematically, entanglement is described as entangled states that cannot be separated into independent states of each system. Entanglement has been used in quantum machine learning for tasks such as quantum data encoding, quantum state preparation, and quantum function extraction.

6. Quantum Algorithms: Quantum algorithms are mathematical algorithms designed to exploit the properties of quantum systems to solve machine learning tasks. Examples of quantum algorithms used in quantum machine learning include quantum support vector machines, quantum neural networks, and quantum clustering algorithms.

7. Representation of quantum data. Quantum machine learning also involves representing classical data in quantum states. Quantum data representation techniques such as quantum feature maps, quantum embedding, and quantum data encoding

are used to transform classical data into quantum states that can be processed by quantum algorithms.

32 Interpretability and Explainability of Deep learning models

Interpretability and interpretability are important aspects of machine learning models, including deep learning models. They refer to the ability to understand and explain how a model makes predictions or decisions that are essential to building trust, gaining insight, and addressing ethical, legal, and societal issues related to AI systems. Interpretability and explainability are especially important in areas where the decisions made by AI models have a significant impact, such as healthcare, finance and self-driving cars.

Following approaches can improve the explainability and interpretability of dl models :

1. Simpler Model Architecture: By using a simpler architecture with fewer layers and fewer neurons, the model's decision-making process can become more transparent and explainable. Complex models with multiple layers and neurons can be more difficult to interpret because they can learn complex and non-linear relationships between features.

2. Feature Visualization: Visualizing features or representations learned in a deep learning model can provide insight into what the model has learned. Techniques such as activation maps, saliency maps, and feature visualization can help you understand which regions of the input data are important to the model's predictions.

3. Hierarchical interpretation: a deep learning model consists of multiple layers, and each layer learns representations at different levels of abstraction. Analysis of cross-layer activations or outputs can provide insight into how the model processes information and makes predictions at different stages of computation.

4. Attention Mechanism: Attention mechanisms are used in some deep learning models, such as recurrent neural network (RNN) and transformer models, to highlight relevant features or regions of the input. Analysis of attention weights can provide an explanation of the model's decision-making process and reveal which input features the model focuses on to make predictions.
5. ** Post hoc methods: ** Post hoc methods are methods used after the model has been trained to explain the model's predictions. Examples of post hoc methods are locally interpretable model agnostic explanation (LIME), SHAP (SHapley additive explanation), and ensemble gradients, which provide explanations for individual forecasts by approximating the behavior of the model in the local neighborhood of the forecast.

6. Rule-based models or decision trees: Using an interpretable model, such as a decision tree or rule-based model, together with a deep learning model can provide explanations for the predictions of a deep learning model. Rule-based models or decision trees are inherently interpretable because they produce decision rules that are easily understood by humans.

7. Documentation and Document Generation Tools: Provides comprehensive model documentation, including model architecture, training data, and training process, to help improve interpretation and interpretation of deep learning models.

Additionally, tools that automatically generate documentation, such as model maps or model fact sheets, can provide standardized explanations and insights into model performance.

8. Ethical Considerations: Incorporating ethical considerations such as fairness, accountability, and transparency into the design, development, and deployment of deep learning models can also improve their explainability and explainability. Ethical considerations help ensure that model predictions are consistent with social norms and values, and identify and prevent potential bias or discriminatory behavior.

<<<End of Notes