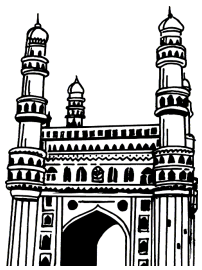
Rahul's ✔
Topper's Voice

AS PER
CBCS SYLLABUS



B.Sc.

III Year VI Sem

Latest **2023** Edition

DEEP LEARNING

DATA SCIENCE PAPER - VII(B)

- ☞ **Study Manual**
- ☞ **Important Questions**
- ☞ **Short Question & Answers**
- ☞ **Choose the Correct Answers**
- ☞ **Fill in the blanks**
- ☞ **One Mark Question & Answers**
- ☞ **Solved Model Papers**

₹.169/-

- by -

WELL EXPERIENCED LECTURER

Rahul Publications™

Hyderabad. Cell : 9391018098, 9505799122

B.Sc.

III Year VI Sem

DEEP LEARNING

DATA SCIENCE PAPER - VII(B)

Inspite of many efforts taken to present this book without errors, some errors might have crept in. Therefore we do not take any legal responsibility for such errors and omissions. However, if they are brought to our notice, they will be corrected in the next edition.

Price ` 169-00

Sole Distributors : **Cell : 9391018098, 9505799122**

VASU BOOK CENTRE

Shop No. 2, Beside Gokul Chat, Koti, Hyderabad.

Maternity Hospital Opp. Lane, Narayan Naik Complex, Koti, Hyderabad.

Near Andhra Bank, Subway, Sultan Bazar, Koti, Hyderabad -195.

DEEP LEARNING

DATA SCIENCE PAPER - VII(B)

CONTENTS

STUDY MANUAL

SOLVED MODEL PAPERS

SYLLABUS

UNIT - I

Introduction: History, Hardware, Data, Algorithms Neural Networks, Data representations for neural networks, Scalars (0D tensors), Vectors (1D tensors), Matrices (2D tensors), 3D tensors and higher-dimensional tensors, Key attributes, Manipulating tensors in NumPy, The notion of data batches, Real-world examples of data tensors, Vector data, Timeseries data or sequence data, Image data, Video data

UNIT - II

Tensor operations: Element-wise operations, Broadcasting, Tensor dot, Tensor reshaping, Geometric interpretation of tensor operations, A geometric interpretation of deep learning

UNIT - III

Gradient-based optimization, Derivative of a tensor operation, Stochastic gradient descent, Chaining derivatives: The Backpropagation algorithm

Neural networks: Anatomy, Layers, Models, Loss functions and optimizers

UNIT - IV

Introduction to Keras, Keras, TensorFlow, Theano, and CNTK Recurrent neural networks: A recurrent layer in Keras, Understanding the LSTM and GRU layers

Contents

Important Questions

UNIT - I

1. Explain the Hardware(System Requirements) for Deep Learning?

Ans:

Refer Unit-I, Q.No. 3

2. Explain the types of data and its use cases.

Ans :

Refer Unit-I, Q.No. 4

3. Explain and List out Deep Learning Algorithms?

Ans:

Refer Unit-I, Q.No. 5

4. What is Neural Networks? Explain in detail about Neural Networks and it types.

Ans :

Refer Unit-I, Q.No. 6

5. Discuss about Data representations for neural networks.

Ans :

Refer Unit-I, Q.No. 7

6. Explain Key Attributes (Essential characteristics) and real world examples of data tensors.

Ans:

Refer Unit-I, Q.No. 8

7. What are the Real-World Examples of Tensors? Give with examples.

Ans:

Refer Unit-I, Q.No. 11

8. What is sequence data and given an example.

Ans:

Refer Unit-I, Q.No. 13

9. What is Image Data? Explain with suitable example.

Ans:

Refer Unit-I, Q.No. 14

UNIT - II

1. Define Tensor? What Does Element-Wise Operations with examples.

Ans:

Refer Unit-II, Q.No. 1

2. Write about Broadcasting in Tensors with suitable examples?

Ans:

Refer Unit-II, Q.No. 2

3. Explain about Reshaping a Tensor with suitable Examples?

Ans:

Refer Unit-II, Q.No. 4

4. Discuss about Geometric interpretation of tensor operations?

Ans:

Refer Unit-II, Q.No. 5

5. Explain briefly about Geometric Interpretation of Deep Learning.

Ans:

Refer Unit-II, Q.No. 6

UNIT - III

1. Discuss about Gradient-based optimization.

Ans :

Refer Unit-III, Q.No. 1

2. Explain about Stochastic Gradient Descent with suitable examples

Ans :

Refer Unit-III, Q.No. 3

3. Explain in detail about The Backpropagation algorithm.

Ans :

Refer Unit-III, Q.No. 4

4. Write a note on Anatomy of a neural network.

Ans :

Refer Unit-III, Q.No. 5

5. Discuss about Layers in Neural Networks.

Ans :

Refer Unit-III, Q.No. 6

6. Discuss about Neural Network Models.

Ans :

Refer Unit-III, Q.No. 7

7. Explain in detail about Loss functions and Optimizers.

Ans :

Refer Unit-III, Q.No. 8

UNIT - IV

1. Explain in detail about Recurrent neural networks (RNN's)? and working of RNN's

Ans :

Refer Unit-IV, Q.No. 7

2. Write about recurrent layer in Keras

Ans:

Refer Unit-IV, Q.No. 8

3. Explain in detail about LSTM layers?

Ans :

Refer Unit-IV, Q.No. 9

4. Write a short note on GRU (Gated Recur-rent Unit) with suitable illustrations.

Ans :

Refer Unit-IV, Q.No. 10

UNIT I

Introduction: History, Hardware, Data, Algorithms Neural Networks, Data representations for neural networks, Scalars (0D tensors), Vectors (1D tensors), Matrices (2D tensors), 3D tensors and higher-dimensional tensors, Key attributes, Manipulating tensors in NumPy, The notion of data batches, Real-world examples of data tensors, Vector data, Timeseries data or sequence data, Image data, Video data

1.1 Introduction to Deep Learning

Q1. Explain in detail about deep learning.

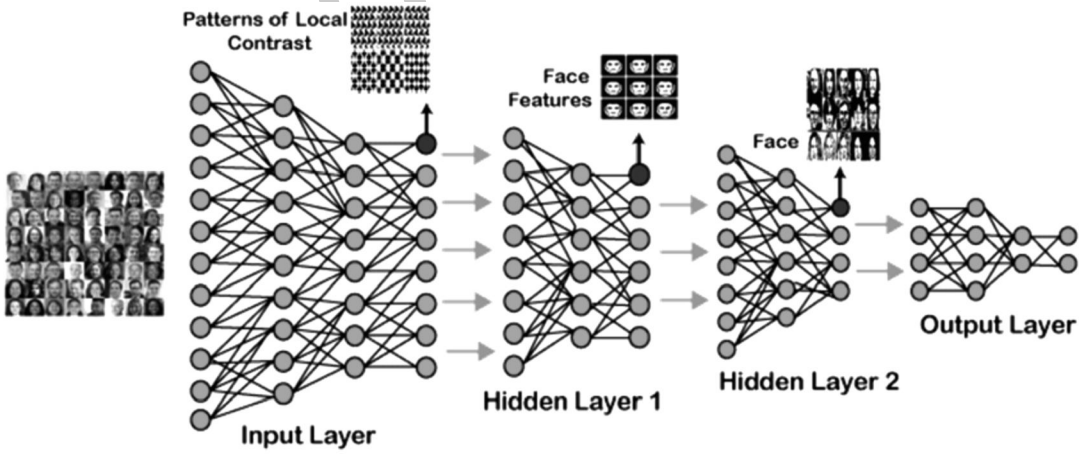
Ans :

Deep Learning

Deep learning is a branch of machine learning which is completely based on artificial neural networks, as neural network is going to mimic the human brain so deep learning is also a kind of mimic of human brain. In deep learning, we don't need to explicitly program everything.

"Deep learning is a collection of statistical techniques of machine learning for learning feature hierarchies that are actually based on artificial neural networks."

Example



In the example given above, we provide the raw data of images to the first layer of the input layer. After then, these input layer will determine the patterns of local contrast that means it will differentiate on the basis of colors, luminosity, etc. Then the 1st hidden layer will determine the face feature, i.e., it will fixate on eyes, nose, and lips, etc. And then, it will fixate those face features on the correct face template. So, in the 2nd hidden layer, it will actually determine the correct face here as it can be seen in the above image, after which it will be sent to the output layer. Likewise, more hidden layers can be added to solve more complex problems, for example, if you want to find out a particular kind of face having large or light complexions. So, as and when the hidden layers increase, we are able to solve complex problems.

1.2 History

Q2. Explain the history of Deep Learning.

(OR)

Explain the evolution of Deep Learning.

Ans :

The history of deep learning can be traced back to 1943, when Walter Pitts and Warren McCulloch created a computer model based on the neural networks of the human brain.

They used a combination of algorithms and mathematics they called "threshold logic" to mimic the thought process. Since that time, Deep Learning has evolved steadily, with only two significant breaks in its development. Both were tied to the infamous Artificial Intelligence winters.

The 1960's

Henry J. Kelley is given credit for developing the basics of a continuous Back Propagation Model in 1960. In 1962, a simpler version based only on the chain rule was developed by Stuart Dreyfus. While the concept of back propagation (the backward propagation of errors for purposes of training) did exist in the early 1960s, it was clumsy and inefficient, and would not become useful until 1985.

The 1970's

During the 1970's the first AI winter kicked in, the result of promises that couldn't be kept. The impact of this lack of funding limited both DL and AI research. Fortunately, there were individuals who carried on the research without funding.

The first "convolutional neural networks" were used by Kunihiko Fukushima. Fukushima designed neural networks with multiple pooling and convolutional layers. In 1979, he developed an artificial neural network, called Neocognitron, which used a hierarchical, multilayered design. This design allowed the computer "learn" to recognize visual patterns.

The 1980's and 90's

In 1989, Yann LeCun provided the first practical demonstration of backpropagation at Bell Labs. He combined convolutional neural networks with back propagation onto read "handwritten" digits. This system was eventually used to read the numbers of handwritten checks.

The 2000's and 2010's

Around the year 2000, The Vanishing Gradient Problem appeared. It was discovered "features" formed in lower layers were not being learned by the upper layers, because no learning signal reached these layers. This was not a fundamental problem for all neural networks, just the ones with gradient-based learning methods. The source of the problem turned out to be certain activation functions. A number of activation functions condensed their input, in turn reducing the output range in a somewhat chaotic fashion.

This produced large areas of input mapped over an extremely small range. In these areas of input, a large change will be reduced to a small change in the output, resulting in a vanishing gradient. Two solutions used to solve this problem were layer-by-layer pre-training and the development of long short-term memory.

In 2001, a research report by META Group (now called Gartner) described he challenges and opportunities of data growth as three-dimensional. The report described the increasing volume of data and the increasing speed of data as increasing the range of data sources and types. This was a call to prepare for the onslaught of Big Data, which was just starting.

In 2009, Fei-Fei Li, an AI professor at Stanford launched ImageNet, assembled a free database of more than 14 million labeled images. The Internet is, and was, full of unlabeled images. Labeled images were needed to "train" neural nets. Professor Li said, "Our vision was that big data would change the way machine learning works. Data drives learning."

The 2011's and 2020

In 2011, the speed of GPUs had increased significantly, making it possible to train convolutional neural networks "without" the layer-by-layer pre-training. With the increased computing speed, it

became obvious deep learning had significant advantages in terms of efficiency and speed. One example is AlexNet, a convolutional neural network whose architecture won several international competitions during 2011 and 2012. Rectified linear units were used to enhance the speed and dropout.

In 2012, Google Brain released the results of an unusual project known as The Cat Experiment. The free-spirited project explored the difficulties of "unsupervised learning." Deep learning uses "supervised learning," meaning the convolutional neural net is trained using labeled data (think images from ImageNet). Using unsupervised learning, a convolutional neural net is given unlabeled data, and is then asked to seek out recurring patterns.

The Cat Experiment used a neural net spread over 1,000 computers. Ten million "unlabeled" images were taken randomly from YouTube, shown to the system, and then the training software was allowed to run. At the end of the training, one neuron in the highest layer was found to respond strongly to the images of cats. Andrew Ng, the project's founder said, "We also found a neuron that responded very strongly to human faces." Unsupervised learning remains a significant goal in the field of deep learning.

1.3 HARDWARE

Q3. Explain the Hardware(System Requirements) for Deep Learning?

Ans: **(Imp.)**

System Requirements

ENVI Deep Learning 2.0 uses TensorFlow version 2.9 and CUDA version 11.2.2, both of which are included in the installation. System requirements are as follows:

Base software

ENVI 5.6.3 and the ENVI Deep Learning 2.0 module.

Operating systems

- Windows 10 and 11 (Intel/AMD 64-bit)
- Linux (Intel/AMD 64-bit, kernel 3.10.0 or higher, glibc 2.17 or higher)

Hardware

- NVIDIA graphics card with CUDA Compute Capability version 3.5 to 8.6. See the list of CUDA-enabled GPU cards. A minimum of 8 GB of GPU memory is recommended for optimal performance, particularly when training deep learning models.
- NVIDIA GPU driver version: Windows 461.33 or higher, Linux 460.32.03 or higher.
- A CPU with the Advanced Vector Extensions (AVX) instruction set. In general, any CPU after 2011 will contain this instruction set.
- Intel CPUs are recommended, though not required. They have an optimized Intel Machine Learning library that offers performance gains for certain Machine Learning algorithms.

To determine if your system meets the requirements for ENVI Deep Learning, start the Deep Learning Guide Map in the ENVI Toolbox. From the Deep Learning Guide Map menu bar, select Tools > Test Installation and Configuration.

1.4 DATA

Q4. Explain in detail about Data for Deep Learning?

(OR)

Explain the types of data and its use cases.

Ans : **(Imp.)**

Data for Deep Learning

The minimum requirements to successfully apply deep learning depends on the problem you're trying to solve. In contrast to static, benchmark datasets like MNIST and CIFAR-10, real-world data is messy, varied and evolving, and that is the data practical deep learning solutions must deal with.

Gather your data wisely. Make sure it's relevant. Neural nets start out knowing nothing. They only learn about what you expose them to. They can't learn facial recognition by training on flowers.

Data pipelines mean "ETL": The extraction, transformation and loading of data into neural nets. This is also called preprocessing. Data should be given a standard format (e.g. make all images into jpegs with the same height and width), normalized, and turned into vectors.

Neural nets have knobs called hyperparameters that must be properly adjusted, or tuned, before a net can train on data, learning its structure to produce an accurate model for classification or clustering. Finding the right combination of hyperparameters requires tuning, training and tuning again...

A collection of pretrained models is called a "model zoo." These models can already classify or cluster data accurately. The work of tuning and training is done.

Deep learning can perform fraud detection, image recognition, speech to text, machine translation, predictive analytics for customers, hardware, health. Integrated with Hadoop, Spark, Kafka, Web apps.

Types of Data

Deep learning can be applied to any data type. The data types you work with, and the data you gather, will depend on the problem you're trying to solve.

1. Sound (Voice Recognition)
2. Text (Classifying Reviews)
3. Images (Computer Vision)
4. Time Series (Sensor Data, Web Activity)
5. Video (Motion Detection)

Use Cases

Deep learning can solve almost any problem of machine perception, including classifying data , clustering it, or making predictions about it.

- Classification: This image represents a horse; this email looks like spam; this transaction is fraudulent
- Clustering: These two sounds are similar. This document is probably what user X is looking for
- Predictions: Given their web log activity, Customer A looks like they are going to stop using your service

Deep learning is best applied to unstructured data like images, video, sound or text. An image is just a blob of pixels, a message is just a blob of text. This data is not organized in a typical, relational database by rows and columns. That makes it more difficult to specify its features manually.

Common use cases for deep learning include sentiment analysis, classifying images, predictive analytics, recommendation systems, anomaly detection and more.

Data Attributes: For deep learning to succeed, your data needs to have certain characteristics.

Relevancy

The data you use to train your neural net must be directly relevant to your problem; that is, it must resemble as much as possible the real-world data you hope to process. Neural networks are born as blank slates, and they only learn what you teach them. If you want them to solve a problem involving certain kinds of data, like CCTV video, then you have to train them on CCTV video, or something similar to it. The training data should resemble the real-world data that they will classify in production.

Proper Classification

If a client wants to build a deep-learning solution that classifies data, then they need to have a labeled dataset. That is, someone needs to apply labels to the raw data: "This image is a flower, that image

is a panda." With time and tuning, this training dataset can teach a neural network to classify new images it has not seen before.

Formatting

Neural networks eat vectors of data and split out decisions about those vectors. All data needs to be vectorized, and the vectors should be the same length when they enter the neural net. To get vectors of the same length, it's helpful to have, say, images of the same size (the same height and width). So sometimes you need to resize the images. This is called data pre-processing.

Accessibility

The data needs to be stored in a place that's easy to work with. A local file system, or HDFS (the Hadoop file system), or an S3 bucket on AWS, for example. If the data is stored in many different databases that are unconnected, you will have to build data pipelines. Building data pipelines and performing preprocessing can account for at least half the time you spend building deep-learning solutions.

Minimum Data Requirement

The minimums vary with the complexity of the problem, but 100,000 instances in total, across all categories, is a good place to start.If you have labeled data (i.e. categories A, B, C and D), it's preferable to have an evenly balanced dataset with 25,000 instances of each label; that is, 25,000 instances of A, 25,000 instances of B and so forth.

1.5 ALGORITHMS

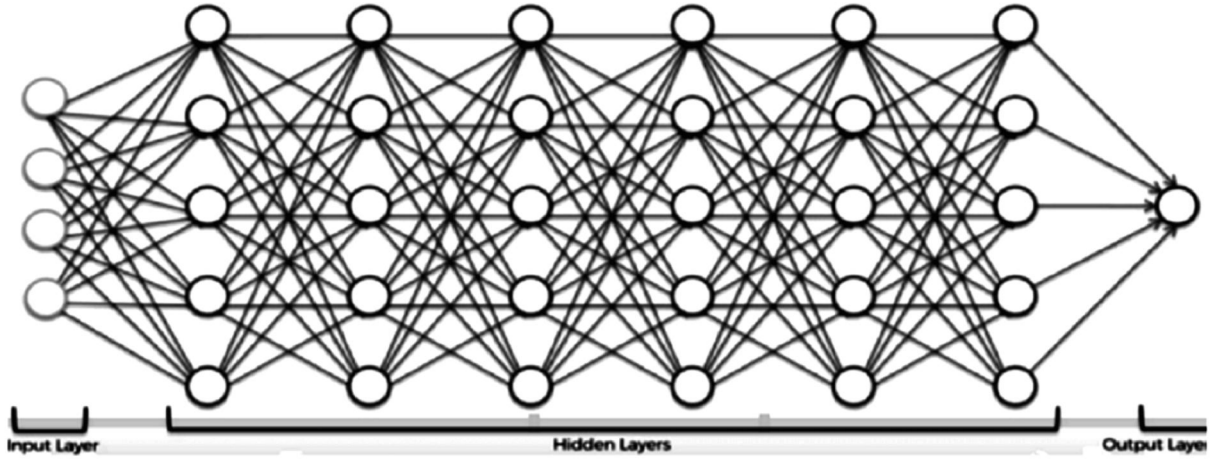
Q5. Explain and List out Deep Learning Algorithms?

Ans: **(Imp.)**

Deep learning is a subset of a Machine Learning algorithm that uses multiple layers of neural networks to perform in processing data and computations on a large amount of data. Deep learning algorithm works based on the function and working of the human brain.

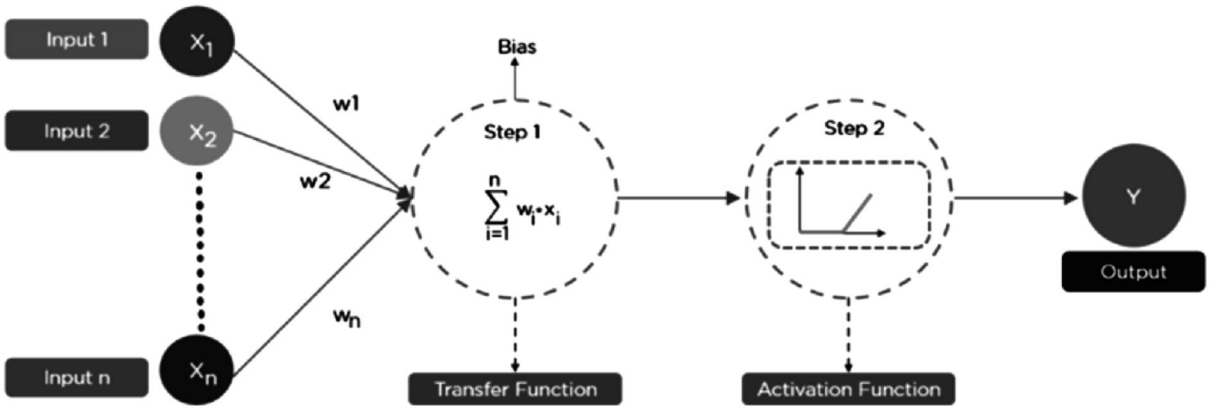
The deep learning algorithm is capable to learn without human supervision, can be used for both structured and unstructured types of data. Deep learning can be used in various industries like healthcare, finance, banking, e-commerce, etc.

Deep Learning algorithms working depends upon Neural network just like the human brain computes information using millions of neurons.



A neural network is structured like the human brain and consists of artificial neurons, also known as nodes. These nodes are stacked next to each other in three layers:

- **Input layer:** The input layer has input features a dataset that is known to us.
- **Hidden Layer :** Hidden layer, just like we need to train the brain through hidden neurons.
- **Output layer :** value that we want to classify



Data provides each node with information in the form of inputs. The node multiplies the inputs with random weights, calculates them, and adds a bias. Finally, nonlinear functions, also known as activation functions, are applied to determine which neuron to fire.

Working of Deep Learning Algorithms

While deep learning algorithms feature self-learning representations, they depend upon ANNs that mirror the way the brain computes information. During the training process, algorithms use unknown elements in the input distribution to extract features, group objects, and discover useful data patterns. Much like training machines for self-learning, this occurs at multiple levels, using the algorithms to build the models.

Deep learning models make use of several algorithms.While no one network is considered perfect, some algorithms are better suited to perform specific tasks. To choose the right ones, it's good to gain a solid understanding of all primary algorithms.

Types of Algorithms used in Deep Learning

Here is the list of top 10 most popular deep learning algorithms:

1. Convolutional Neural Networks (CNNs)
2. Long Short Term Memory Networks (LSTMs)
3. Recurrent Neural Networks (RNNs)
4. Generative Adversarial Networks (GANs)
5. Radial Basis Function Networks (RBFNs)
6. Multilayer Perceptrons (MLPs)
7. Self Organizing Maps (SOMs)
8. Deep Belief Networks (DBNs)

1. Restricted Boltzmann Machines (RBMs)
2. Autoencoders

Deep learning algorithms work with almost any kind of data and require large amounts of computing power and information to solve complicated issues. Now, let us, deep-dive, into the top 10 deep learning algorithms.

1. Convolutional Neural Networks (CNNs): CNN's, also known as ConvNets, consist of multiple layers and are mainly used for image processing and object detection. Yann LeCun developed the first CNN in 1988 when it was called LeNet. It was used for recognizing characters like ZIP codes and digits.

CNN's are widely used to identify satellite images, process medical images, forecast time series, and detect anomalies.

Working of CNNs:

CNN's have multiple layers that process and extract features from data:

Convolution Layer

- CNN has a convolution layer that has several filters to perform the convolution operation.

Rectified Linear Unit (ReLU)

- CNN's have a ReLU layer to perform operations on elements. The output is a rectified feature map.

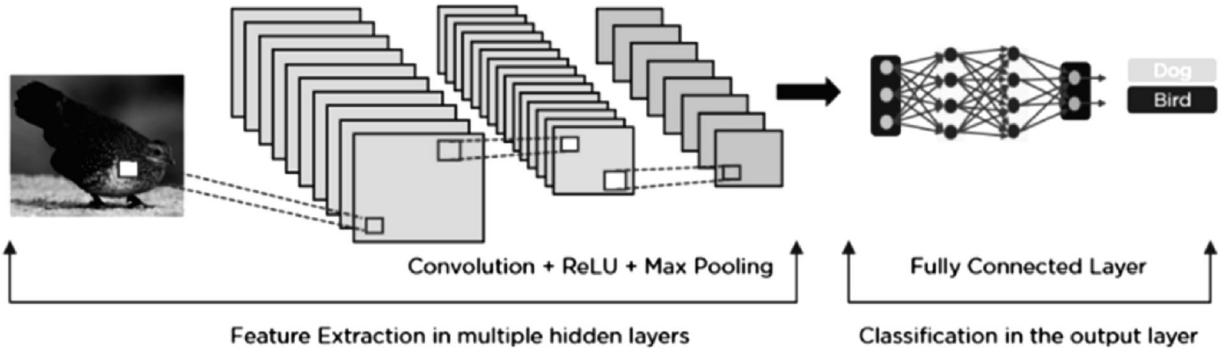
Pooling Layer

- The rectified feature map next feeds into a pooling layer. Pooling is a down-sampling operation that reduces the dimensions of the feature map.
- The pooling layer then converts the resulting two-dimensional arrays from the pooled feature map into a single, long, continuous, linear vector by flattening it.

Fully Connected Layer

- A fully connected layer forms when the flattened matrix from the pooling layer is fed as an input, which classifies and identifies the images.

Below is an example of an image processed via CNN.



2. Long Short-Term Memory Networks (LSTMs): LSTMs are a type of Recurrent Neural Network (RNN) that can learn and memorize long-term dependencies. Recalling past information for long periods is the default behavior.

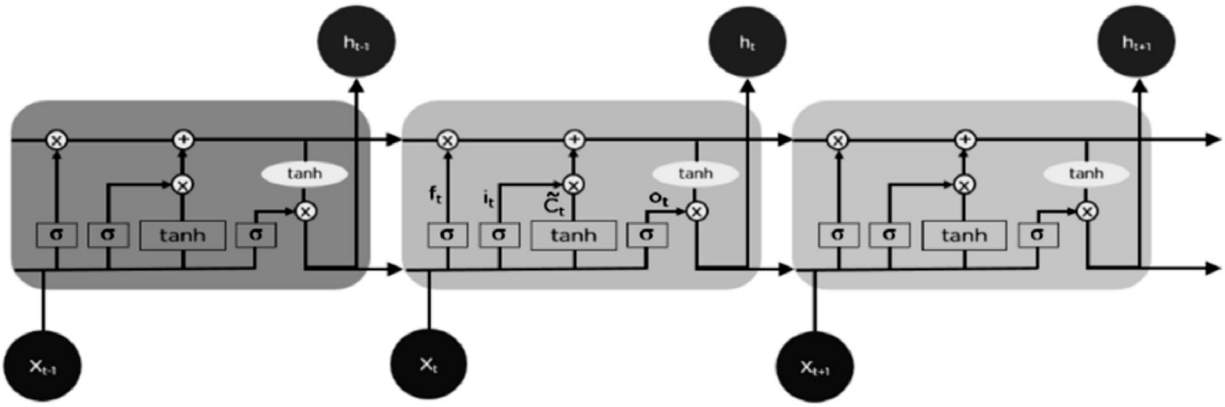
LSTMs retain information over time. They are useful in time-series prediction because they remem-

ber previous inputs. LSTMs have a chain-like structure where four interacting layers communicate in a unique way. Besides time-series predictions, LSTMs are typically used for speech recognition, music composition, and pharmaceutical development.

Working of LSTMs

- First, they forget irrelevant parts of the previous state
- Next, they selectively update the cell-state values
- Finally, the output of certain parts of the cell state

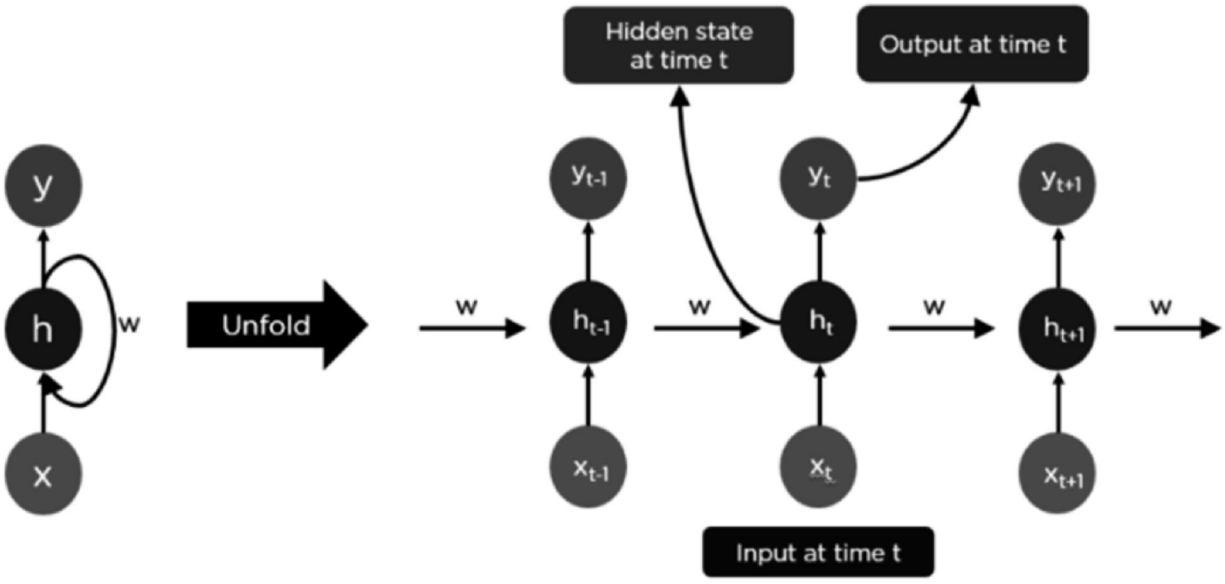
Below is a diagram of how LSTMs operate:



3. Recurrent Neural Networks (RNNs): RNNs have connections that form directed cycles, which allow the outputs from the LSTM to be fed as inputs to the current phase.

The output from the LSTM becomes an input to the current phase and can memorize previous inputs due to its internal memory. RNNs are commonly used for image captioning, time-series analysis, natural-language processing, handwriting recognition, and machine translation.

An unfolded RNN looks like this:

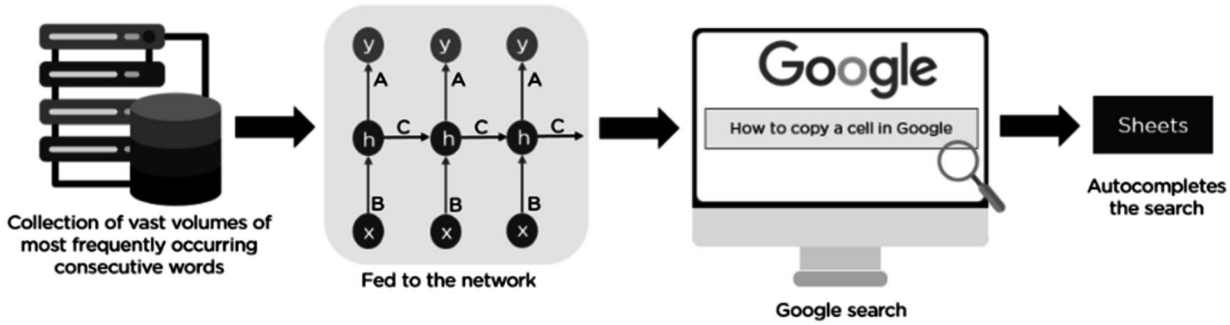


Working of RNNs

The output at time t-1 feeds into the input at time t.

- Similarly, the output at time t feeds into the input at time t+1.
- RNNs can process inputs of any length.
- The computation accounts for historical information, and the model size does not increase with the input size.

Here is an example of how Google's autocompleting feature works :



4. Generative Adversarial Networks (GANs): GANs are generative deep learning algorithms that create new data instances that resemble the training data. GAN has two components: a generator, which learns to generate fake data, and a discriminator, which learns from that false information.

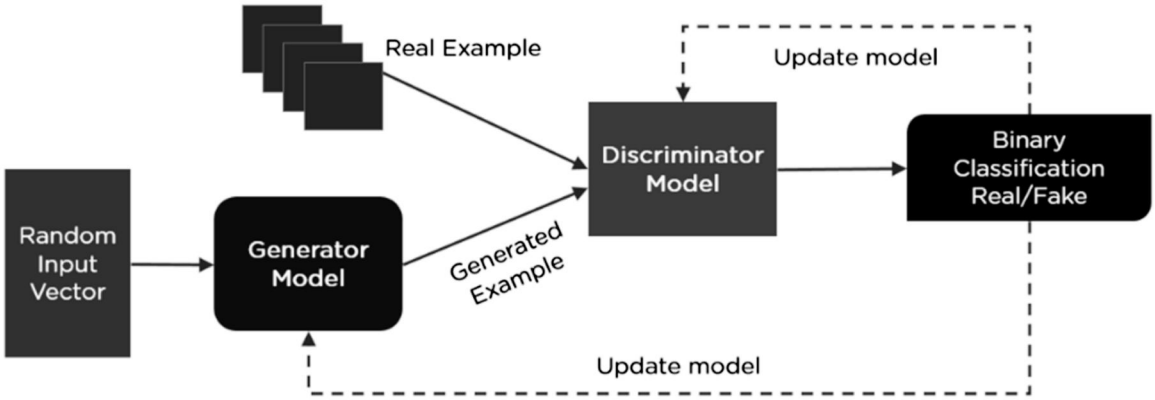
The usage of GANs has increased over a period of time. They can be used to improve astronomical images and simulate gravitational lensing for dark-matter research. Video game developers use GANs to upscale low-resolution, 2D textures in old video games by recreating them in 4K or higher resolutions via image training.

GANs help generate realistic images and cartoon characters, create photographs of human faces, and render 3D objects.

Working of GANs

- The discriminator learns to distinguish between the generator's fake data and the real sample data.
- During the initial training, the generator produces fake data, and the discriminator quickly learns to tell that it's false.
- The GAN sends the results to the generator and the discriminator to update the model.

Below is a diagram of how GANs operate:



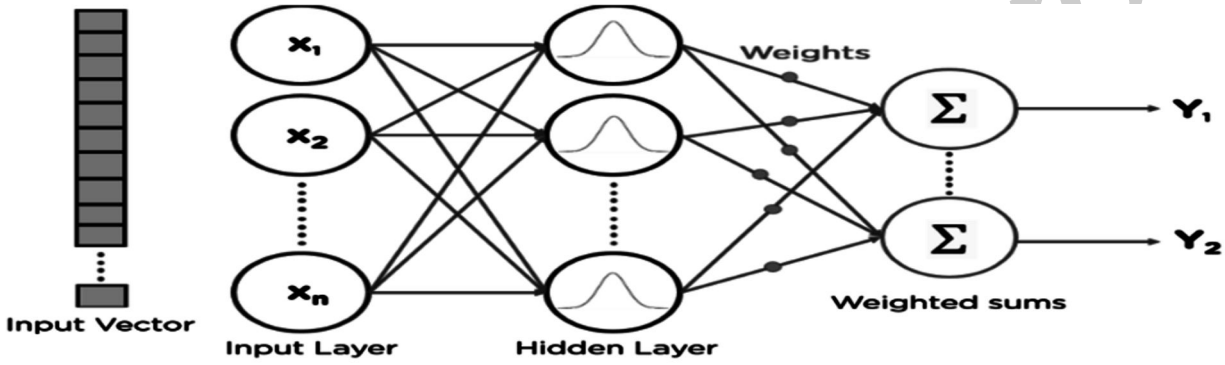
5. Radial Basis Function Networks (RBFNs): RBFNs are special types of feedforward neural networks that use radial basis functions as activation functions. They have an input layer, a hidden layer, and an output layer and are mostly used for classification, regression, and time-series prediction.

Working of RBFNs

RBFNs perform classification by measuring the input's similarity to examples from the training set.

- RBFNs have an input vector that feeds to the input layer. They have a layer of RBF neurons.
- The function finds the weighted sum of the inputs, and the output layer has one node per category or class of data.
- The neurons in the hidden layer contain the Gaussian transfer functions, which have outputs that are inversely proportional to the distance from the neuron's center.
- The network's output is a linear combination of the input's radial-basis functions and the neuron's parameters.

Example of an RBFN



6. Multilayer perceptron's (MLPs): MLPs are an excellent place to start learning about deep learning technology.

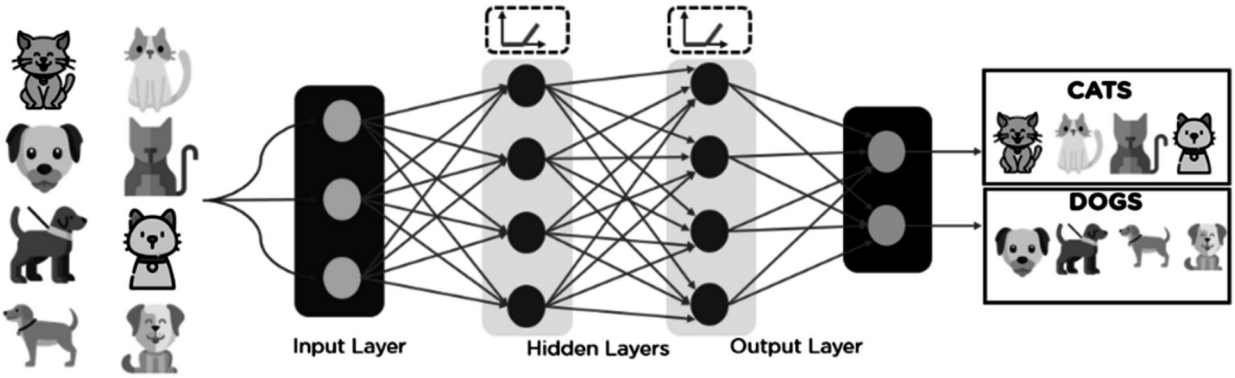
MLPs belong to the class of feedforward neural networks with multiple layers of perceptrons that have activation functions. MLPs consist of an input layer and an output layer that are fully connected. They have the same number of input and output layers but may have multiple hidden layers and can be used to build speech-recognition, image-recognition, and machine-translation software.

Working of MLPs

MLPs feed the data to the input layer of the network. The layers of neurons connect in a graph so that the signal passes in one direction.

- MLPs compute the input with the weights that exist between the input layer and the hidden layers.
- MLPs use activation functions to determine which nodes to fire. Activation functions include ReLUs, sigmoid functions, and tanh.
- MLPs train the model to understand the correlation and learn the dependencies between the independent and the target variables from a training data set.

Below is an example of an MLP. The diagram computes weights and bias and applies suitable activation functions to classify images of cats and dogs.



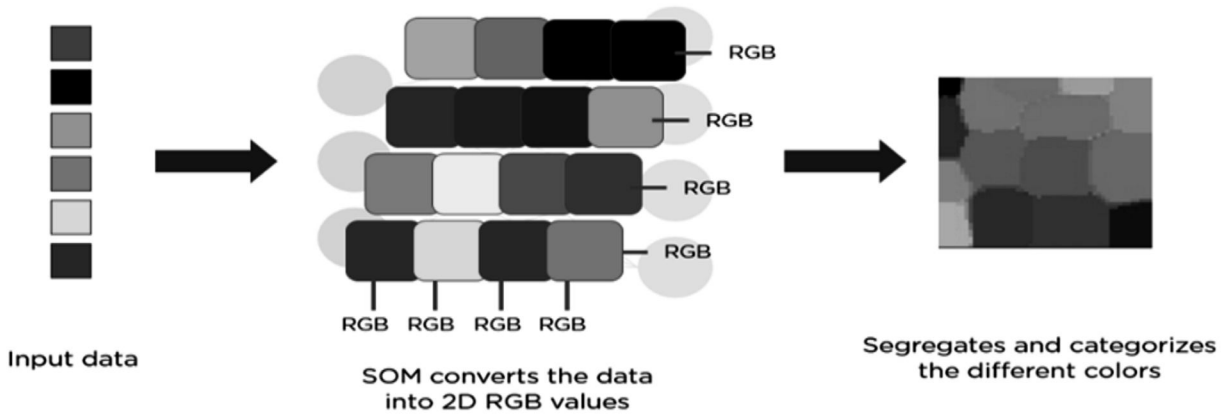
7. Self-Organizing Maps (SOMs): Professor TeuvoKohonen invented SOMs, which enable data visualization to reduce the dimensions of data through self-organizing artificial neural networks.

Data visualization attempts to solve the problem that humans cannot easily visualize high-dimensional data. SOMs are created to help users understand this high-dimensional information.

Working Of SOMs:SOMs initialize weights for each node and choose a vector at random from the training data.

- SOMs examine every node to find which weights are the most likely input vector. The winning node is called the Best Matching Unit (BMU).
- SOMs discover the BMU's neighborhood, and the amount of neighbors lessens over time.
- SOMs award a winning weight to the sample vector. The closer a node is to a BMU, the more its weight changes..
- The further the neighbor is from the BMU, the less it learns. SOMs repeat step two for N iterations.

Below, see a diagram of an input vector of different colors. This data feeds to a SOM, which then converts the data into 2D RGB values. Finally, it separates and categorizes the different colors.



8. Deep Belief Networks (DBNs): DBNs are generative models that consist of multiple layers of stochastic, latent variables. The latent variables have binary values and are often called hidden units.

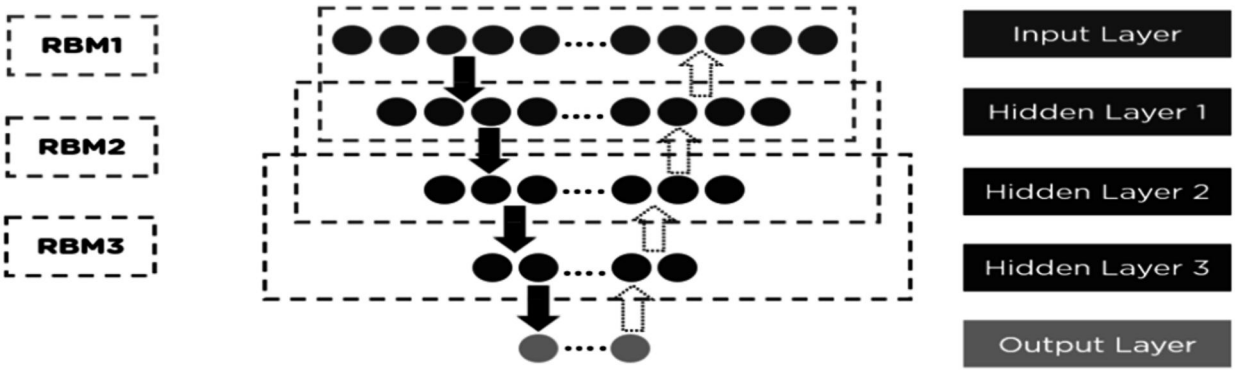
DBNs are a stack of Boltzmann Machines with connections between the layers, and each RBM layer communicates with both the previous and subsequent layers. Deep Belief Networks (DBNs) are used for image-recognition, video-recognition, and motion-capture data.

Working of DBNs

Greedy learning algorithms train DBNs. The greedy learning algorithm uses a layer-by-layer approach for learning the top-down, generative weights.

- DBNs run the steps of Gibbs sampling on the top two hidden layers. This stage draws a sample from the RBM defined by the top two hidden layers.
- DBNs draw a sample from the visible units using a single pass of ancestral sampling through the rest of the model.
- DBNs learn that the values of the latent variables in every layer can be inferred by a single, bottom-up pass.

Below is an example of DBN architecture:



9. Restricted Boltzmann Machines (RBMs): Developed by Geoffrey Hinton, RBMs are stochastic neural networks that can learn from a probability distribution over a set of inputs.

This deep learning algorithm is used for dimensionality reduction, classification, regression, collaborative filtering, feature learning, and topic modeling. RBMs constitute the building blocks of DBNs.

RBMs consist of two layers:

- Visible units
- Hidden units

Each visible unit is connected to all hidden units. RBMs have a bias unit that is connected to all the visible units and the hidden units, and they have no output nodes.

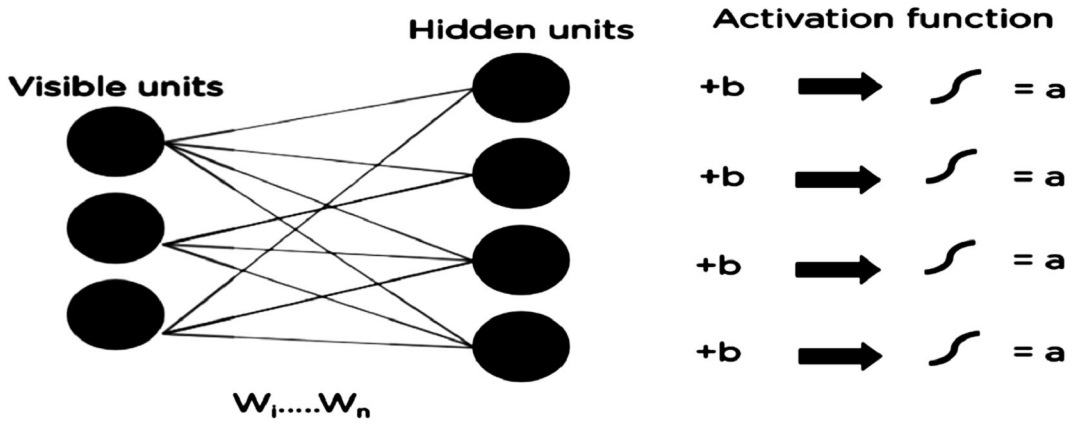
Working of RBMs

RBMs have two phases: forward pass and backward pass.

- RBMs accept the inputs and translate them into a set of numbers that encodes the inputs in the forward pass.
- RBMs combine every input with individual weight and one overall bias. The algorithm passes the output to the hidden layer.
- In the backward pass, RBMs take that set of numbers and translate them to form the reconstructed inputs.
- RBMs combine each activation with individual weight and overall bias and pass the output to the visible layer for reconstruction.

- At the visible layer, the RBM compares the reconstruction with the original input to analyze the quality of the result.

  Below is a diagram of how RBMs function:



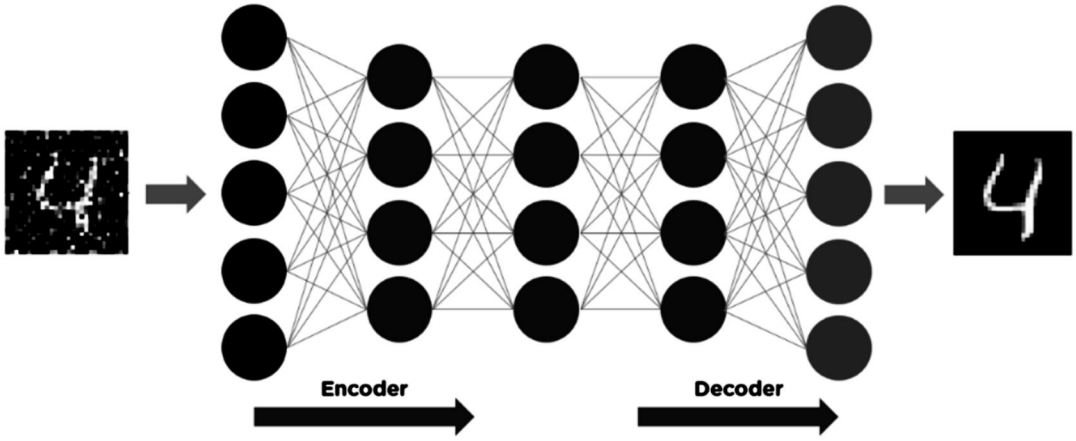
10. Autoencoders: Autoencoders are a specific type of feedforward neural network in which the input and output are identical. Geoffrey Hinton designed autoencoders in the 1980s to solve unsupervised learning problems. They are trained neural networks that replicate the data from the input layer to the output layer. Autoencoders are used for purposes such as pharmaceutical discovery, popularity prediction, and image processing.

Working of Autoencoders

An autoencoder consists of three main components: the encoder, the code, and the decoder.

- Autoencoders are structured to receive an input and transform it into a different representation. They then attempt to reconstruct the original input as accurately as possible.
- When an image of a digit is not clearly visible, it feeds to an autoencoder neural network.
- Autoencoders first encode the image, then reduce the size of the input into a smaller representation.
- Finally, the autoencoder decodes the image to generate the reconstructed image.

The following image demonstrates how autoencoders operate:



1.6 Neural Networks

Q6. What is Neural Networks? Explain in detail about Neural Networks and it types.

Ans : **(Imp.)**

A neural network is a method in artificial intelligence that teaches computers to process data in a way that is inspired by the human brain. It is a type of machine learning process, called deep learning, that uses interconnected nodes or neurons in a layered structure that resembles the human brain. It creates an adaptive system that computers use to learn from their mistakes and improve continuously. Thus, artificial neural networks attempt to solve complicated problems, like summarizing documents or recognizing faces, with greater accuracy.

Importance of Neural Networks

Neural networks can help computers make intelligent decisions with limited human assistance. This is because they can learn and model the relationships between input and output data that are nonlinear and complex. For instance, they can do the following tasks.

Make generalizations and inferences

Neural networks can comprehend unstructured data and make general observations without explicit training. For instance, they can recognize that two different input sentences have a similar meaning:

- Can you tell me how to make the payment?
- How do I transfer money?

A neural network would know that both sentences mean the same thing. Or it would be able to broadly recognize that Sardar Patel Road is a place, but Sardar Patel is a person's name.

Uses of Neural Networks:

Neural networks have several use cases across many industries, such as the following:

- Medical diagnosis by medical image classification
- Targeted marketing by social network filtering and behavioral data analysis
- Financial predictions by processing historical data of financial instruments
- Electrical load and energy demand forecasting
- Process and quality control
- Chemical compound identification

We give four of the important applications of neural networks below.

Computer vision

Computer vision is the ability of computers to extract information and insights from images and videos. With neural networks, computers can distinguish and recognize images similar to humans. Computer vision has several applications, such as the following:

- Visual recognition in self-driving cars so they can recognize road signs and other road users
- Content moderation to automatically remove unsafe or inappropriate content from image and video archives
- Facial recognition to identify faces and recognize attributes like open eyes, glasses, and facial hair
- Image labeling to identify brand logos, clothing, safety gear, and other image details

Speech recognition

Neural networks can analyze human speech despite varying speech patterns, pitch, tone, language, and accent. Virtual assistants like Amazon Alexa and automatic transcription software use speech recognition to do tasks like these:

- Assist call center agents and automatically classify calls
- Convert clinical conversations into documentation in real time
- Accurately subtitle videos and meeting recordings for wider content reach

Natural language processing

Natural language processing (NLP) is the ability to process natural, human-created text. Neural

networks help computers gather insights and meaning from text data and documents. NLP has several use cases, including in these functions:

- Automated virtual agents and chatbots
- Automatic organization and classification of written data
- Business intelligence analysis of long-form documents like emails and forms
- Indexing of key phrases that indicate sentiment, like positive and negative comments on social media
- Document summarization and article generation for a given topic

Recommendation engines

Neural networks can track user activity to develop personalized recommendations. They can also analyze all user behavior and discover new products or services that interest a specific user. For example, Curalate, a Philadelphia-based startup, helps brands convert social media posts into sales. Brands use Curalate's intelligent product tagging (IPT) service to automate the collection and curation of user-generated social content. IPT uses neural networks to automatically find and recommend products relevant to the user's social media activity. Consumers don't have to hunt through online catalogs to find a specific product from a social media image. Instead, they can use Curalate's auto product tagging to purchase the product with ease.

Working of Neural Networks

The human brain is the inspiration behind neural network architecture. Human brain cells, called neurons, form a complex, highly interconnected network and send electrical signals to each other to help humans process information. Similarly, an artificial neural network is made of artificial neurons that work together to solve a problem. Artificial neurons are software modules, called nodes, and artificial neural networks are software programs or algorithms that, at their core, use computing systems to solve mathematical calculations.

Simple neural network architecture

A basic neural network has interconnected artificial neurons in three layers:

Input Layer

Information from the outside world enters the artificial neural network from the input layer. Input nodes process the data, analyze or categorize it, and pass it on to the next layer.

Hidden Layer

Hidden layers take their input from the input layer or other hidden layers. Artificial neural networks can have a large number of hidden layers. Each hidden layer analyzes the output from the previous layer, processes it further, and passes it on to the next layer.

Output Layer

The output layer gives the final result of all the data processing by the artificial neural network. It can have single or multiple nodes. For instance, if we have a binary (yes/no) classification problem, the output layer will have one output node, which will give the result as 1 or 0. However, if we have a multi-class classification problem, the output layer might consist of more than one output node.

Types of Neural Networks

Artificial neural networks can be categorized by how the data flows from the input node to the output node. Below are some examples:

Feedforward neural networks

Feedforward neural networks process data in one direction, from the input node to the output node. Every node in one layer is connected to every node in the next layer. A feedforward network uses a feedback process to improve predictions over time.

Backpropagation algorithm

Artificial neural networks learn continuously by using corrective feedback loops to improve their predictive analytics. In simple terms, you can think of the data flowing from the input node to the output node through many different paths in the neural network. Only one path is the correct one that maps the input node to the correct output node. To find this path, the neural network uses a feedback loop, which works as follows:

Each node makes a guess about the next node in the path.

It checks if the guess was correct. Nodes assign higher weight values to paths that lead to more correct guesses and lower weight values to node paths that lead to incorrect guesses.

For the next data point, the nodes make a new prediction using the higher weight paths and then repeat Step 1.

Convolutional neural networks

The hidden layers in convolutional neural networks perform specific mathematical functions, like summarizing or filtering, called convolutions. They are very useful for image classification because they can extract relevant features from images that are useful for image recognition and classification. The new form is easier to process without losing features that are critical for making a good prediction. Each hidden layer extracts and processes different image features, like edges, color, and depth.

Training of Neural Networks

Neural network training is the process of teaching a neural network to perform a task. Neural networks learn by initially processing several large sets of labeled or unlabeled data. By using these examples, they can then process unknown inputs more accurately.

Supervised learning

In supervised learning, data scientists give artificial neural networks labeled datasets that provide the right answer in advance. For example, a deep learning network training in facial recognition initially processes hundreds of thousands of images of human faces, with various terms related to ethnic origin, country, or emotion describing each image.

The neural network slowly builds knowledge from these datasets, which provide the right answer in advance. After the network has been trained, it starts making guesses about the ethnic origin or emotion of a new image of a human face that it has never processed before.

1.7 Data Representations for Neural Networks

Q7. Discuss about Data representations for neural networks.

Ans : **(Imp.)**

A deep learning task typically entails analyzing an image, text, or table of data (cross-sectional and time-series) to produce a number, label, additional text, additional images, or a mix of these. Simple examples include:

- Identifying a dog or cat in a picture.
- Guess the word that will come next in a sentence.
- Creating captions for images.
- Changing the style of a picture (like the Prisma app on iOS/Android).

A tensor can be a generic structure that can be used for storing, representing, and changing data.Tensors are the fundamental data structure used by all machine and deep learning algorithms. The term "TensorFlow" was given to Google's TensorFlow because tensors are essential to the discipline.

Tensor

A tensor is just a container for data, typically numerical data. It is, therefore, a container for numbers. Tensors are a generalization of matrices to any number of dimensions. You may already be familiar with matrices, which are 2D tensors (note that in the context of tensors, a dimension is often called an axis).

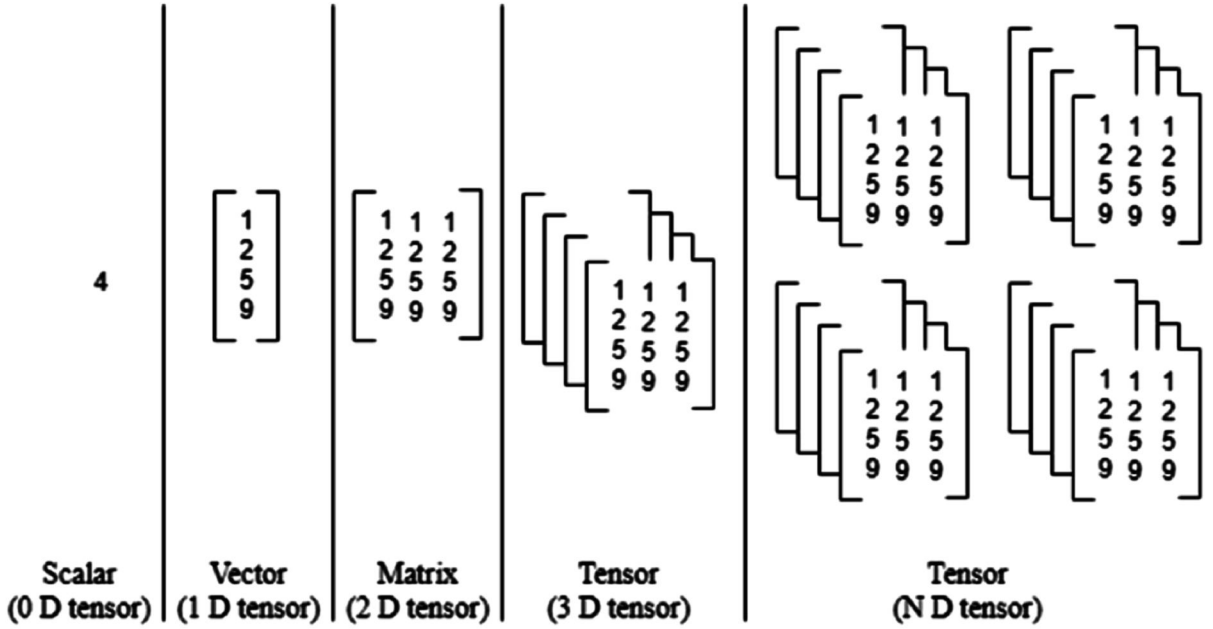


Fig.: Different types of Tensors

A scalar, vector, and matrix can all be represented as tensors in a more generalized fashion. An n-dimensional matrix serves as the definition of a tensor. A scalar is a zero-dimensional tensor (i.e., a single number), a vector is a one-dimensional tensor, a matrix is a two-dimensional tensor, a cube is a three-dimensional tensor, etc. The rank of a tensor is another name for a matrix's dimension.

The simplest method is to create a tensor in Python via lists. The experiment that follows will show a variety of tensor functions that are frequently employed in creating deep learning applications.

1. Scalars (0 D tensors)

The term "scalar" (also known as "scalar-tensor," "0-dimensional tensor," or "0D tensor") refers to a tensor that only holds a single number. A float32 or float64 number is referred to as a scalar-tensor (or scalar array) in Numpy. The "ndim" feature of a Numpy tensor can be used to indicate the number of axes; a scalar-tensor has no axes (ndim == 0). A tensor's rank is another name for the number of its axes. Here is a Scalar in Numpy:

```
>>> import numpy as np
>>> x = np.array(10)
>>> x
array(10)
>>>x.ndim
0
```

2. Vectors (1 D tensors)

A vector, often known as a 1D tensor, is a collection of numbers. It is claimed that a 1D tensor has just one axis. A NumPy vector can be written as:

```
>>> x = np.array([12, 3, 6, 14, 8])
>>> x
array([12, 3, 6, 14, 8])
>>>x.ndim
1
```

This vector is referred to as a 5D vector since it has five elements. A 5D tensor is not the same as a 5D vector. A 5D tensor will have five axes, whereas a 5D vector has just only one axis and five dimensions along it (and may have any number of dimensions along each axis). Dimensionality can refer to the number of axes in a tensor, such as a 5D tensor, or the number of entries along a particular axis, as in the case of our 5D vector. Although using the ambiguous notation 5D tensor is widespread, using a tensor of rank 5 (the rank of a tensor being the number of axes) is mathematically more accurate in the latter situation.

3. Matrices (2D tensors)

A matrix, or 2D tensor, is a collection of vectors. Two axes constitute a matrix (often referred to as rows and columns). A matrix can be visualized as a square box of numbers. The NumPy matrix can be written as:

```
>>> x = np.array([[5, 8, 2, 34, 0],
[6, 79, 30, 35, 1],
[9, 80, 49, 6, 2]])
>>>x.ndim
2
```

Rows and columns are used to describe the elements from the first and second axes. The first row of x in the example is [5, 8, 2, 34, 0], while the first column is [5, 6, 9].

4. 3D tensors or higher dimensional tensors

These matrices can be combined into a new array to create a 3D tensor, which can be seen as a cube of integers. Listed below is a Numpy 3D tensor:

```
>>> x = np.array([[[5, 8, 20, 34, 0],
[6, 7, 3, 5, 1],
[7, 80, 4, 36, 2]],
[[5, 7, 2, 34, 0],
[6, 79, 3, 35, 1],
[7, 8, 4, 36, 2]],
[[5, 78, 2, 3, 0],
[6, 19, 3, 3, 1],
[7, 8, 4, 36, 24]]])
>>>x.ndim
3
```

A 4D tensor can be produced by stacking 3D tensors in an array, and so on. In deep learning, you typically work with tensors that range from 0 to 4D, though if you're processing video data, you might go as high as 5D.

1.8 Key Attributes

Q8. Explain Key Attributes (Essential characteristics) and real world examples of data tensors.

Ans: **(Imp.)**

Key Attributes

Three essential characteristics are used to describe tensors:

1. **Number of axes (rank):** A matrix contains two axes, while a 3D tensor possesses three. In Python libraries like Numpy, this is additionally referred to as the tensor's ndim.
2. **Shape:** The number of dimensions the tensor contains across each axis is specified by a tuple of integers. For instance, the 3D tensor example has shape (3, 5) while the prior matrix example has shape (3, 3, 5). A scalar has an empty shape as (), but a vector has a shape with a single element, like (5,).
3. **Date type (sometimes abbreviated as "dtype" in Python libraries):** The format of the data which makes up the tensor; examples include float32, uint8, float64, and others. A 'char' tensor might appear in exceptional cases. Due to the string's changeable duration and the fact that tensors reside well before shared memory sections, string

tensors are not present in Numpy (or in the majority of other libraries).

Real-world examples of data tensors:

With some situations that are representative of what you'll see later, let's give data tensors additional context. Nearly all of the time, the data you work with will belong to one of the following groups:

1. **Vector data:** 2D tensors (samples, features).
2. **Sequence or time-series data:** 3D tensors of shape (samples, timesteps, characteristics)
3. **Images:** 4D tensors of shape (samples, height, width, channels) or (samples, channels, height, width)
4. **Video:** 5D shape tensors of shapes (samples, frames, height, width, channels) or (samples, frames, channels, height, width)

Vector Data

The most typical scenario is this. A batch of data in a dataset will be stored as a 2D tensor (i.e., an array of vectors), in which the first axis is the samples axis and the second axis is the features axis. Each individual data point in such a dataset is stored as a vector. Let's focus on two instances:

- A statistical dataset of consumers, where each individual's age, height, and gender are taken into account. Since each individual may be represented as a vector of three values, the full dataset of 100 individuals can be stored in a 2D tensor of the shape (100, 3).
- A collection of textual information in which each article is represented by the number of times each word occurs in it (out of a dictionary of 2000 common words). A full dataset of 50 articles can be kept in a tensor of shape (50, 2000) since each article can be represented as a vector of 20,00 values (one count per word in the dictionary).

Time series data or sequence data

It's imperative to store data in a 3D tensor with an explicit time axis whenever time (or the idea of sequence order) is important. A batch of data will be encoded as a 3D tensor because each instance can be represented as a series of vectors (a 2D tensor).

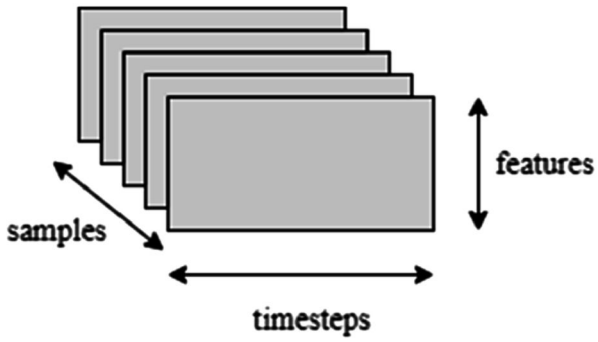


Fig.: 3D time-series data tensor

The time axis has always been the second axis (axis of index 1) by convention. Let's examine a couple of instances:

- **A stock price dataset:** We keep track of the stock's current market price as well as its peak and lowest prices from the previous hour. Since there are 390 minutes in a trading day, each minute is encoded as a 3D vector, a trading day may be represented as a 2D tensor of the form (390, 3), and 250 day's worth of data can be kept in a 3D tensor of shape (250, 390, 3). Each sample, in this case, corresponds to a day's worth of data.
- **Tweets dataset:** let's 300 characters be used to represent each tweet in a dataset of tweets, with a total of 125 different characters in the alphabet. Each character in this scenario can be represented as a binary vector of size 125 that is all zeros with the exception of a single item at the character-specific index. Then, a dataset of 10 million tweets can be kept in a tensor of shape by encoding each tweet as a 2D tensor of shape (300, 125). (10000000, 300, 125).

Image Data

Height, width, and colour depth are the three dimensions that most images have. By definition, image tensors are always 3D, with a one-dimensional colour channel for grayscale images. Even though grayscale images (like our MNIST digits) only have a single colour channel and may therefore be stored in 2D tensors. Thus, a tensor of shape (32, 64, 64, 1) might be used to save a batch of 32 grayscale photos of size 64 x 64, while a tensor of

shape (32, 64, 64, 1) could be used to store a batch of 32 colour images (32, 64, 64, 3).

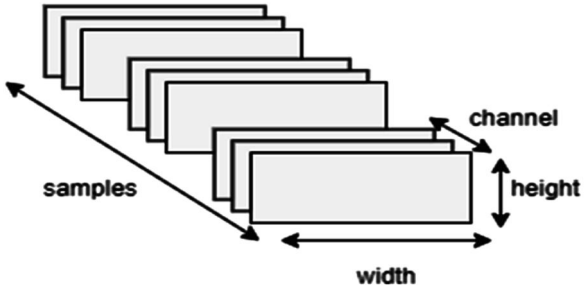


Fig.: 4D image data tensor

The channels-last format (used by TensorFlow) and the channels-first format are the two standards for the shapes of image tensors (used by Theano). The colour-depth axis is located at the end of the list of dimensions in the Google TensorFlow: (samples, height, width, colour-depth). The batch axis is placed first, followed by the samples, colour-depth, height, and width axes by Theano. The previous instances would be transformed into (32, 1, 64, 64) and (32, 3, 64, 64). Both file types are supported by the Keras framework.

Video data

Among the few real-world datasets for which you'll require 5D tensors is video data. A video could be viewed as a set of colored images called frames. A batch of various movies can be saved in a 5D tensor of shape (samples, frames, height, width, and colour-depth) since each frame can be kept in a 3D tensor (height, width, and colour-depth). A series of frames can also be saved in a 4D tensor (frames, height, width, and colour-depth).

For instance, 240 frames would be present in a 60-second, 144 x 256 YouTube video clip sampled at 4 frames per second. Four of these video clips would be saved in a tensor shape as a batch (4, 240, 144, 256, 3). There are 106,168,320 values in all. The tensor could store 405 MB if the dtype of the tensor was float32 since each value would be recorded in 32 bits. Heavy! Because they aren't saved in float32 and are often greatly compressed, videos you see in real life are significantly lighter (such as in the MPEG format).

1.9 Manipulating Tensors in NumPy

Q9. Explain in detail about Manipulating Tensors in NumPy with suitable examples.

Ans:

Tensor is a generalization of vectors and matrices and is easily understood as a multidimensional array.In the general case, an array of numbers arranged on a regular grid with a variable number of axes is known as a "tensor".

- Tensors are a type of data structure used in linear algebra, and like vectors and matrices, you can calculate arithmetic operations with tensors.
- A vector is a one-dimensional or first order tensor and a matrix is a two-dimensional or second order tensor.
- Tensor notation is much like matrix notation with a capital letter representing a tensor and lowercase letters with subscript integers representing scalar values within the tensor.
- Many of the operations that can be performed with scalars, vectors, and matrices can be reformulated to be performed with tensors.

Tensors in Python

Like vectors and matrices, tensors can be represented in Python using the N-dimensional array (ndarray).

- A tensor can be defined in-line to the constructor of array() as a list of lists.

The example below defines a 3x3x3 tensor as a NumPy ndarray. Three dimensions is easier to wrap your head around. Here, we first define rows, then a list of rows stacked as columns, then a list of columns stacked as levels in a cube.

```
# create tensor
from numpy import array
T = array([
[[1,2,3], [4,5,6], [7,8,9]],
[[11,12,13], [14,15,16], [17,18,19]],
[[21,22,23], [24,25,26], [27,28,29]],
])
print(T.shape)
print(T)
```

Running the example first prints the shape of the tensor, then the values of the tensor itself.

The Output shows that at least in three-dimensions, the tensor is printed as a series of matrices, one for each layer. For this 3D tensor, axis 0 specifies the level, axis 1 specifies the row, and axis 2 specifies the column.

OUTPUT:

```
(3, 3, 3)
[[[ 1  2  3]
[ 4  5  6]
[ 7  8  9]]
[[11 12 13]
[14 15 16]
[17 18 19]]
[[21 22 23]
[24 25 26]
[27 28 29]]]
```


Element-Wise Tensor Operations

As with matrices, we can perform element-wise arithmetic between tensors.

We will work through the four main arithmetic operations.

Tensor Addition

The element-wise addition of two tensors with the same dimensions results in a new tensor with the same dimensions where each scalar value is the element-wise addition of the scalars in the parent tensors.

a111, a121, a131, a112, a122, a132

A = (a211, a221, a231), (a112, a122, a132) b111, b121, b131, b112, b122, b132

B = (b211, b221, b231), (b112, b122, b132)

C = A + B

a111 + b111, a121 + b121, a131 + b131, a112 + b112, a122 + b122, a132 + b132

C = (a211 + b211, a221 + b221, a231 + b231), (a112 + b112, a122 + b122, a132 + b132)

In NumPy, we can add tensors directly by adding arrays.

```
# tensor addition
from numpy import array
A = array([
[[1,2,3], [4,5,6], [7,8,9]],
[[11,12,13], [14,15,16], [17,18,19]],
[[21,22,23], [24,25,26], [27,28,29]],
])
B = array([
[[1,2,3], [4,5,6], [7,8,9]],
[[11,12,13], [14,15,16], [17,18,19]],
[[21,22,23], [24,25,26], [27,28,29]],
])
C = A + B
print(C)
```

Running the example prints the addition of the two parent tensors.

```
[[[ 2  4  6]
  [ 8 10 12]
  [14 16 18]]
[[22 24 26]
  [28 30 32]
  [34 36 38]]
[[42 44 46]
  [48 50 52]
  [54 56 58]]]
```

Tensor Subtraction

The element-wise subtraction of one tensor from another tensor with the same dimensions results in a new tensor with the same dimensions where each scalar value is the element-wise subtraction of the scalars in the parent tensors.

$$A = \begin{matrix} a111, a121, a131 & a112, a122, a132 \\ (a211, a221, a231), & (a112, a122, a132) \end{matrix}$$

$$B = \begin{matrix} b111, b121, b131 & b112, b122, b132 \\ (b211, b221, b231), & (b112, b122, b132) \end{matrix}$$

C = A - B

a111 - b111, a121 - b121, a131 - b131 a112 - b112, a122 - b122, a132 - b132

C = (a211 - b211, a221 - b221, a231 - b231), (a112 - b112, a122 - b122, a132 - b132)

In NumPy, we can subtract tensors directly by subtracting arrays.

```
# tensor subtraction
from numpy import array
A = array([
    [[1,2,3], [4,5,6], [7,8,9]],
    [[11,12,13], [14,15,16], [17,18,19]],
    [[21,22,23], [24,25,26], [27,28,29]],
    ])
B = array([
    [[1,2,3], [4,5,6], [7,8,9]],
    [[11,12,13], [14,15,16], [17,18,19]],
    [[21,22,23], [24,25,26], [27,28,29]],
    ])
        C = A - B
    print(C)
```

Running the example prints the result of subtracting the first tensor from the second.

OUTPUT:

```
[[[0 0 0]
  [0 0 0]
  [0 0 0]]
[[0 0 0]
 [0 0 0]
 [0 0 0]]
[[0 0 0]
 [0 0 0]
 [0 0 0]]]
```

Tensor Hadamard Product

The element-wise multiplication of one tensor from another tensor with the same dimensions results in a new tensor with the same dimensions where each scalar value is the element-wise multiplication of the scalars in the parent tensors.

As with matrices, the operation is referred to as the Hadamard Product to differentiate it from tensor multiplication. Here, we will use the "o" operator to indicate the Hadamard product operation between tensors.

```
      a111, a121, a131      a112, a122, a132
A = (a211, a221, a231),   (a112, a122, a132)
      b111, b121, b131      b112, b122, b132
B = (b211, b221, b231),   (b112, b122, b132)
C = A o B
      a111 * b111, a121 * b121, a131 * b131      a112 * b112, a122 * b122, a132 * b132
C = (a211 * b211, a221 * b221, a231 * b231),   (a112 * b112, a122 * b122, a132 * b132)
```

In NumPy, we can multiply tensors directly by multiplying arrays.

```
# tensor Hadamard product
from numpy import array
A = array([
    [[1,2,3],     [4,5,6],     [7,8,9]],
    [[11,12,13], [14,15,16], [17,18,19]],
    [[21,22,23], [24,25,26], [27,28,29]],
    ])
B = array([
    [[1,2,3],     [4,5,6],     [7,8,9]],
    [[11,12,13], [14,15,16], [17,18,19]],
    [[21,22,23], [24,25,26], [27,28,29]],
    ])
C = A * B
print(C)
```

Running the example prints the result of multiplying the tensors.

```
[[[  1   4   9]
  [ 16  25  36]
  [ 49  64  81]]
[[121 144 169]
 [196 225 256]
 [289 324 361]]
[[441 484 529]
 [576 625 676]
 [729 784 841]]]
```

Tensor Division:

The element-wise division of one tensor from another tensor with the same dimensions results in a new tensor with the same dimensions where each scalar value is the element-wise division of the scalars in the parent tensors.

a111, a121, a131 a112, a122, a132

A = (a211, a221, a231), (a112, a122, a132)

b111, b121, b131 b112, b122, b132

B = (b211, b221, b231), (b112, b122, b132)

C = A / B

a111 / b111, a121 / b121, a131 / b131 a112 / b112, a122 / b122, a132 / b132

C = (a211 / b211, a221 / b221, a231 / b231), (a112 / b112, a122 / b122, a132 / b132)

In NumPy, we can divide tensors directly by dividing arrays.

```
# tensor division
from numpy import array
A = array([
    [[1,2,3], [4,5,6], [7,8,9]],
    [[11,12,13], [14,15,16], [17,18,19]],
    [[21,22,23], [24,25,26], [27,28,29]],
    ])
B = array([
    [[1,2,3], [4,5,6], [7,8,9]],
    [[11,12,13], [14,15,16], [17,18,19]],
    [[21,22,23], [24,25,26], [27,28,29]],
    ])
C = A / B
print(C)
```

Running the example prints the result of dividing the tensors.

```
[[[ 1.  1.  1.]
  [ 1.  1.  1.]
  [ 1.  1.  1.]]
 [[ 1.  1.  1.]
  [ 1.  1.  1.]
  [ 1.  1.  1.]]
 [[ 1.  1.  1.]
  [ 1.  1.  1.]
  [ 1.  1.  1.]]]
```

1.10 The Notion of Data Batches

Q10. What is the notion of data batches? Explain with an example.

Ans:

The batch size is a hyperparameter that defines the number of samples to work through before updating the internal model parameters.

Think of a batch as a for-loop iterating over one or more samples and making predictions. At the end of the batch, the predictions are compared to the expected output variables and an error is calculated. From this error, the update algorithm is used to improve the model, e.g. move down along the error gradient.

A training dataset can be divided into one or more batches.

When all training samples are used to create one batch, the learning algorithm is called batch gradient descent. When the batch is the size of one sample, the learning algorithm is called stochastic gradient descent. When the batch size is more than one sample and less than the size of the training dataset, the learning algorithm is called mini-batch gradient descent.

- **Batch Gradient Descent**. Batch Size = Size of Training Set
- **Stochastic Gradient Descent**. Batch Size = 1
- **Mini-Batch Gradient Descent**. 1 < Batch Size < Size of Training Set

In the case of mini-batch gradient descent, popular batch sizes include 32, 64, and 128 samples.

The notion of data batches

In general, the first axis (axis 0, because indexing starts at 0) in all data tensors you'll come across in deep learning will be the samples axis (sometimes called the samples dimension). In the MNIST example, samples are images of digits. In addition, deep-learning models don't process an entire dataset at once; rather, they break the data into small batches. Concretely, here's one batch of our MNIST digits, with batch size of 128:

```
batch = X_train_images[:128]
```

And here's the next batch:

```
batch = X_train_images[128:256]
```

And the n th batch:

```
batch = X_train_images[128 * n:128 * (n + 1)]
```

When considering such a batch tensor, the first axis (axis 0) is called the batch axis or batch dimension. This is a term we will frequently encounter when using Keras and other deep-learning libraries.

1.11 Real-World Examples of Tensors

Q11. What are the Real-World Examples of Tensors? Give with examples.

Ans: **(Imp.)**

Real-World Examples of (0D, 1D, 2D, 3D, 4D and 5D) Tensors:

- Tensors are the basic data structure in machine learning and deep learning models.
- Tensors are nothing but multi-dimensional NumPy arrays.

A tensor can be considered as a container for numerical data (numbers). In neural networks, data is represented by using tensors. The input layer of a neural network holds text, speech, audio, images, videos or any other kind of data as tensors that takes numbers. We perform tensor operations (tensor addition, multiplication, reshaping, etc.) throughout the network with these tensors of numerical data.

Dimensions, axes and rank of a tensor:

Dimensions

A tensor can contain an arbitrary number of dimensions. Therefore, a tensor can be 0D (no dimension!), 1D, 2D, 3D, 4D, 5D and so on. Specific names are given to tensors depending on the number of dimensions they have.

Therefore, a 0D tensor is specifically known as a Scalar. A 1D tensor is specifically known as a Vector. A 2D tensor is specifically known as a Matrix. 3D and higher-dimensional arrays are just tensors!In some contexts, only the 3D and higher-dimensional arrays are considered as tensors. But in deep learning, "tensor" is a general term that is used to refer to an array of any dimension.

Axes:The term "axis" (plural: axes) is another way to refer to a dimension of a tensor. For example, a 1D tensor (i.e. a vector) has a single axis. Likewise, a 2D tensor (i.e. a matrix) has two axes and so on.Rank:The number of axes (dimensions) of a tensor is called its rank. For example, a 0D tensor (i.e. scalar) is a rank-0 tensor. Likewise, a 1D tensor (i.e. a vector) is a rank-1 tensor and so on.

Top takeaway: The dimensionality or rank denotes the number of axes in a tensor.

Real-world examples of tensors

We can start with rank-0 tensors and end up with rank-5 tensors. The real-world examples for each type of tensor along with what each axis means in the case of rank-2 and higher tensors.

Rank-0 tensors (scalars or 0D tensors)

A rank-0 tensor contains only one number. It is just a scalar!

Example: A pixel value in a grayscale image.

Consider a pixel value of 10 that represents a color close to black.

Grayscale pixel value: An example of a rank-0 tensor.

Rank-1 tensors (vectors or 1D tensors):

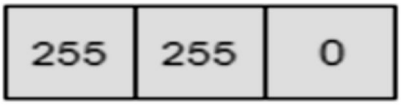
A rank-1 tensor contains a set of numbers in a single axis.

Example 1:

A pixel value in an RGB image.

Let's consider an RGB pixel value of [255, 255, 0] that represents the yellow color.

consider an RGB pixel value of [255, 255, 0] that represents the yellow color.

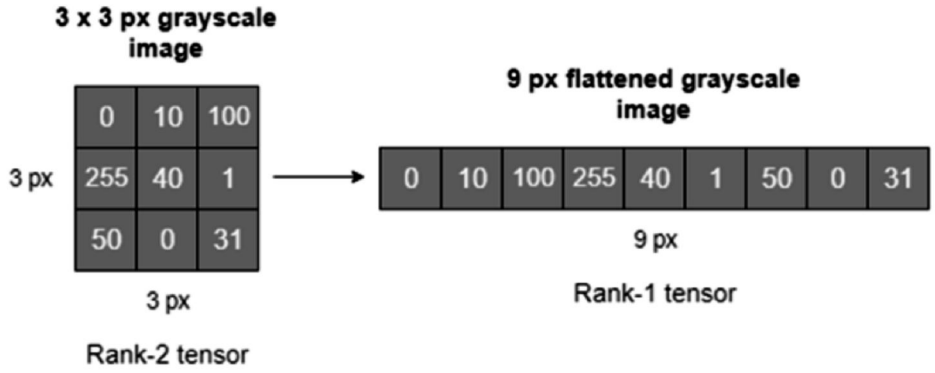


RGB pixel value: An example of a rank-1 tensor

Example 2:

A flattened grayscale image.

A grayscale image can be flattened into a rank-1 tensor as in the following diagram.



Flattened grayscale image: An example of a rank-1 tensor

Rank-2 tensors (matrices or 2D tensors):

A rank-2 tensor is an array of vectors. It has two axes.

Example 1:

A tabular dataset with rows and columns.

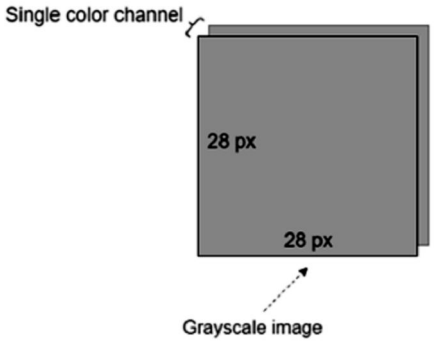
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

Tabular dataset: An example of a rank-2 tensor

In this case, the two axes denote (samples, features) which is equivalent to rows and columns. For example, (5, 4) means that the dataset has 5 observations (rows or samples) and 4 variables (columns or features).

Example 2:

A single non-flattened grayscale image.

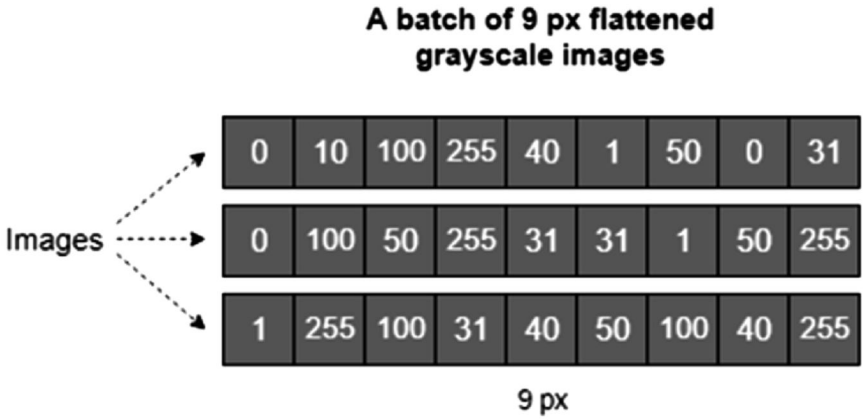


Non-flattened grayscale image: An example of a rank-2 tensor

In this case, the two axes denote (height, width) which is equivalent to the height and width of an image in pixels. For example, (28, 28) means that the array holds a single grayscale image of size 28 x 28 in pixels.

Example 3:

A batch of flattened grayscale images.



A batch of flattened grayscale images: An example of a rank-2 tensor

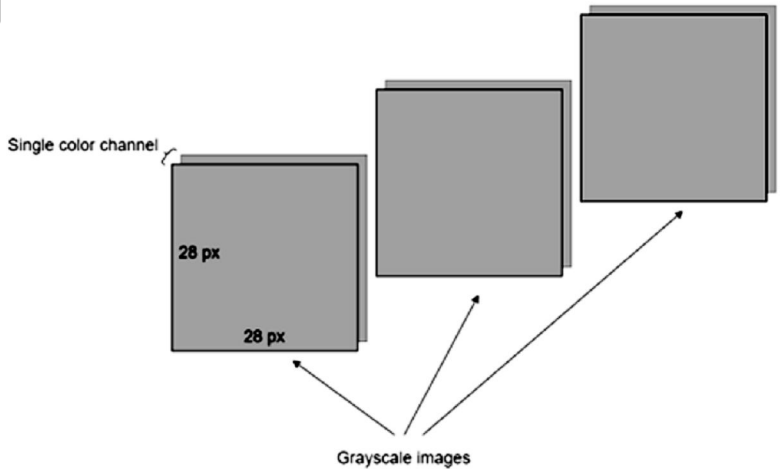
In this case, the two axes denote (samples, features) which is equivalent to the number of flattened grayscale images and pixel values of each image. For example, (3, 9) means that the array contains 3 flattened grayscale images each having 9 pixels.

Rank-3 tensors (3D tensors)

A rank-3 tensor is an array of several matrices. It has 3 axes.

Example 1:

A batch of grayscale images.

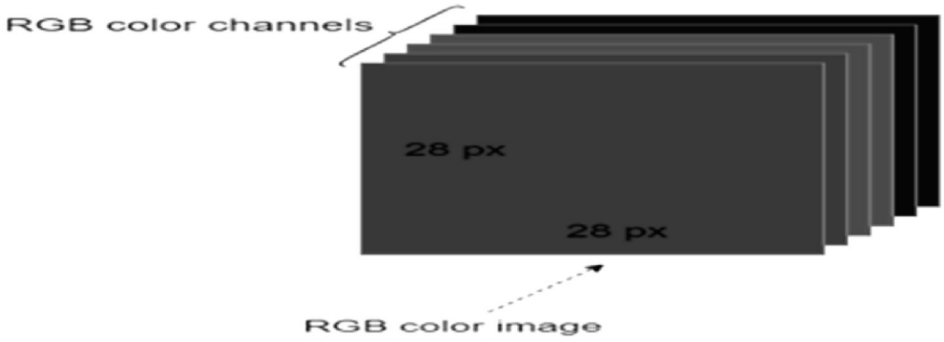


A batch of grayscale images: An example of a rank-3 tensor

In this case, the three axes denote (samples, height, width). For example, (1000, 28, 28) means that the array contains 1000 grayscale images of size 28 x 28 in pixels.

Example 2:

A single RGB image.



RGB image: An example of a rank-3 tensor (Image by author)

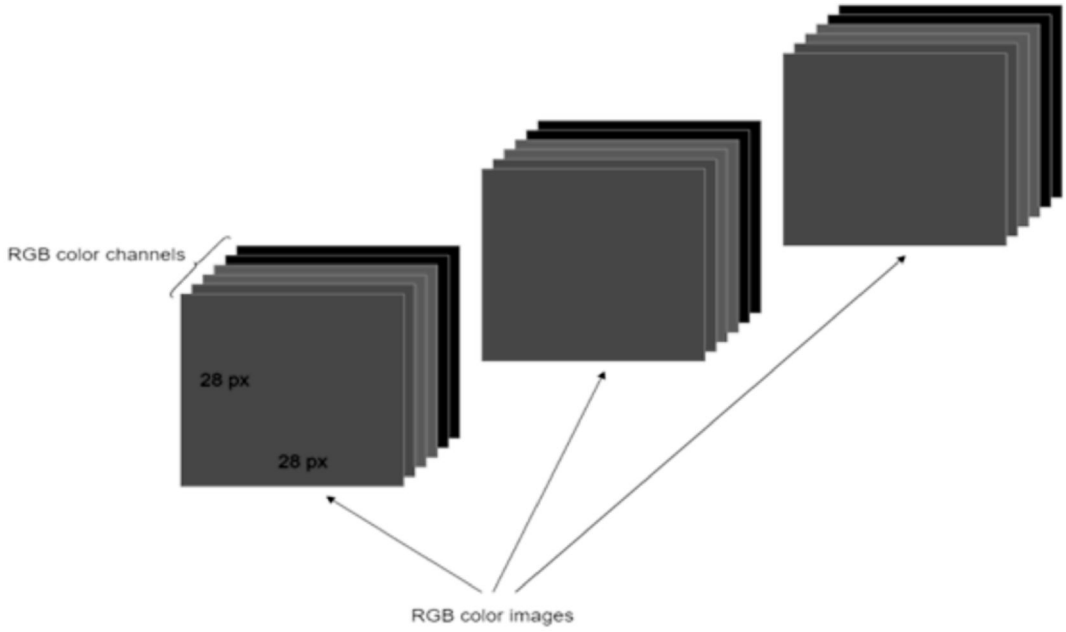
In this case, the three axes denote (height, width, color_channels). For example, (28, 28, 3) means that the array contains a single RGB image of size 28 x 28. The color_channels axis always takes the value 3 in the case of an RGB (Red, Green, Blue) image.

Rank-4 tensors (4D tensors):

A rank-4 tensor is created by arranging several 3D tensors into a new array. It has 4 axes.

Example 1:

A batch of RGB images.



A batch of RGB images: An example of a rank-4 tensor

In this case, the four axes denote (samples, height, width, color_channels). For example, (1000, 28, 28, 3) means that the array holds 1000 RGB images of size 28 x 28 in pixels. The color_channels axis always takes the value 3 in the case of an RGB (Red, Green, Blue) image.

Example 2: A single video.

In this case, the four axes denote (frames, height, width, color_channels). For example, (240, 720, 1280, 3) means that the array holds a 240-frame HD (720 x 1280) video. Each frame in a video is a color image. Therefore, a frame is denoted by (height, width, color_channels). A video can be considered as a sequence of frames. A 240-frame video is sampled at 4 (240/60) frames per second. In an HD video, each frame is 720 x 1280 size in pixels.

Rank-5 tensors (5D tensors):

A rank-5 tensor is created by arranging several 4D tensors into a new array. It has 5 axes.

Example:

A batch of videos.

In this case, the five axes denote (samples, frames, height, width, color_channels). For example, (5, 240, 720, 1280, 3) means that the array holds a batch of five 240-frame HD (720 x 1280) videos.

The samples axis (samples dimension)

In the above notations such as (samples, features) , (samples, height, width) , (samples, height, width, color_channels) , (samples, frames, heights, width, color_channels) , the first axis (i.e. the axis 0 in index notation) is always the samples axis. In a tabular dataset, the term "samples" means observations. In image data, the term "samples" means the number of images. In video data, the term "samples" means the number of videos.

The batch axis (batch dimension)

When we consider a small batch (part) of the dataset instead of considering the whole dataset, the sample axis is referred to as the batch axis. This is just a technical term often found in the context of deep learning.

1.12 VECTOR DATA

Q12. Define vector? Explain about Vector Data with an example.

Ans:

A vector is a mathematical object that encodes a length and direction. Conceptually they can be thought of as representing a position or even a change in some mathematical framework or space. More formally they are elements of a vector space: a collection of objects that is closed under an addition rule and a rule for multiplication by scalars.

A vector is often represented as a 1-dimensional array of numbers, referred to as components and is displayed either in column form or row form. Represented geometrically, vectors typically represent coordinates within a n-dimensional space, where n is the number of dimensions. A simplistic representation of a vector might be a arrow in a vector space, with an origin, direction, and magnitude (length).

Working of Vectors

As basic units for computational arithmetic, vectors can be transformed utilizing basic mathematics. For example, vectors can be added, subtracted and multiplied by. For instance, vector addition can be denoted as:

$$a + b = (a1 + b1, a2 + b2, a3 + b3)$$

Furthermore, several rules vector multiplication can be defined. One such is the pointwise product and is denoted in a similar way:

a * b = (a1 * b1, a2 * b2, a3 * b3)

Additionally, a vector dot product can calculate the sum of the multiplied elements of two vectors of the same length to give a scalar:

a . b = (a1 * b1 + a2 * b2 + a3 * b3)

Other vector products such as the cross product or outer product can be defined in other ways.

Example of Vector data:

This is the most common case. In such a dataset, each single data point can be encoded as a vector, and thus a batch of data will be encoded as a 2D tensor (that is, an array of vectors), where the first axis is the samples axis and the second axis is the features axis.

Let's take a look at two examples:

An actuarial dataset of people, where we consider each person's age, ZIP code, and income. Each person can be characterized as a vector of 3 values, and thus an entire dataset of 100,000 people can be stored in a 2D tensor of shape (100000, 3).

A dataset of text documents, where we represent each document by the counts of how many times each word appears in it (out of a dictionary of 20,000 common words). Each document can be encoded as a vector of 20,000 values (one count per word in the dictionary), and thus an entire dataset of 500 documents can be stored in a tensor of shape (500, 20000).

1.13 TIMESERIES DATA OR SEQUENCE DATA

Q13. What is Timeseries data and give an example?

(OR)

What is sequence data and given an example.

Ans: **(Imp.)**

Time series is a machine learning technique that forecasts target value based solely on a known history of target values. It is a specialized form of regression, known in the literature as auto-regressive modeling.

The input to time series analysis is a sequence of target values. A case id column specifies the order of the sequence. The case id can be of type NUMBER or a date type (date, datetime, timestamp with time zone, or timestamp with local time zone). Regardless of case id type, the user can request that the model include trend, seasonal effects or both in its forecast computation. When the case id is a date type, the user must specify a time interval (for example, month) over which the target values are to be aggregated, along with an aggregation procedure (for example, sum). Aggregation is performed by the algorithm prior to constructing the model.

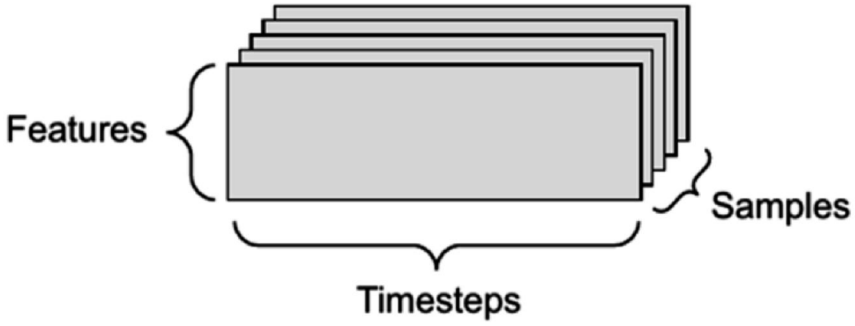
The time series model provide estimates of the target value for each step of a time window that can include up to 30 steps beyond the historical data. Like other regression models, time series models compute various statistics that measure the goodness of fit to historical data.

Forecasting is a critical component of business and governmental decision making. It has applications at the strategic, tactical and operation level. The following are the applications of forecasting:

- Projecting return on investment, including growth and the strategic effect of innovations
- Addressing tactical issues such as projecting costs, inventory requirements and customer satisfaction
- Setting operational targets and predicting quality and conformance with standards

Example of Timeseries data or sequence data:

Whenever time matters in your data (or the notion of sequence order), it makes sense to store it in a 3D tensor with an explicit time axis. Each sample can be encoded as a sequence of vectors (a 2D tensor), and thus a batch of data will be encoded as a 3D tensor



The time axis is always the second axis (axis of index 1), by convention.

Let's look at few examples:

- **A dataset of stock prices**. Every minute, we store the current price of the stock, the highest price in the past minute, and the lowest price in the past minute. Thus every minute is encoded as a 3D vector, an entire day of tradingencoded as a 2D tensor of shape (390, 3) (there are 390 minutes in a trading day), and 250 days' worth of data can be stored in a 3D tensor of shape (250, 390, 3). Here, each sample would be one day's worth of data.
- **A dataset of tweets**, where we encode each tweet as a sequence of 280 characters out of an alphabet of 128 unique characters. In this setting, each character can be encoded as a binary vector of size 128 (an all-zeros vector except for a 1 entry at the index corresponding to the character). Then each tweet can be encoded as a 2D tensor of shape (280, 128), and a dataset of 1 million tweets can be stored in a tensor of shape (1000000, 280, 128).

1.14 IMAGE DATA

Q14. What is Image Data? Explain with suitable example.

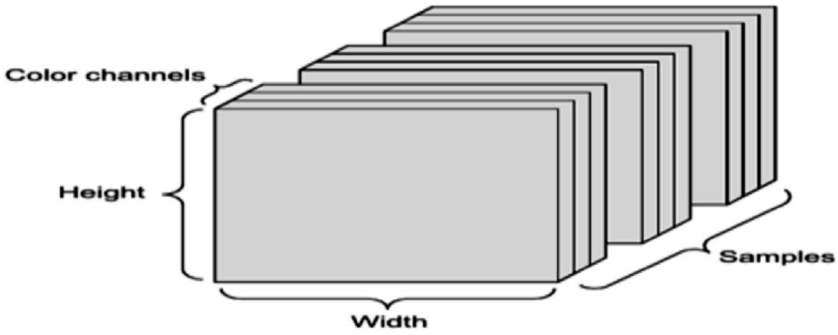
Ans: **(Imp.)**

Image classification is one of the most important applications of deep learning and Artificial Intelligence. Image classification refers to assigning labels to images based on certain characteristics or features present in them.

Animage dataset is a collection of data curated for a machine learning project. An image dataset includes digital images curated for testing, training, and evaluating the performance of machine learning and artificial intelligence (AI) algorithms, commonly computer vision algorithms.

Example of Image data:

Images typically have three dimensions: height, width, and color depth. Although grayscale images (like our MNIST digits) have only a single color channel and could thus be stored in 2D tensors, by convention image tensors are always 3D, with a one-dimensional color channel for grayscale images. A batch of 128 grayscale images of size 256 × 256 could thus be stored in a tensor of shape (128, 256, 256, 1), and batch of 128 color images could be stored in a tensor of shape (128, 256, 256, 3)



The above is a A 4D image data tensor

There are two conventions for shapes of images tensors: the channels-last convention (used by TensorFlow) and the channels-first convention (used by Theano). TensorFlow, places the color-depth axis at the end: (samples, height, width, color_depth). Meanwhile, Theano places the color depth axis right after the batch axis: (samples, color_depth, height, width). With the Theano convention, the previous examples would become (128, 1, 256, 256) and (128, 3, 256, 256). The Keras framework provides support for both formats.

1.15 VIDEO DATA

Q15. Explain about video data with an example.

Ans:

Video data contains a rich amount of information, and has a more complex and large structure than image data. Being able to classify videos in a memory-efficient way using deep learning can help us better understand the contents within the data. On tensorflow.org, we have published a series of tutorials on how to load, preprocess, and classify video data.

1. Load video data

- Read sequences of frames out of the video files.
- Visualize the video data.
- Wrap the frame-generator tf.data.Dataset.

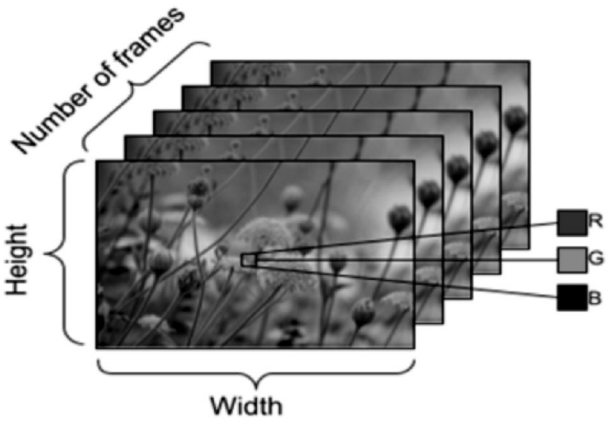
2. Video classification with a 3D convolutional neural network

- Build an input pipeline
- Build a 3D convolutional neural network model with residual connections using Keras functional API
- Train the model
- Evaluate and test the model

3. MoViNet for streaming action recognition

4. Transfer learning for video classification with MoViNet

Example of Video data



Video data is one of the few types of real-world data for which you'll need 5D tensors.

A video can be understood as a sequence of frames, each frame being a color image. Because each frame can be stored in a 3D tensor (height, width, color_depth), a sequence of frames can be stored in a 4D tensor (frames, height, width, color_depth), and thus a batch of different videos can be stored in a 5D tensor of shape (samples, frames, height, width, color_depth).

For instance, a 60-second, 144 × 256 YouTube video clip sampled at 4 frames per second would have 240 frames. A batch of four such video clips would be stored in a tensor of shape (4, 240, 144, 256, 3). That's a total of 106,168,320 values! If the dtype of the tensor was float32, then each value would be stored in 32 bits, so the tensor would represent 405 MB. Heavy! Videos we encounter in real life are much lighter, because they aren't stored in float32, and they're typically compressed by a large factor (such as in the MPEG format).

Short Question and Answers

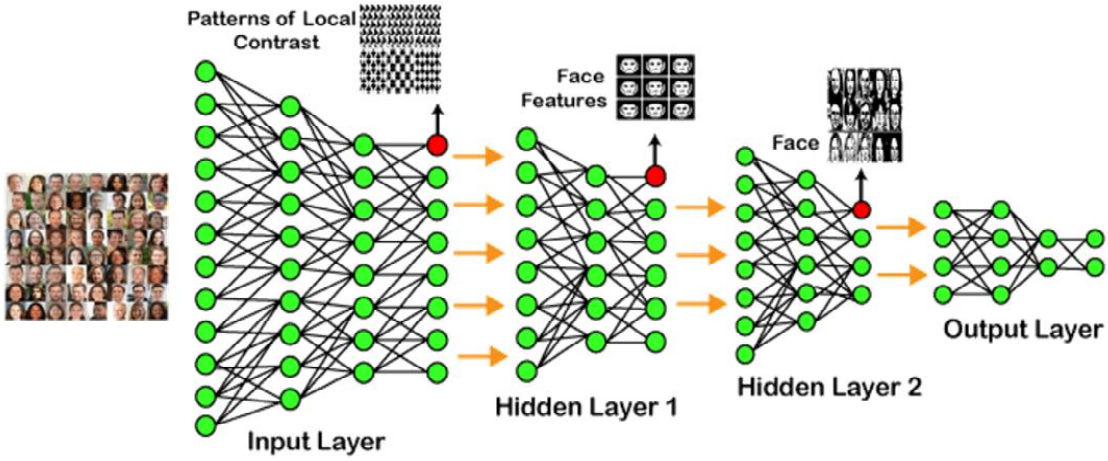
1. Write note on Deep Learning?

Ans :

Deep Learning: Deep learning is a branch of machine learning which is completely based on artificial neural networks, as neural network is going to mimic the human brain so deep learning is also a kind of mimic of human brain. In deep learning, we don't need to explicitly program everything.

"Deep learning is a collection of statistical techniques of machine learning for learning feature hierarchies that are actually based on artificial neural networks."

Example of Deep Learning:



In the example given above, we provide the raw data of images to the first layer of the input layer. After then, these input layer will determine the patterns of local contrast that means it will differentiate on the basis of colors, luminosity, etc. Then the 1st hidden layer will determine the face feature, i.e., it will fixate on eyes, nose, and lips, etc. And then, it will fixate those face features on the correct face template. So, in the 2nd hidden layer, it will actually determine the correct face here as it can be seen in the above image, after which it will be sent to the output layer. Likewise, more hidden layers can be added to solve more complex problems, for example, if you want to find out a particular kind of face having large or light complexions. So, as and when the hidden layers increase, we are able to solve complex problems.

2. Explain the Hardware (System Requirements) for Deep Learning?

Ans :

System Requirements(Software & Hardware)

ENVI Deep Learning 2.0 uses TensorFlow version 2.9 and CUDA version 11.2.2, both of which are included in the installation. System requirements are as follows:

Base software: ENVI 5.6.3 and the ENVI Deep Learning 2.0 module

Operating systems

- Windows 10 and 11 (Intel/AMD 64-bit)
- Linux (Intel/AMD 64-bit, kernel 3.10.0 or higher, glibc 2.17 or higher)

Hardware

- NVIDIA graphics card with CUDA Compute Capability version 3.5 to 8.6. See the list of CUDA-enabled GPU cards. A minimum of 8 GB of GPU memory is recommended for optimal performance, particularly when training deep learning models.
- NVIDIA GPU driver version: Windows 461.33 or higher, Linux 460.32.03 or higher.
- A CPU with the Advanced Vector Extensions (AVX) instruction set. In general, any CPU after 2011 will contain this instruction set.
- Intel CPUs are recommended, though not required. They have an optimized Intel Machine Learning library that offers performance gains for certain Machine Learning algorithms.

To determine if your system meets the requirements for ENVI Deep Learning, start the Deep Learning Guide Map in the ENVI Toolbox. From the Deep Learning Guide Map menu bar, select Tools > Test Installation and Configuration.

3. Differences between ML and DL?

Ans :

Key comparisons between Machine Learning and Deep Learning

The key differences between these two terms based on different parameters:

Parameter	Machine Learning	Deep Learning
Data Dependency	Although machine learning depends on the huge amount of data, it can work with a smaller amount of data.	Deep Learning algorithms highly depend on a large amount of data, so we need to feed a large amount of data for good performance.
Execution time	Machine learning algorithm takes less time to train the model than deep learning, but it takes a long-time duration to test the model.	Deep Learning takes a long execution time to train the model, but less time to test the model.
Hardware Dependencies	Since machine learning models do not need much amount of data, so they can work on low-end machines.	The deep learning model needs a huge amount of data to work efficiently, so they need GPU's and hence the high-end machine.
Feature Engineering	Machine learning models need a step of feature extraction by the expert, and then it proceeds further.	Deep learning is the enhanced version of machine learning, so it does not need to develop the feature extractor for each problem; instead, it tries to learn high-level features from the data on its own.
Problem-solving approach	To solve a given problem, the traditional ML model breaks the problem in sub-parts, and after solving each part, produces the final result.	The problem-solving approach of a deep learning model is different from the traditional ML model, as it takes input for a given problem, and produce the end result. Hence it follows the end-to-end approach.

Interpretation of result	The interpretation of the result for a given problem is easy. As when we work with machine learning, we can interpret the result easily, it means why this result occur, what was the process.	The interpretation of the result for a given problem is very difficult. As when we work with the deep learning model, we may get a better result for a given problem than the machine learning model, but we cannot find why this particular outcome occurred, and the reasoning.
Type of data	Machine learning models mostly require data in a structured form.	Deep Learning models can work with structured and unstructured data both as they rely on the layers of the Artificial neural network.
Suitable for	Machine learning models are suitable for solving simple or bit-complex problems.	Deep learning models are suitable for solving complex problems.

4. Applications of Deep Learning?

Ans :

Applications of Deep Learning Algorithms

Here are some ways where deep learning is being used in diverse industries.

(i) Computer Vision: Computer Vision is mainly depending on image processing methods. Before deep learning, the best computer vision algorithm depending on conventional I machine learning and image processing obtained a 25% error rate. But, when a deep neural network used for image processing, the error rate dropped to 16 per cent, and now with advancement in deep learning algorithms, the error rate dropped to less than 4 %.

(ii) Text Analysis & Understanding: Text analysis consists of the classification of documents, sentiment analysis, automatic translation, etc. Recurrent neural networks are the most useful deep learning algorithm here, because of the sequential type of textual data.

(iii) Speech Recognition: Speech Recognition enables to process of human speech into text by computers. Traditionally, Speech recognition mainly relies upon a hefty feature extraction process but deep learning is directly working on raw data and training done on a large dataset of audio recording.

(iv) Pattern Recognition: Pattern recognition is the automated identification of patterns and regularities in data. The data type can vary anything from text, images to sounds or audio.

PayPal is using deep learning via H2O, a predictive analytics platform, to help prevent payment transactions and fraudulent purchases and

(v) Autonomous vehicles: The autonomous vehicle accomplished to collect data on its surrounding from various sensors, explain it, and based on explanation choose what actions need to be taken. Deep learning enables us to learn how to perform the work as effectively as humans.

Thanks for reading! In my next article, I will be explaining various activation functions with applications.

5. Write short note on Deep Learning Algorithms?

Ans :

Types of Algorithms used in Deep Learning

Here is the list of top 10 most popular deep learning algorithms:

(i) Convolutional Neural Networks (CNNs)

CNN's are widely used to identify satellite images, process medical images, forecast time series, recognizing characters like ZIP codes and digits. And detect anomalies.

(ii) Long Short-Term Memory Networks (LSTMs)

LSTMs are typically used for speech recognition, music composition, and pharmaceutical development.

(iii) Recurrent Neural Networks (RNNs)

RNNs are commonly used for image captioning, time-series analysis, natural-language processing, handwriting recognition, and machine translation.

(iv) Generative Adversarial Networks (GANs)

GANs help generate realistic images and cartoon characters, create photographs of human faces, and render 3D objects.

(v) Radial Basis Function Networks (RBFNs)

RBFNs mostly used for classification, regression, and time-series prediction.

(vi) Multilayer Perceptron's (MLPs)

MLPs used to build speech-recognition, image-recognition, and machine-translation software.

(vii) Self-Organizing Maps (SOMs)

SOMs which enable data visualization to reduce the dimensions of data through self-organizing artificial neural networks.

(viii) Deep Belief Networks (DBNs)

(DBNs) are used for image-recognition, video-recognition, and motion-capture data.

(ix) Restricted Boltzmann Machines (RBMs)

RBMs used for dimensionality reduction, classification, regression, collaborative filtering, feature learning, and topic modeling.

(x) Autoencoders

Autoencoders are used for purposes such as pharmaceutical discovery, popularity prediction, and image processing.

6. Explain about Simple neural network architecture.

Ans :

A basic neural network has interconnected artificial neurons in three layers:

Input Layer

Information from the outside world enters the artificial neural network from the input layer. Input nodes process the data, analyze or categorize it, and pass it on to the next layer.

Hidden Layer

Hidden layers take their input from the input layer or other hidden layers. Artificial neural networks can have a large number of hidden layers. Each hidden layer analyzes the output from the previous layer, processes it further, and passes it on to the next layer.

Output Layer

The output layer gives the final result of all the data processing by the artificial neural network. It can have single or multiple nodes. For instance, if we have a binary (yes/no) classification problem, the output layer will have one output node, which will give the result as 1 or 0. However, if we have a multi-class classification problem, the output layer might consist of more than one output node.

7. Discuss about Data representations for neural networks?

Ans :

Tensor

A tensor is just a container for data, typically numerical data. It is, therefore, a container for numbers.

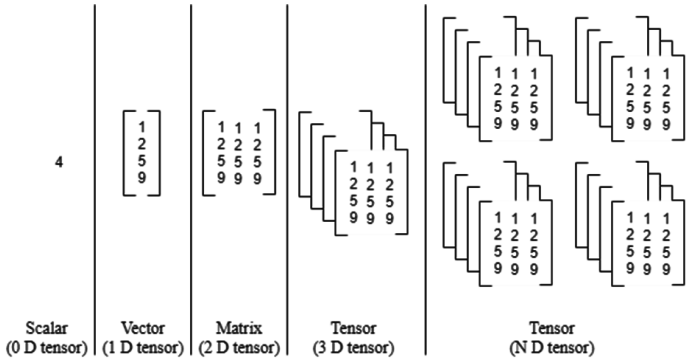
(i) Scalars (0D tensors): The term "scalar" (also known as "scalar-tensor," "0-dimensional tensor," or "0D tensor") refers to a tensor that only holds a single number. A float32 or float64 number is referred to as a scalar-

tensor (or scalar array) in Numpy. The "*ndim*" feature of a Numpy tensor can be used to indicate the number of axes; a scalar-tensor has no axes (ndim = = 0).

(ii) Vectors (1D tensors): A vector, often known as a 1D tensor, is a collection of numbers. It is claimed that a 1D tensor has just one axis.

(iii) Matrices (2D tensors): A matrix, or 2D tensor, is a collection of vectors. Two axes constitute a matrix (often referred to as rows and columns). A matrix can be visualized as a square box of numbers.

(iv) 3D tensors or higher dimensional tensors: These matrices can be combined into a new array to create a 3D tensor, which can be seen as a cube of integers.



8. What is the use of Deep learning in today's age, and how is it adding data scientists?

Ans :

Deep learning has brought significant changes or revolution in the field of machine learning and data science. The concept of a complex neural network (CNN) is the main center of attention for data scientists. It is widely taken because of its advantages in performing next-level machine learning operations. The advantages of deep learning also include the process of clarifying and simplifying issues based on an algorithm due to its utmost flexible and adaptable nature. It is one of the rare procedures which allow the movement of data in independent pathways. Most of the data scientists are viewing this particular medium as an advanced additive and extended way to the existing process of machine learning and utilizing the same for solving complex day to day issues.

9. What do you understand by Boltzmann Machine?

Ans :

A Boltzmann machine (also known as stochastic Hopfield network with hidden units) is a type of recurrent neural network. In a Boltzmann machine, nodes make binary decisions with some bias. Boltzmann machines can be strung together to create more sophisticated systems such as deep belief networks. Boltzmann Machines can be used to optimize the solution to a problem.

Some important points about Boltzmann Machine :

- It uses a recurrent structure.
- It consists of stochastic neurons, which include one of the two possible states, either 1 or 0.
- The neurons present in this are either in an adaptive state (free state) or clamped state (frozen state).

If we apply simulated annealing or discrete Hopfield network, then it would become a Boltzmann Machine.

Choose the Correct Answer

1. Which of the following is a subset of machine learning? [c]
   (a) NumPy (b) SciPy
   (c) Deep Learning (d) All
2. How many layers Deep learning algorithms are constructed? [b]
   (a) 2 (b) 3
   (c) 4 (d) 5
3. The first layer is called the? [a]
   (a) inner layer (b) outer layer
   (c) hidden layer (d) None of the above
4. RNNs stands for? [d]
   (a) Receives neural networks (b) Report neural networks
   (c) Recording neural networks (d) Recurrent neural networks
5. Which of the following is/are Common uses of RNNs? [d]
   (a) Businesses Help securities traders to generate analytic reports
   (b) Detect fraudulent credit-card transaction
   (c) Provide a caption for images
   (d) All of the above
6. Which of the following is well suited for perceptual tasks? [c]
   (a) Feed-forward neural networks (b) Recurrent neural networks
   (c) Convolutional neural networks (d) Reinforcement Learning
7. CNN is mostly used when there is an? [b]
   (a) structured data (b) unstructured data
   (c) Both A and B (d) None of them
8. Which neural network has only one hidden layer between the input and output? [a]
   (a) Shallow neural network (b) Deep neural network
   (c) Feed-forward neural networks (d) Recurrent neural networks
9. Which of the following is/are Limitations of deep learning? [c]
   (a) Data labeling (b) Obtain huge training datasets
   (c) Both A and B (d) None of them
10. Deep learning algorithms are _____ more accurate than machine learning algorithm in image classification. [d]
   (a) 33% (b) 37%
   (c) 40% (d) 41%

Fill in the blanks

1 The amount of output of one unit received by another unit depends on _____.

2. The process of adjusting the weight is known as _____.

3. Learning is a _____.

4. _____ is a branch of machine learning which is completely based on artificial neural networks.

5. RLU stands for _____

6. _____ are generative deep learning algorithms that create new data instances that resemble the training data.

7. BMU stands for _____

8. _____ networks can comprehend unstructured data and make general observations without explicit training.

9. In Neural Networks learning processes, learning with a teacher is also referred to as _____ learning

10. The _____ is used to control the amount of weight adjustment at each step of training.

ANSWERS

1. Weight
2. Learning
3. Slow process
4. Deep learning
5. Rectified Linear Unit (ReLU)
6. GANs
7. Best Matching Unit
8. Neural
9. Supervised
10. Learning rate

One Mark Answers

1. Define Deep Learning?

Ans :

Deep Learning is a part of Machine Learning, which involves mimicking the human brain in terms of structures called neurons, thereby, forming neural networks.

2. What is a perceptron?

Ans :

A perceptron is similar to the actual neuron in the human brain. It receives inputs from various entities and applies functions to these inputs, which transform them to be the output.

3. What are some of the most used applications of Deep Learning?

Ans :

- Sentiment Analysis
- Computer Vision
- Automatic Text Generation
- Object Detection
- Natural Language Processing
- Image Recognition

4. What is the meaning of overfitting?

Ans :

Overfitting is a very common issue when working with Deep Learning. It is a scenario where the Deep Learning algorithm vigorously hunts through the data to obtain some valid information.

5. What are activation functions?

Ans :

Activation functions are entities in Deep Learning that are used to translate inputs into a usable output parameter.

6. Why is Fourier transform used in Deep Learning?

Ans :

Fourier transform is an effective package used for analyzing and managing large amounts of data present in a database.

7. What is the use of the loss function?

Ans :

The loss function is used as a measure of accuracy to see if a neural network has learned accurately from the training data or not. This is done by comparing the training dataset to the testing dataset.

8. What are some of the Deep Learning frameworks or tools that you have used?

Ans :

- Keras
- PyTorch
- Caffe2
- CNTK
- MXNet
- Theano
- TensorFlow

9. What are autoencoders?

Ans :

Autoencoders are artificial neural networks that learn without any supervision. Here, these networks have the ability to automatically learn by mapping the inputs to the corresponding outputs.

10. Define Encoder and Decoder?

Ans :

- Encoder: Used to fit the input into an internal computation state.
- Decoder: Used to convert the computational state back into the output.

UNIT II

Tensor operations

Element-wise operations, Broadcasting, Tensor dot, Tensor reshaping, Geometric interpretation of tensor operations, A geometric interpretation of deep learning

2.1 TENSOR OPERATIONS

2.1.1 Element-wise operations

Q1. Define Tensor? What Does Element-Wise Operations with examples.

Ans: **(Imp.)**

A tensor is a generalization of vector matrices and is easily understood as a multidimensional array. In the general ease, an array of numbers arranged on a regular grid with a variable number of axes is known as a 'tensor'.

Element-wise operations are extremely common operations with tensors in neural network programming.

An element-wise operation is an operation between two tensors that operates on corresponding elements within the respective tensors.

An element-wise operation operates on corresponding elements between tensors.

Two elements are said to be corresponding if the two elements occupy the same position within the tensor. The position is determined by the indexes used to locate each element.

Suppose we have the following two tensors:

```
> t1 = torch.tensor([
      [1,2],
      [3,4]
      ], dtype=torch.float32)
> t2 = torch.tensor([
      [9,8],
       [7,6]
], dtype=torch.float32)
```

Both of these tensors are rank-2 tensors with a shape of 2×2.

This means that we have two axes that both have a length of two elements each. The elements of the first axis are arrays and the elements of the second axis are numbers.

```
# Example of the first axis
> print(t1[0])
tensor([1., 2.])
# Example of the second axis
> print(t1[0][0])
tensor(1.)
```

We know that two elements are said to be corresponding if the two elements occupy the same position within the tensor, and the position is determined by the indexes used to locate each element. Let's see an example of corresponding elements.

```
> t1[0][0]
      tensor(1.)
> t2[0][0]
      tensor(9.)
```

This allows us to see that the corresponding element for the 1 in t1 is the 9 in t2.

The correspondence is defined by the indexes. This is important because it reveals an important feature of element-wise operations. We can deduce that tensors must have the same number of elements in order to perform an element-wise operation.

Two tensors must have the same shape in order to perform element-wise operations on them.

Addition is An Element-Wise Operation:

Let's look at our first element-wise operation, addition. Don't worry. It will get a more interesting.

```
> t1 + t2
tensor([[10., 10.],
        [10., 10.]])
```

This allow us to see that addition between tensors is an element-wise operation. Each pair of elements in corresponding locations are added together to produce a new tensor of the same shape.

So, addition is an element-wise operation, and in fact, all the arithmetic operations, add, subtract, multiply, and divide are element-wise operations.

Arithmetic Operations Are Element-Wise Operations:

An operation we commonly see with tensors are arithmetic operations using scalar values. There are two ways we can do this:

(1) Using these symbolic operations:

```
>print(t + 2)
tensor([[3., 4.],
        [5., 6.]])
>print(t - 2)
tensor([[-1., 0.],
        [ 1., 2.]])
>print(t * 2)
tensor([[2., 4.],
        [6., 8.]])
>print(t / 2)
tensor([[0.5000, 1.0000],
        [1.5000, 2.0000]])
```

(or) equivalently, to these built-in tensor object methods:

```
> print(t1.add(2))
tensor([[3., 4.],
        [5., 6.]])
> print(t1.sub(2))
tensor([[-1., 0.],
        [ 1., 2.]])
> print(t1.mul(2))
tensor([[2., 4.],
        [6., 8.]])
> print(t1.div(2))
tensor([[0.5000, 1.0000],
        [1.5000, 2.0000]])
```

Both of these options work the same. We can see that in both cases, the scalar value, 2, is applied to each element with the corresponding arithmetic operation.

Something seems to be wrong here. These examples are breaking the rule we established that said element-wise operations operate on tensors of the same shape.

Scalar values are Rank-0 tensors, which means they have no shape, and our tensor t1 is a rank-2 tensor of shape 2×2.

So how does this fit in, Let's break it down.

- The first solution that may come to mind is that the operation is simply using the single scalar value and operating on each element within the tensor.
- This logic kind of works. However, it's a bit misleading, and it breaks down in more general situations where we're note using a scalar.
- To think about these operations differently, we need to introduce the concept of tensor broadcasting or broadcasting.

2.2 BROADCASTING

Q2. Write about Broadcasting in Tensors with suitable examples?

Ans: **(Imp.)**

"Broadcasting describes how tensors with different shapes are treated during arithmetic operations."

Tensor broadcasting is about bringing the tensors of different dimensions/shape to the compatible shape such that arithmetic operations can be performed on them. In broadcasting, the smaller array is found, the new axes are added as per the larger array, and data is added appropriately to the transformed array.

Broadcasting Example 1: Same Shapes

For example, it might be relatively easy to look at these two rank-2 tensors and figure out what the sum of them would be.

Tensor 1:

[[1, 2, 3],]

rank: 2

shape: (1,3)

Tensor 2:

[[4, 5, 6],]

rank: 2

shape: (1,3)

They have the same shape, so we just take the element-wise sum of the two tensors, where we calculate the sum element-by-element, and our resulting tensor looks like this.

Tensor 1 + Tensor 2:

[[1, 2, 3],]

+

[[4, 5, 6],]

[[5, 7, 9],]

rank: 2

shape: (1,3)

Now, since these two tensors have the same shape, (1, 3), no broadcasting is happening here. Remember, broadcasting comes into play when we have tensors with different shapes.

Example 2: Same Rank, Different Shapes

So, what would happen if our two rank-2 tensors looked like this, and we wanted to sum them

Tensor 1:

[[1, 2, 3],]

rank: 2

shape: (1,3)

Tensor 2:

[[4],

[5],

[6]]

rank: 2

shape: (3,1)

We have one tensor with shape (1, 3), and the other with shape (3, 1). Well, here is where broadcasting will come into play.

Before we cover how this is done, go ahead and pause and see just intuitively, what comes to mind as the resulting tensor from adding these two together. Give it a go, write it down, and keep what you write handy because we'll circle back around to what you wrote later.

We're first going to look at the result, and then we'll go over how we arrived there.

Our result from summing these two tensors is this (3, 3) tensor.

Tensor 1 + Tensor 2:

[[1, 2, 3],]

+

[[4],

[5],

[6]]

[[5, 6, 7],

[6, 7, 8],

[7, 8, 9]]

rank: 2

shape: (3,3)

Here's how broadcasting works.

We have two tensors with different shapes. The goal of broadcasting is to make the tensors have the same shape so we can perform element-wise operations on them.

First, we have to see if the operation we're trying to do is even possible between the given tensors. Based on the tensors' original shapes, there may not be a way to reshape them to force them to be compatible, and if we can't do that, then we can't use broadcasting.

Tensor 1 Broadcast to Shape (3,3):

Before:

[[1, 2, 3],]

After:

[[1, 2, 3],

[1, 2, 3],

[1, 2, 3]]

The values in our (3, 1) tensor will now be broadcast to this (3, 3) tensor.

Tensor 2 Broadcast to Shape (3,3):

Before:

[[4],

[5],

[6]]

After:

[[4, 4, 4],

[5, 5, 5],

[6, 6, 6]]

We can now easily take the element-wise sum of these two to get this resulting (3, 3) tensor.

[[1, 2, 3],

[1, 2, 3],

[1, 2, 3]]

+

[[4, 4, 4],

[5, 5, 5],

[6, 6, 6]]

[[5, 6, 7],

[6, 7, 8],

[7, 8, 9]]

Broadcasting Example 3: Different Ranks

What if we wanted to multiply this rank-2 tensor of shape (1, 3) with this rank-0 tensor, better known as a scalar?

Tensor 1:

[[1, 2, 3],]

rank: 2

shape: (1,3)

Tensor 2:

5

rank: 0

shape: ()

2.3 TENSOR DOT

Q3. Discuss about Tensor dot with suitable examples?

Ans:

numpy.tensordot

numpy.tensordot(a,b, axes = 2)

Compute tensor dot product along specified axes.

Given two tensors, a and b, and an array_like object containing two array _ like objects, (a_axes, b_axes), sum the products of a's and b's elements (components) over the axes specified by a_axes and b_axes. The third argument can be a single non-negative integer_like scalar, N; if it is such, then the last N dimensions of a and the first N dimensions of b are summed over.

Parameters:

a, barray_like

Tensors to "dot".

axesint or (2,) array_like

- integer_like If an int N, sum over the last N axes of a and the first N axes of b in order. The sizes of the corresponding axes must match.
- (2,) array_like Or, a list of axes to be summed over, first sequence applying to a, second to b. Both elements array_like must be of the same length.

Returns:

Output ndarray

The tensor dot product of the input.

Notes:

Three common use cases are:

- axes = 0 : tensor product $a \otimes b$
- axes = 1 : tensor dot product a.b
- axes = 2 : (default) tensor double contraction a : b

When axes isinteger_like, the sequence for evaluation will be: first the -Nth axis in a and 0th axis in b, and the -1th axis in a and Nth axis in b last.

When there is more than one axis to sum over - and they are not the last (first) axes of a (b) - the argument axes should consist of two sequences of the same length, with the first axis to sum over given first in both sequences, the second axis second, and so forth.

The shape of the result consists of the non-contracted axes of the first tensor, followed by the non-contracted axes of the second.

Examples

A "traditional" example:

```
a = np.arange(60.).reshape(3,4,5)
b = np.arange(24.).reshape(4,3,2)
c = np.tensordot(a,b, axes=([1,0],[0,1]))
c. shape
(5, 2)
c
array([[4400., 4730.],
[4532., 4874.],
[4664., 5018.],
[4796., 5162.],
[4928., 5306.]])
# A slower but equivalent way of computing the same...
d = np.zeros((5,2))
for i in range(5):
for j in range(2):
for k in range(3):
for n in range(4):
d[i,j] += a[k,n,i] * b[n,k,j]
c == d
array([[ True, True],
 [ True, True],
 [ True, True],
 [ True, True],
 [ True, True]])
```

An extended example taking advantage of the overloading of + and *:

```
a = np.array(range(1, 9))
a.shape = (2, 2, 2)
A = np.array((‘a’, ‘b’, ‘c’, ‘d’), dtype=object)
A.shape = (2, 2)
a; A
array([[[1, 2],
        [3, 4]],
       [[5, 6],
        [7, 8]]])
array([[‘a’, ‘b’],
       [‘c’, ‘d’]], dtype=object)
np.tensordot(a, A) # third argument default is 2 for double-contraction
array([‘abbcccdddd’, ‘aaaaabbbbbbccccccddddddd’], dtype=object)
np.tensordot(a, A, 1)
array([[[‘acc’, ‘bdd’],
        [‘aaacccc’, ‘bbbdddd’]],
       [[‘aaaaacccccc’, ‘bbbbbddddddd’],
        [‘aaaaaaaccccccc’, ‘bbbbbbbddddddd’]]], dtype=object)
np.tensordot(a, A, 0) # tensor product (result too long to incl.)
array([[[[[‘a’, ‘b’],
          [‘c’, ‘d’]],
          ...
np.tensordot(a, A, (0, 1))
array([[[‘abbbbb’, ‘cddddd’],
        [‘aabbbbbb’, ‘ccddddddd’]],
       [[‘aaabbbbbbb’, ‘cccddddddd’],
        [‘aaaabbbbbbbb’, ‘ccccddddddddd’]]], dtype=object)
np.tensordot(a, A, (2, 1))
array([[[‘abb’, ‘cdd’],
        [‘aaabbbb’, ‘cccdddd’]],
       [[‘aaaaabbbbbb’, ‘ccccddddddd’],
        [‘aaaaaaabbbbbbbb’, ‘ccccccddddddddd’]]], dtype=object)
np.tensordot(a, A, ((0, 1), (0, 1)))
array([‘abbbcccccddddddd’, ‘aabbbbccccccddddddddd’], dtype=object)
np.tensordot(a, A, ((2, 1), (1, 0)))
array([‘acccbbdddd’, ‘aaaaaccccccbbbbbbddddddddd’], dtype=object)
```

2.4 Tensor Reshaping

Q4. Explain about Reshaping a Tensor with suitable Examples?

Ans: **(Imp.)**

Reshaping allows us to change the shape with the same data and number of elements as self but with the specified shape, which means it returns the same data as the specified array, but with different specified dimension sizes.

Creating Tensor for demonstration:

Python code to create a 1D Tensor and display it.

```
# import torch module
    import torch
# create an 1 D etnsor with 8 elements
    a = torch.tensor([1,2,3,4,5,6,7,8])
# display tensor shape
    print(a.shape)
# display tensor
    a
```

Output:

```
torch.Size([8])
tensor([1, 2, 3, 4, 5, 6, 7, 8])
```

Method 1: Using reshape() Method

This method is used to reshape the given tensor into a given shape (Change the dimensions)

Syntax: tensor.reshape([row,column])

where,

- Tensor is the input tensor
- Row represents the number of rows in the reshaped tensor
- Column represents the number of columns in the reshaped tensor

Example 1: Python program to reshape a 1 D tensor to a two-dimensional tensor.

```
# import torch module
    import torch
# create an 1 D etnsor with 8 elements
    a = torch.tensor([1, 2, 3, 4, 5, 6, 7, 8])
# display tensor shape
    print(a.shape)
# display actual tensor
    print(a)
# reshape tensor into 4 rows and 2 columns
    print(a.reshape([4, 2]))
# display shape of reshaped tensor
    print(a.shape)
```

Output:

```
torch.Size([8])
tensor([1, 2, 3, 4, 5, 6, 7, 8])
tensor([[1, 2],
        [3, 4],
        [5, 6],
        [7, 8]])
torch.Size([8])
```

Example 2: Python code to reshape tensors into 4 rows and 2 columns

```
# import torch module
    import torch
# create an 1 D etnsor with 8 elements
    a = torch.tensor([1, 2, 3, 4, 5, 6, 7, 8])
# display tensor shape
    print(a.shape)
# display actual tensor
    print(a)
# reshape tensor into 4 rows and 2 columns
    print(a.reshape([4, 2]))
# display shape
    print(a.shape)
```

Output:

```
torch.Size([8])
tensor([1, 2, 3, 4, 5, 6, 7, 8])
```

```
tensor([[1, 2],
        [3, 4],
        [5, 6],
        [7, 8]])
         torch.Size([8])
```

Example 3: Python code to reshape tensor into 8 rows and 1 column.

```
# import torch module
        import torch
# create an 1 D etnsor with 8 elements
        a = torch.tensor([1, 2, 3, 4, 5, 6, 7, 8])
# display tensor shape
        print(a.shape)
# display actual tensor
        print(a)
# reshape tensor into 8 rows and 1 column
        print(a.reshape([8, 1]))
# display shape
        print(a.shape)
# import torch module
        import torch
# create an 1 D etnsor with 8 elements
        a = torch.tensor([1, 2, 3, 4, 5, 6, 7, 8])
# display tensor shape
        print(a.shape)
# display actual tensor
        print(a)
# reshape tensor into 8 rows and 1 column
        print(a.reshape([8, 1]))
# display shape
        print(a.shape)
```

Method 2 : Using flatten() method

flatten() is used to flatten an N-Dimensional tensor to a 1D Tensor.

Syntax: torch.flatten(tensor)

Where, tensor is the input tensor

Example 1: Python code to create a tensor with 2 D elements and flatten this vector

```
# import torch module
        import torch
# create an 2 D tensor with 8 elements each
a = torch.tensor([[1,2,3,4,5,6,7,8],
                  [1,2,3,4,5,6,7,8]])
# display actual tensor
        print(a)
# flatten a tensor with flatten() function
        print(torch.flatten(a))
```

Example 2: Python code to create a tensor with 3 D elements and flatten this vector

```
# import torch module
        import torch
# create an 3 D tensor with 8 elements each
a = torch.tensor([[[1,2,3,4,5,6,7,8],
                   [1,2,3,4,5,6,7,8]],
                   [[1,2,3,4,5,6,7,8],
                   [1,2,3,4,5,6,7,8]]])
# display actual tensor
        print(a)
# flatten a tensor with flatten() function
        print(torch.flatten(a))
```

Output:

```
tensor ([[[1, 2, 3, 4, 5, 6, 7, 8],
         [1, 2, 3, 4, 5, 6, 7, 8]],
         [[1, 2, 3, 4, 5, 6, 7, 8],
         [1, 2, 3, 4, 5, 6, 7, 8]]])
```

tensor ([1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8])

Method 3: Using view() method

view() is used to change the tensor in two-dimensional format IE rows and columns. We have to specify the number of rows and the number of columns to be viewed.

Syntax: tensor.view(no_of_rows,no_of_columns)

where,

- tensor is an input one dimensional tensor
- no_of_rowsis the total number of the rows that the tensor is viewed
- no_of_columns is the total number of the columns that the tensor is viewed.

Example 1: Python program to create a tensor with 12 elements and view with 3 rows and 4 columns and vice versa.

```
# importing torch module
    import torch
# create one dimensional tensor 12 elements
    a=torch.FloatTensor([24, 56, 10, 20, 30,
                         40, 50, 1, 2, 3, 4, 5])
# view tensor in 4 rows and 3 columns
    print(a.view(4, 3))
# view tensor in 3 rows and 4 columns
    print(a.view(3, 4))
```

Out put:

```
tensor([[24., 56., 10.],
    [20., 30., 40.],
    [50., 1., 2.],
    [ 3., 4., 5.]])
tensor([[24., 56., 10., 20.],
    [30., 40., 50., 1.],
    [ 2., 3., 4., 5.]])
```

Example 2: Python code to change the view of a tensor into 10 rows and one column and vice versa.

```
# importing torch module
    import torch
# create one dimensional tensor 10 elements
    a = torch.FloatTensor([24, 56, 10, 20, 30,
                           40, 50, 1, 2, 3])
# view tensor in 10 rows and 1 column
    print(a.view(10, 1))
# view tensor in 1 row and 10 columns
    print(a.view(1, 10))
```

Output:

```
tensor([[24.],
    [56.],
    [10.],
    [20.],
    [30.],
    [40.],
    [50.],
    [ 1.],
    [ 2.],
    [ 3.]])
tensor([[24., 56., 10., 20., 30., 40., 50., 1., 2., 3.]])
```

Method 4: Using resize() method

This is used to resize the dimensions of the given tensor.

Syntax : tensor. resize _ (no _ of _ tensors, no _ of_rows,no_of_columns)

where:

- tensor is the input tensor
- no_of_tensorsrepresents the total number of tensors to be generated
- no_of_rowsrepresents the total number of rows in the new resized tensor
- no_of_columnsrepresents the total number of columns in the new resized tensor

Example 1: Python code to create an empty one D tensor and create 4 new tensors with 4 rows and 5 columns

```
# importing torch module
    import torch
# create one dimensional tensor
    a = torch.Tensor()
# resize the tensor to 4 tensors.
# each tensor with 4 rows and 5 columns
    print(a.resize_(4, 4, 5))
```

Output:

```
tensor([[[-3.7238e-25,  3.0841e-41,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00]],

        [[ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00]],

        [[ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00]],

        [[ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00]]])
```

2.5 Geometric Interpretation of Tensor Operations

Q5. Discuss about Geometric interpretation of tensor operations?

Ans: **(Imp.)**

The contents of the tensors manipulated by tensor operations can be interpreted as coordinates of points in some geometric space, all tensor operations have a geometric interpretation. For instance, let's consider addition. We'll start with the following vector

A = [0.5, 1]

It's a point in a 2D space. It's common to picture a vector as an arrow linking the origin to the point, as shown in figure.

Figure. A point in a 2D space

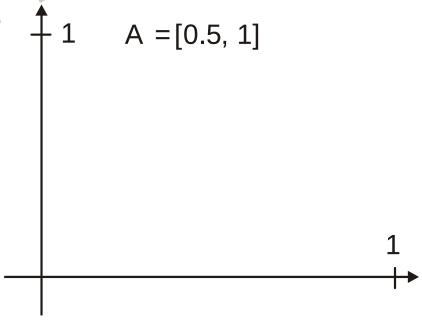
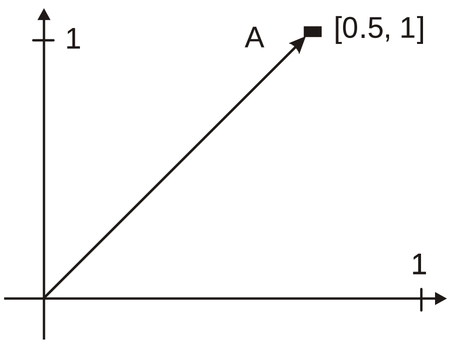
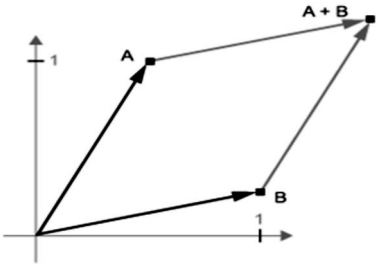


Fig.: A point in a 2D space pictured as an arrow



Consider a new point, B = [1, 0.25], which we'll add to the previous one. This is done geometrically by chaining together the vector arrows, with the resulting location being the vector representing sum of the previous two vectors (see figure below).

Fig.: Geometric interpretation of the sum of two vectors



In general, elementary geometric operations such as affine transformations, rotations, scaling, and so on can be expressed as tensor operations. For instance, a rotation of a 2D vector by an angle theta can be achieved via a dot product with a 2 × 2 matrix R = [u, v], where u and v are both vectors of the plane: u = [cos(theta), sin(theta)] and v = [-sin(theta), cos(theta)].

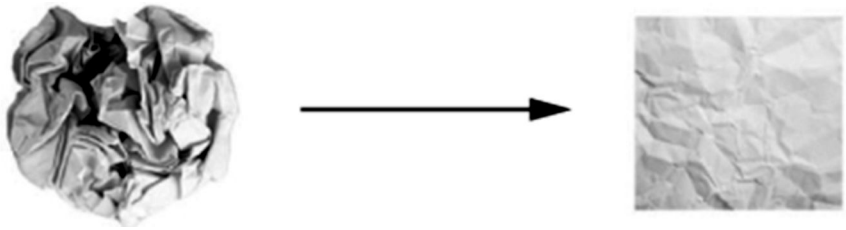
2.6 A Geometric Interpretation of Deep Learning

Q6. Explain briefly about Geometric Interpretation of Deep Learning.

Ans:

Neural networks consist entirely of chains of tensor operations and that all of these tensor operations are just geometric transformations of the input data. It follows that you can interpret a neural network as a very complex geometric transformation in a high-dimensional space, implemented via a long series of simple steps.

In 3D, the following mental image may prove useful. Imagine two sheets of colored paper: one red and one blue. Put one on top of the other. Now crumple them together into a small ball. That crumpled paper ball is your input data, and each sheet of paper is a class of data in a classification problem. What a neural network (or any other machine-learning model) is meant to do is figure out a transformation of the paper ball that would uncrumple it, so as to make the two classes cleanly separable again. With deep learning, this would be implemented as a series of simple transformations of the 3D space, such as those you could apply on the paper ball with your fingers, one movement at a time.



Uncrumpling paper balls is what machine learning is about: finding neat representations for complex, highly folded data manifolds. At this point, you should have a pretty good intuition as to why deep learning excels at this: it takes the approach of incrementally decomposing a complicated geometric transformation into a long chain of elementary ones, which is pretty much the strategy a human would follow to uncrumple a paper ball. Each layer in a deep network applies a transformation that disentangles the data a little and a deep stack of layers makes tractable an extremely complicated disentanglement process.

Short Question and Answers

1. Write about Broadcasting in Tensors with suitable examples?

Ans :

"Broadcasting describes how tensors with different shapes are treated during arithmetic operations."

Tensor broadcasting is about bringing the tensors of different dimensions/shape to the compatible shape such that arithmetic operations can be performed on them. In broadcasting, the smaller array is found, the new axes are added as per the larger array, and data is added appropriately to the transformed array.

Broadcasting Example 1: Same Shapes

For example, it might be relatively easy to look at these two rank-2 tensors and figure out what the sum of them would be.

Tensor 1:
[[1, 2, 3],]
rank: 2
shape: (1,3)
Tensor 2:
[[4, 5, 6],]
rank: 2
shape: (1,3)

They have the same shape, so we just take the element-wise sum of the two tensors, where we calculate the sum element-by-element, and our resulting tensor looks like this.

Tensor 1 + Tensor 2:
[[1, 2, 3],]
+
[[4, 5, 6],]
———————————
[[5, 7, 9],]
rank: 2
shape: (1,3)

2. Discuss about Tensor dot with suitable examples?

Ans :

numpy.tensordot

numpy.tensordot(a, b, axes=2)

Compute tensor dot product along specified axes.

Given two tensors, a and b, and an array_like object containing two array_like objects, (a_axes, b_axes), sum the products of a's and b's elements (components) over the axes specified by a_axes and b_axes. The third argument can be a single non-negative integer_like scalar, N; if it is such, then the last N dimensions of a and the first N dimensions of b are summed over.

Three common use cases are:

- axes = 0 : tensor product a—"b
- axes = 1 : tensor dot product aÅ"b
- axes = 2 : (default) tensor double contraction a:b

When axes isinteger_like, the sequence for evaluation will be: first the -Nth axis in a and 0th axis in b, and the -1th axis in a and Nth axis in b last.

When there is more than one axis to sum over - and they are not the last (first) axes of a (b) - the argument axes should consist of two sequences of the same length, with the first axis to sum over given first in both sequences, the second axis second, and so forth.

The shape of the result consists of the non-contracted axes of the first tensor, followed by the non-contracted axes of the second.

Examples

A "traditional" example:

```
a = np.arange(60.).reshape(3,4,5)
b = np.arange(24.).reshape(4,3,2)
c = np.tensordot(a,b, axes=([1,0],[0,1]))
c.shape
(5, 2)
c.array([[4400., 4730.],
[4532., 4874.],
[4664., 5018.],
[4796., 5162.],
[4928., 5306.]])
```


3. Explain about Reshaping a Tensor with suitable examples?

Ans :

Reshaping allows us to change the shape with the same data and number of elements as self but with the specified shape, which means it returns the same data as the specified array, but with different specified dimension sizes.

Method 1: Using reshape() Method

This method is used to reshape the given tensor into a given shape (Change the dimensions)

Syntax: tensor.reshape([row,column])

where,

- tensor is the input tensor
- row represents the number of rows in the reshaped tensor
- column represents the number of columns in the reshaped tensor

Example 1: Python program to reshape a 1 D tensor to a two-dimensional tensor.

```
# import torch module
import torch
```

```
# create an 1 D etnsor with 8 elements
a = torch.tensor([1, 2, 3, 4, 5, 6, 7, 8])
# display tensor shape
print(a.shape)
# display actual tensor
print(a)
# reshape tensor into 4 rows and 2 columns
print(a.reshape([4, 2]))
# display shape of reshaped tensor
print(a.shape)
```

Output:

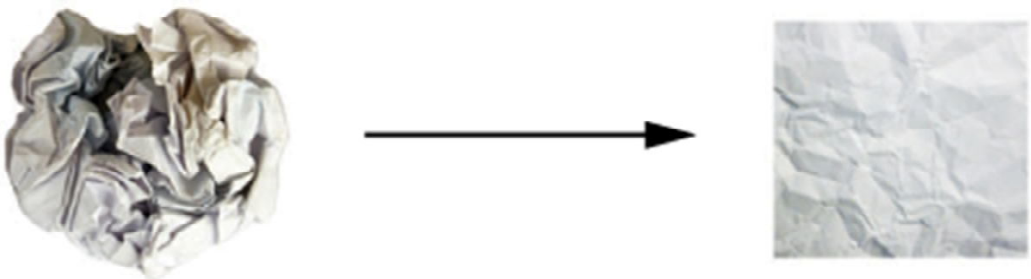
```
torch.Size([8])
tensor([1, 2, 3, 4, 5, 6, 7, 8])
tensor([[1, 2],
        [3, 4],
        [5, 6],
        [7, 8]])
torch.Size([8])
```

4. Discuss about geometric interpretation of deep learning?

Ans :

Neural networks consist entirely of chains of tensor operations and that all of these tensor operations are just geometric transformations of the input data. It follows that you can interpret a neural network as a very complex geometric transformation in a high-dimensional space, implemented via a long series of simple steps.

In 3D, the following mental image may prove useful. Imagine two sheets of colored paper: one red and one blue. Put one on top of the other. Now crumple them together into a small ball. That crumpled paper ball is your input data, and each sheet of paper is a class of data in a classification problem. What a neural network (or any other machine-learning model) is meant to do is figure out a transformation of the paper ball that would uncrumple it, so as to make the two classes cleanly separable again. With deep learning, this would be implemented as a series of simple transformations of the 3D space, such as those you could apply on the paper ball with your fingers, one movement at a time.



Uncrumpling paper balls is what machine learning is about: finding neat representations for complex, highly folded data manifolds. At this point, you should have a pretty good intuition as to why deep learning excels at this: it takes the approach of incrementally decomposing a complicated geometric transformation into a long chain of elementary ones, which is pretty much the strategy a human would follow to uncrumple a paper ball. Each layer in a deep network applies a transformation that disentangles the data a little—and a deep stack of layers makes tractable an extremely complicated disentanglement process.

5. What are the three steps to developing the necessary assumption structure in Deep learning?

Ans :

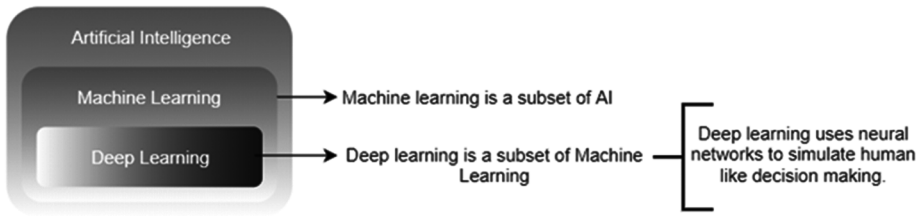
The procedure of developing an assumption structure involves three specific actions.

- The first step contains algorithm development. This particular process is lengthy.
- The second step contains algorithm analyzing, which represents the in-process methodology.
- The third step is about implementing the general algorithm in the final procedure. The entire framework is interlinked and required for throughout the process.

6. What are the main differences between AI, Machine Learning, and Deep Learning?

Ans :

- AI stands for Artificial Intelligence. It is a technique which enables machines to mimic human behavior.
- Machine Learning is a subset of AI which uses statistical methods to enable machines to improve with experiences.



- Deep learning is a part of Machine learning, which makes the computation of multi-layer neural networks feasible. It takes advantage of neural networks to simulate human-like decision making.

7. How can hyperparameters be trained in neural networks?

Ans :

Hyperparameters can be trained using four components as shown below:

- **Batch size:** This is used to denote the size of the input chunk. Batch sizes can be varied and cut into sub-batches based on the requirement.
- **Epochs:** An epoch denotes the number of times the training data is visible to the neural network so that it can train. Since the process is iterative, the number of epochs will vary based on the data.
- **Momentum:** Momentum is used to understand the next consecutive steps that occur with the current data being executed at hand. It is used to avoid oscillations when training.
- **Learning rate:** Learning rate is used as a parameter to denote the time required for the network to update the parameters and learn.

8. What are some of the examples of supervised learning and unsupervised learning algorithms in Deep Learning?

Ans :

There are three main supervised learning algorithms in Deep Learning:

- Artificial neural networks
- Convolutional neural networks
- Recurrent neural networks

There are three main unsupervised learning algorithms in Deep Learning:

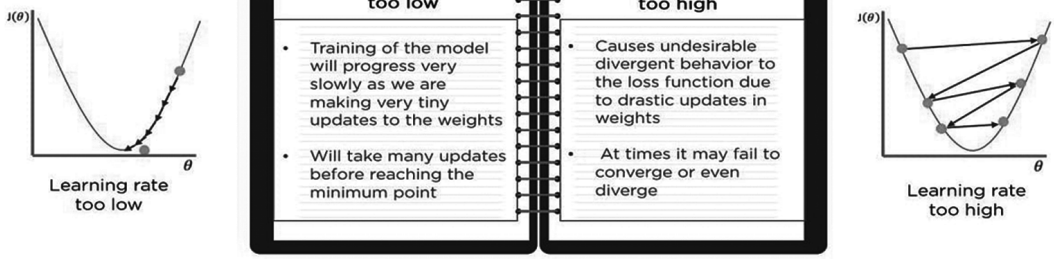
- Autoencoders
- Boltzmann machines
- Self-organizing maps

9. What Will Happen If the Learning Rate Is Set Too Low or Too High?

Ans :

When you're learning rate is too low, training of the model will progress very slowly as we are making minimal updates to the weights. It will take many updates before reaching the minimum point.

If the learning rate is set too high, this causes undesirable divergent behavior to the loss function due to drastic updates in weights. It may fail to converge (model can give a good output) or even diverge (data is too chaotic for the network to train).



10. Explain about resize() method with an example?

Ans :

This is used to resize the dimensions of the given tensor.

Using resize() method

Syntax: tensor.resize_(no_of_tensors,no_of_rows,no_of_columns)

Where:

- tensor is the input tensor
- no_of_tensorsrepresents the total number of tensors to be generated
- no_of_rowsrepresents the total number of rows in the new resized tensor
- no_of_columnsrepresents the total number of columns in the new resized tensor

Example 1:

Python code to create an empty one D tensor and create 4 new tensors with 4 rows and 5 columns

```
# importing torch module
import torch
# create one dimensional tensor
a = torch.Tensor()
# resize the tensor to 4 tensors.
# each tensor with 4 rows and 5 columns
print(a.resize_(4, 4, 5))
```

Output:

```
tensor([[[-3.7238e-25,  3.0841e-41,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00]],

        [[ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00]],

        [[ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00]],

        [[ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
         [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00]]])
```

Choose the Correct Answer

1. Which of the following would have a constant input in each epoch of training a Deep Learning model? [a]

   (a) Weight between input and hidden layer (b) Weight between hidden and output layer

   (c) Biases of all hidden layer neurons (d) Activation function of output layer

2. The number of nodes in the input layer is 10 and the hidden layer is 5. The maximum number of connections from the input layer to the hidden layer are [a]

   (a) 50 (b) less than 50

   (c) more than 50 (d) It is an arbitrary value

3. In a simple MLP model with 8 neurons in the input layer, 5 neurons in the hidden layer and 1 neuron in the output layer. What is the size of the weight matrices between hidden output layer and input hidden layer? [b]

   (a) [1 X 5] , [5 X 8] (b) [5 x 1] , [8 X 5]

   (c) [8 X 5] , [5 X 1] (d) [8 X 5] , [1 X 5]

4. Which of the following functions can be used as an activation function in the output layer if we wish to predict the probabilities of n classes (p1, p2..pk) such that sum of p over all n equals to 1? [a]

   (a) Softmax (b) ReLu

   (c) Sigmoid (d) Tanh

5. Assume a simple MLP model with 3 neurons and inputs = 1,2,3. The weights to the input neurons are 4,5 and 6 respectively. Assume the activation function is a linear constant value of 3. What will be the output? [c]

   (a) 32 (b) 64

   (c) 96 (d) 128

6. For an image recognition problem (recognizing a cat in a photo), which architecture of neural network would be better suited to solve the problem? [b]

   (a) Multi Layer Perceptron (b) Convolutional Neural Network

   (c) Recurrent Neural Network (d) Perceptron

7. Consider the scenario. The problem you are trying to solve has a small amount of data. Fortunately, you have a pre-trained neural network that was trained on a similar problem. Which of the following methodologies would you choose to make use of this pre-trained network? [d]

   (a) Re-train the model for the new dataset

   (b) Assess on every layer how the model performs and only select a few of them

   (c) Fine tune the last couple of layers only

   (d) Freeze all the layers except the last, re-train the last layer

8. A perceptron is _____. [a]

(a) a single layer feed-forward neural network with pre-processing

(b) an auto-associative neural network

(c) a double layer auto-associative neural network

(d) a neural network that contains feedback

9. Neural Networks are complex _____ with many parameters. [a]

(a) Linear Functions (b) Nonlinear Functions

(c) Discrete Functions (d) Exponential Functions

10. Which of the following is an application of Neural Network [d]

(a) Sales forecasting (b) Data validation

(c) Risk management (d) All of the above

Fill in the blanks

1. Madaline stands for _____.
2. The training of the Back Propagation Network is done in _____ stages
3. CAM stands for _____.
4. In the _____ associative memory network, the training input vector and training output vector are the same.
5. The BAM is a _____ associative pattern-marching network that encodes binary or bipolar patterns using Hebbian learning rule
6. The junctions that allow signal transmission between the axons terminals and dendrites are called _____.
7. The weight updating in case of perceptron learning, if $y \neq t$ is _____.
8. Deep Learning algorithms are _____ more accurate than machine learning algorithm in image classification.
9. Why do we normalize the inputs X is _____.
10. Low bias and high variance, we get _____ model.

ANSWERS

1. Multiple Adaptive Linear Neuron
2. three
3. Content Addressable Memories
4. auto
5. Recurrent hetero
6. synapses
7. wi(new) = wi(old) + ±txi
8. 41%
9. It makes the cost function faster to optimize
10. over fitting

One Mark Answers

1. What are tensors?

Ans :

Tensors are multidimensional arrays in Deep Learning that are used to represent data. They represent the data with higher dimensions

2. What is a Boltzmann machine?

Ans :

A Boltzmann machine is a type of recurrent neural network that uses binary decisions.

3 What are some of the advantages of using TensorFlow?

Ans :

- High amount of flexibility and platform independence
- Trains using CPU and GPU
- Supports auto differentiation and its features
- Handles threads and asynchronous computation easily
- Open-source
- Has a large community

4. What is the use of LSTM?

Ans :

LSTM stands for long short-term memory. It is a type of RNN that is used to sequence a string of data. It consists of feedback chains that give it the ability to perform like a general-purpose computational entity.

5. What are some of the examples of supervised learning algorithms in Deep Learning?

Ans :

There are three main supervised learning algorithms in Deep Learning:

- Artificial neural networks
- Convolutional neural networks
- Recurrent neural networks

6. What are the elements in TensorFlow that are programmable?

Ans :

In TensorFlow, users can program three elements:

- Constants
- Variables
- Placeholders

7. Data Normalization.

Ans :

Data normalization is an essential preprocessing step, which is used to rescale values to fit in a specific range.

8. What are the prerequisites for starting in Deep Learning?

Ans :

There are some basic requirements for starting in Deep Learning, which are:

- Machine Learning
- Mathematics
- Python Programming

9. In which layer softmax activation function used?

Ans :

Softmax activation function has to be used in the output layer.

10. What are the Applications of a Recurrent Neural Network (RNN)?

Ans :

The RNN can be used for sentiment analysis, text mining, and image captioning. Recurrent Neural Networks can also address time series problems such as predicting the prices of stocks in a month or quarter.

UNIT III

Gradient-based optimization, Derivative of a tensor operation, Stochastic gradient descent, Chaining derivatives: The Backpropagation algorithm Neural networks: Anatomy, Layers, Models, Loss functions and optimizers

3.1 Gradient-Based Optimization

Q1. Discuss about Gradient-based optimization.

Ans : **(Imp.)**

Gradient descent is an optimization algorithm that's used when training deep learning models. It's based on a convex function and updates its parameters iteratively to minimize a given function to its local minimum.

Gradient Descent

$$\odot_j = \odot_j - \alpha \frac{\partial}{\partial \odot_j} J(\odot_0, \odot_1)$$

Learning Rate

The notation used in the above Formula is given below :

In the above formula,

- α is the learning rate,
- J is the cost function, and
- Θ is the parameter to be updated.

As you can see, the gradient represents the partial derivative of J(cost function) with respect to Θ_j.

Note that, as we reach closer to the global minima, the slope or the gradient of the curve becomes less and less steep, which results in a smaller value of derivative, which in turn reduces the step size or learning rate automatically.

It is the most basic but most used optimizer that directly uses the derivative of the loss function and learning rate to reduce the loss function and tries to reach the global minimum.

Thus, the Gradient Descent Optimization algorithm has many applications including-

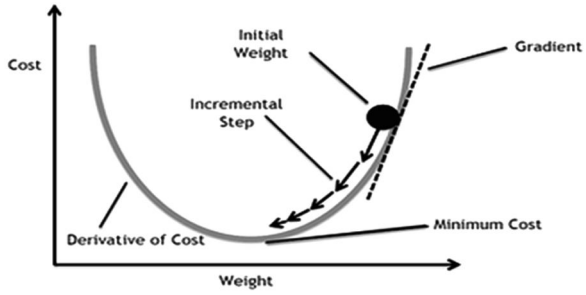
- Linear Regression,
- Classification Algorithms,
- Back propagation in Neural Networks, etc.

Repeat until convergence {

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(q)$$

}

The above-described equation calculates the gradient of the cost function $J(\theta)$ with respect to the network parameters θ for the entire training dataset:



Our aim is to reach at the bottom of the graph (Cost vs weight), or to a point where we can no longer move downhill–a local minimum.

Role of Gradient

In general, Gradient represents the slope of the equation while gradients are partial derivatives and they describe the change reflected in the loss function with respect to the small change in parameters of the function. Now, this slight change in loss functions can tell us about the next step to reduce the output of the loss function.

Role of Learning Rate

Learning rate represents the size of the steps our optimization algorithm takes to reach the global minima. To ensure that the gradient descent algorithm reaches the local minimum we must set the learning rate to an appropriate value, which is neither too low nor too high.

Taking very large steps i.e, a large value of the learning rate may skip the global minima, and the model will never reach the optimal value for the loss function. On the contrary, taking very small steps i.e, a small value of learning rate will take forever to converge.

Thus, the size of the step is also dependent on the gradient value.

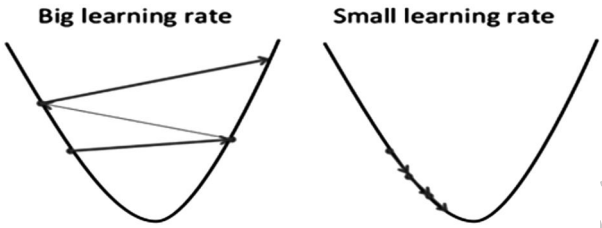


Image Source: Google Images

The gradient represents the direction of increase. But our aim is to find the minimum point in the valley so we have to go in the opposite direction of the gradient. Therefore, we update parameters in the negative gradient direction to minimize the loss.

Algorithm: $\theta = \theta - \alpha \nabla J(\theta)$

In code, Batch Gradient Descent looks something like this:

```
for x in range(epochs):

params_gradient = find_gradient(loss_ function, data, parameters)

parameters = parameters - learning_rate * params_gradient
```

Advantages of Batch Gradient Descent:

1. Easy computation.
2. Easy to implement.
3. Easy to understand.

Disadvantages of Batch Gradient Descent:

1. May trap at local minima.
2. Weights are changed after calculating the gradient on the whole dataset. So, if the dataset is too large then this may take years to converge to the minima.
3. Requires large memory to calculate gradient on the whole dataset.

3.2 Derivative of a Tensor Operation

Q2. Write a note on Derivative of a tensor operation

Ans :

A gradient is the derivative of a tensor operation. It's the generalization of the concept of derivatives to functions of multidimensional inputs: that is, to functions that take tensors as inputs.

Consider an input vector x, a matrix W, a target y, and a loss function loss. You can use W to compute a target candidate y_pred, and compute the loss, or mismatch, between the target candidate y_pred and the target y:

```
y_pred = dot(W, x)

loss_value = loss(y_pred, y)
```

If the data inputs x and y are frozen, then this can be interpreted as a function mapping values of W to loss values:

```
loss_value = f(W)
```

Let's say the current value of W is W0. Then the derivative of f in the point W0 is a tensor gradient(f)(W0) with the same shape as W, where each coefficient gradient(f) (W0)[i, j] indicates the direction and magnitude of the change in loss_value you observe when modifying W0[i, j]. That tensor gradient(f)(W0) is the gradient of the function f(W) = loss_value in W0.

You saw earlier that the derivative of a function f(x) of a single coefficient can be interpreted as the slope of the curve of f. Likewise, gradient(f)(W0) can be interpreted as the tensor describing the curvature of f(W) around W0.

For this reason, in much the same way that, for a function f(x), you can reduce the value of f(x) by moving x a little in the opposite direction from the derivative, with a function f(W) of a tensor, you can reduce f(W) by moving W in the opposite direction from the gradient: for example, W1 = W0 - step * gradient(f)(W0) (where step is a small scaling factor). That means going against the curvature, which intuitively should put you lower on the curve. Note that the scaling factor step is needed because gradient(f)(W0) only approximates the curvature when you're close to W0, so you don't want to get too far from W0.

3.3 Stochastic Gradient Descent

Q3. Explain about Stochastic Gradient Descent with suitable examples.

Ans : **(Imp.)**

Stochastic Gradient Descent

To overcome some of the disadvantages of the GD algorithm, the SGD algorithm comes into the picture as an extension of the Gradient Descent. One of the disadvantages of the Gradient Descent algorithm is that it requires a lot of memory to load the entire dataset at a time to compute the derivative of the loss function. So, In the SGD algorithm, we compute the derivative by taking one data point at a time i.e, tries to update the model's parameters more frequently. Therefore, the model parameters are updated after the computation of loss on each training example.

So, let's have a dataset that contains 1000 rows, and when we apply SGD it will update the model parameters 1000 times in one complete cycle of a dataset instead of one time as in Gradient Descent.

Algorithm: $\theta = \theta - \alpha \nabla J(\theta; x(i); y(i))$, where {x(i) ,y(i)} are the training examples

We want the training, even more, faster, so we take a Gradient Descent step for each training example. Let's see the implications in the image below:

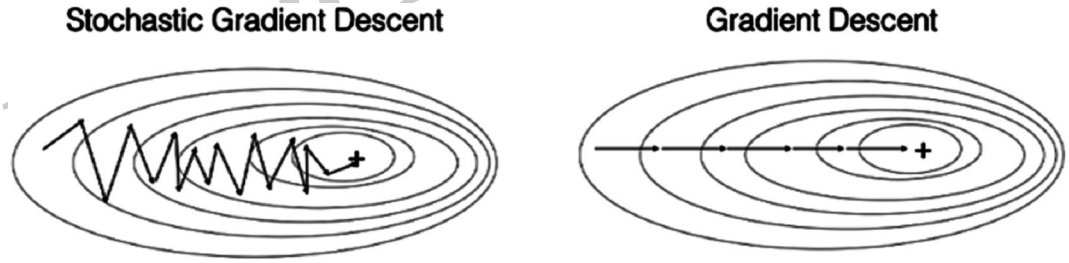


Fig.: SGD vs GD

"+" denotes a minimum of the cost. SGD leads to many oscillations to reach convergence. But each step is a lot faster to compute for SGD than for GD. as it uses only one training example (vs. the whole batch for GD).

To find some insights from the above diagram:

- In the left diagram of the above picture, we have SGD (where 1 per step time) we take a Gradient Descent step for each example and on the right diagram is GD(1 step per entire training set).
- SGD seems to be quite noisy, but at the same time it is much faster than others and also it might be possible that it not converges to a minimum.

It is observed that in SGD the updates take more iterations compared to GD to reach minima. On the contrary, the GD takes fewer steps to reach minima but the SGD algorithm is noisier and takes more iterations as the model parameters are frequently updated parameters having high variance and fluctuations in loss functions at different values of intensities.

Its code snippet simply adds a loop over the training examples and finds the gradient with respect to each of the training examples.

```
for x in range(epochs):
np.random.shuffle(data)
    for example in data:
params_gradient = find_gradient(loss_function, example, parameters)
    parameters = parameters - learning_rate * params_gradient
```

Advantages of Stochastic Gradient Descent

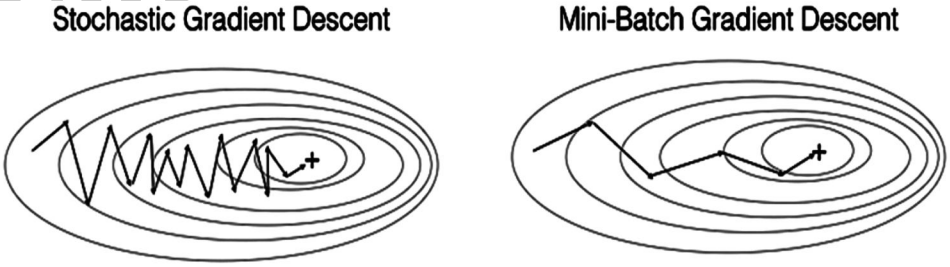
1. Convergence takes less time as compared to others since there are frequent updates in model parameters.
2. Requires less memory as no need to store values of loss functions.
3. May get new minima's.

Disadvantages of Stochastic Gradient Descent

1. High variance in model parameters.
2. Even after achieving global minima, it may overshoots.
3. To reach the same convergence as that of gradient descent, we need to slowly reduce the value of the learning rate.

Mini-Batch Gradient Descent

To overcome the problem of large time complexity in the case of the SGD algorithm. MB-GD algorithm comes into the picture as an extension of the SGD algorithm. It's not all but it also overcomes the problem of Gradient descent. Therefore, It's considered the best among all the variations of gradient descent algorithms. MB-GD algorithm takes a batch of points or subset of points from the dataset to compute derivate.



It is observed that the derivative of the loss function for MB-GD is almost the same as a derivate of the loss function for GD after some number of iterations. But the number of iterations to achieve minima is large for MB-GD compared to GD and the cost of computation is also large.

Therefore, the weight updation is dependent on the derivate of loss for a batch of points. The updates in the case of MB-GD are much noisy because the derivative is not always towards minima.

It updates the model parameters after every batch. So, this algorithm divides the dataset into various batches and after every batch, it updates the parameters.

Algorithm: $\theta = \theta - \alpha . \nabla J(\theta; B(i))$, where {B(i)} are the batches of training examples

In the code snippet, instead of iterating over examples, we now iterate over mini-batches of size 30:

```
for x in range(epochs):
np.random.shuffle(data)
    for batch in get_batches(data, batch_size=30):
params_gradient = find_gradient(loss_function, batch, parameters)
        parameters = parameters - learning_rate * params_gradient
```

Advantages of Mini Batch Gradient Descent

1. Updates the model parameters frequently and also has less variance.
2. Requires not less or high amount of memory i.e requires a medium amount of memory.

Disadvantages of Mini Batch Gradient Descent

1. The parameter updation in MB-SGD is much noisy compared to the weight updation in the GD algorithm.
2. Compared to the GD algorithm, it takes a longer time to converge.
3. May get stuck at local minima.

Challenges with all types of Gradient-based Optimizers

Optimum Learning Rate

Choosing an optimum value of the learning rate. If we choose the learning rate as a too-small value, then gradient descent may take a very long time to converge. For more about this challenge, refer to the above section of Learning Rate which we discussed in the Gradient Descent Algorithm.

Constant Learning Rate

For all the parameters, they have a constant learning rate but there may be some parameters that we may not want to change at the same rate.

Local minimum

May get stuck at local minima i.e, not reach up to the local minimum.

3.4 Chaining Derivatives: The Backpropagation Algorithm

Q4. Explain in detail about The Backpropagation algorithm.

Ans : **(Imp.)**

Backpropagation evaluates the expression for the derivative of the cost function as a product of derivatives between each layer from left to right — "backwards" — with the gradient of the weights between each layer being a simple modification of the partial products (the "backwards propagated error) we casually assumed that because a function is differentiable, we can explicitly compute its derivative. In practice, a neural network function consists of many tensor operations chained together, each of which has a simple, known derivative. For instance, this is a network f composed of three tensor operations, a, b, and c, with weight matrices W1, W2, and W3:

f(W1, W2, W3) = a(W1, b(W2, c(W3)))

Calculus tells us that such a chain of functions can be derived using the following identity, called the chain rule: f(g(x)) = f'(g(x)) * g'(x). Applying the chain rule to the computation of the gradient values of a neural network gives rise to an algorithm called Backpropagation (also sometimes called reverse-mode

differentiation). Backpropagation starts with the final loss value and works backward from the top layers to the bottom layers, applying the chain rule to compute the contribution that each parameter had in the loss value.

Nowadays, and for years to come, people will implement networks in modern frameworks that are capable of symbolic differentiation, such as TensorFlow. This means that, given a chain of operations with a known derivative, they can compute a gradient function for the chain (by applying the chain rule) that maps network parameter values to gradient values. When you have access to such a function, the backward pass is reduced to a call to this gradient function. Thanks to symbolic differentiation, you'll never have to implement the Backpropagation algorithm by hand. For this reason, we won't waste your time and your focus on deriving the exact formulation of the Backpropagation algorithm in these pages. All you need is a good understanding of how gradient-based optimization works.

This was the input data:

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

Now we understand that the input images are stored in Numpy tensors, which are here formatted as float32 tensors of shape (60000, 784) (training data) and (10000, 784) (test data), respectively.

This was our network:

```
network = models.Sequential()
network.add(layers.Dense(512,activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

Now we understand that this network consists of a chain of two Dense layers, that each layer applies a few simple tensor operations to the input data, and that these operations involve weight tensors. Weight tensors, which are attributes of the layers, are where the knowledge of the network persists.

This was the network-compilation step

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

Now we understand that categorical_crossentropy is the loss function that's used as a feedback signal for learning the weight tensors, and which the training phase will attempt to minimize. You also know that this reduction of the loss happens via mini-batch stochastic gradient descent. The exact rules governing a specific use of gradient descent are defined by the rmsprop optimizer passed as the first argument.

Finally, this was the training loop

```
network.fit(train_images,train_labels,epochs=5,batch_size=128)
```

Now we understand what happens when you call fit: the network will start to iterate on the training data in mini-batches of 128 samples, 5 times over (each iteration over all the training data is called an epoch). At each iteration, the network will compute the gradients of the weights with regard to the loss on

the batch, and update the weights accordingly. After these 5 epochs, the network will have performed 2,345 gradient updates (469 per epoch), and the loss of the network will be sufficiently low that the network will be capable of classifying handwritten digits with high accuracy.

3.5 Neural Networks

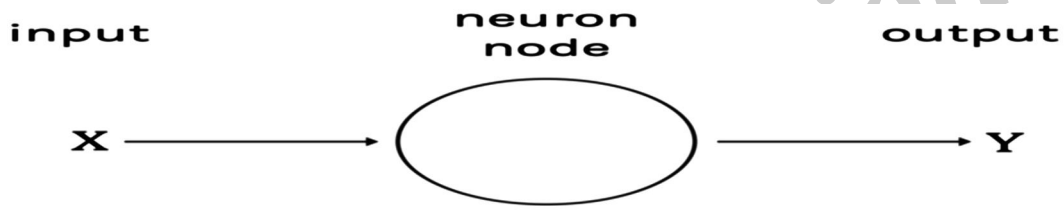
3.5.1 Anatomy

Q5. Write a note on Anatomy of a neural network.

Ans : **(Imp.)**

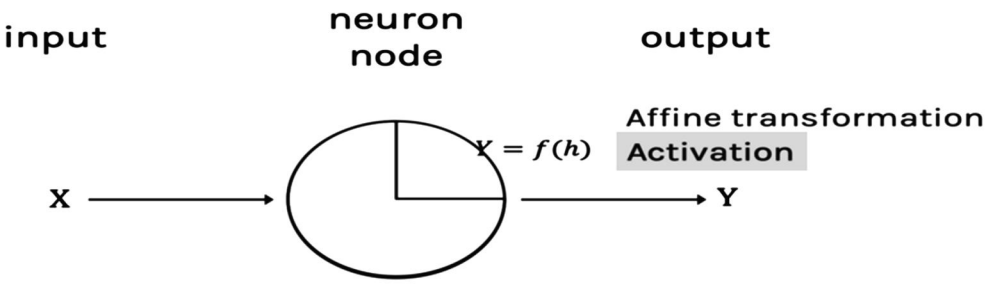
Anatomy of a neural network

Artificial neural networks are one of the main tools used in machine learning. As the "neural" part of their name suggests, they are brain-inspired systems which are intended to replicate the way that we humans learn. Neural networks consist of input and output layers, as well as (in most cases) a hidden layer consisting of units that transform the input into something that the output layer can use. They are excellent tools for finding patterns which are far too complex or numerous for a human programmer to extract and teach the machine to recognize.

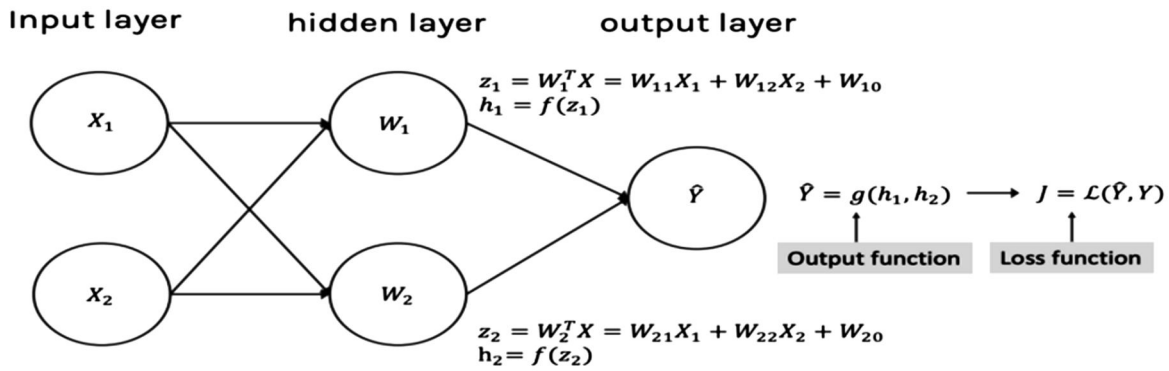


While vanilla neural networks (also called "perceptrons") have been around since the 1940s, it is only in the last several decades where they have become a major part of artificial intelligence. This is due to the arrival of a technique called backpropagation (which we discussed in the previous tutorial), which allows networks to adjust their neuron weights in situations where the outcome doesn't match what the creator is hoping for — like a network designed to recognize dogs, which misidentifies a cat, for example.

The fact that neural networks make use of affine transformations in order to concatenate input features together that converge at a specific node in the network. This concatenated input is then passed through an activation function, which evaluates the signal response and determines whether the neuron should be activated given the current inputs.as this can be an important factor in obtaining a functional network. So far we have only talked about sigmoid as an activation function but there are several other choices, and this is still an active area of research in the machine learning literature.



This idea can be extended to multilayer and multi-feature networks in order to increase the explanatory power of the network by increasing the number of degrees of freedom (weights and biases) of the network, as well as the number of features available which the network can use to make predictions.

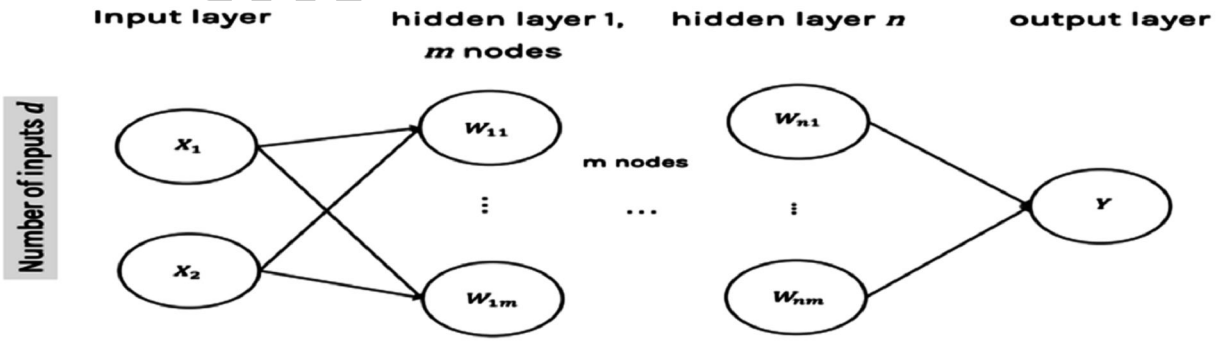


Finally, The network parameters (weights and biases) could be updated by assessing the error of the network. This is done using backpropagation through the network in order to obtain the derivatives for each of the parameters with respect to the loss function, and then gradient descent can be used to update these parameters in an informed manner such that the predictive power of the network is likely to improve.

Together, the process of assessing the error and updating the parameters is what is referred to as training the network. This can only be done if the ground truth is known, and thus a training set is needed in order to generate a functional network. The performance of the network can then be assessed by testing it on unseen data, which is often known as a test set.

Neural networks have a large number of degrees of freedom and as such, they need a large amount of data for training to be able to make adequate predictions, especially when the dimensionality of the data is high (as is the case in images, for example — each pixel is counted as a network feature).

A generalized multilayer and multi-featured network looks like this:



Generalized multilayer perceptron with n hidden layers, m nodes, and d input features.

We have m nodes, where m refers to the width of a layer within the network. Notice that this is no relation between the number of features and the width of a network layer.

We also have n hidden layers, which describe the depth of the network. In general, anything that has more than one hidden layer could be described as deep learning. Sometimes, networks can have hundreds of hidden layers, as is common in some of the state-of-the-art convolutional architectures used for image analysis.

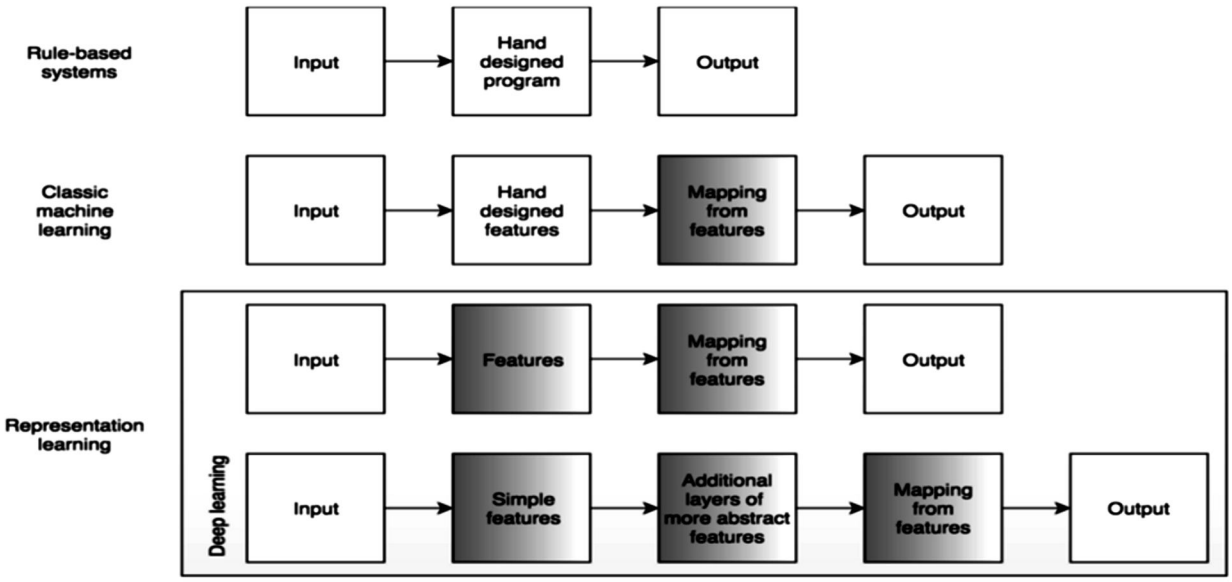
The number of inputs, d, is pre-specified by the available data. For an image, this would be the number of pixels in the image after the image is flattened into a one-dimensional array, for a normal Pandas data frame, d would be equal to the number of feature columns.

In general, it is not required that the hidden layers of the network have the same width (number of nodes); the number of nodes may vary across the hidden layers. The output layer may also be of an arbitrary dimension depending on the required output. If you are trying to classify images into one of ten classes, the output layer will consist of ten nodes, one each corresponding to the relevant output class — this is the case for the popular MNIST database of handwritten numbers.

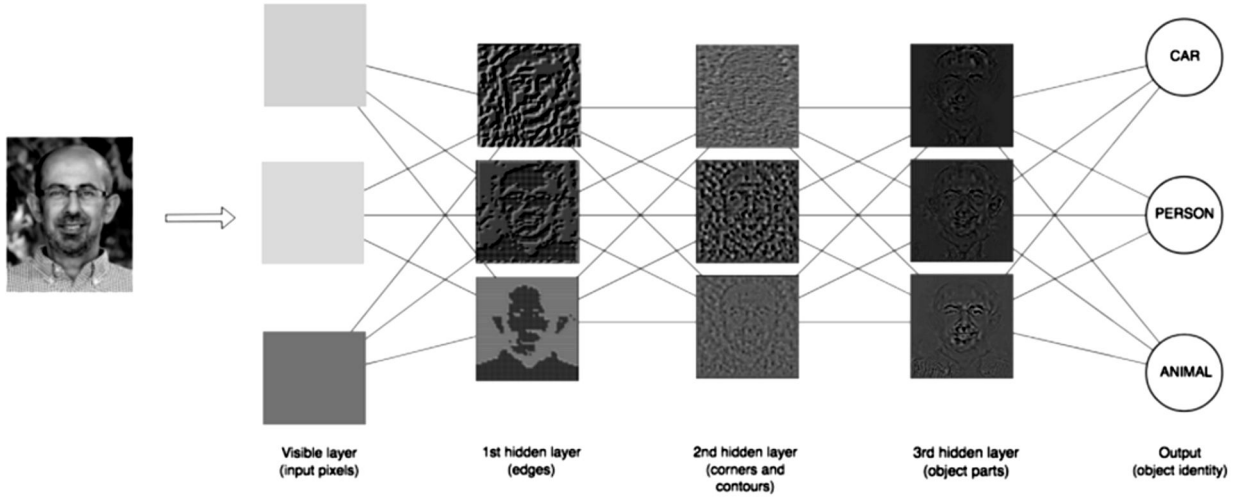
Prior to neural networks, rule-based systems have gradually evolved into more modern machine learning, whereby more and more abstract features can be learned. This means that much more complex selection criteria are now possible.

To understand this idea, imagine that you are trying to classify fruit based on the length and width of the fruit. It may be easy to separate if you have two very dissimilar fruit that you are comparing, such as an apple and a banana. However, this rule system breaks down in some cases due to the oversimplified features that were chosen.

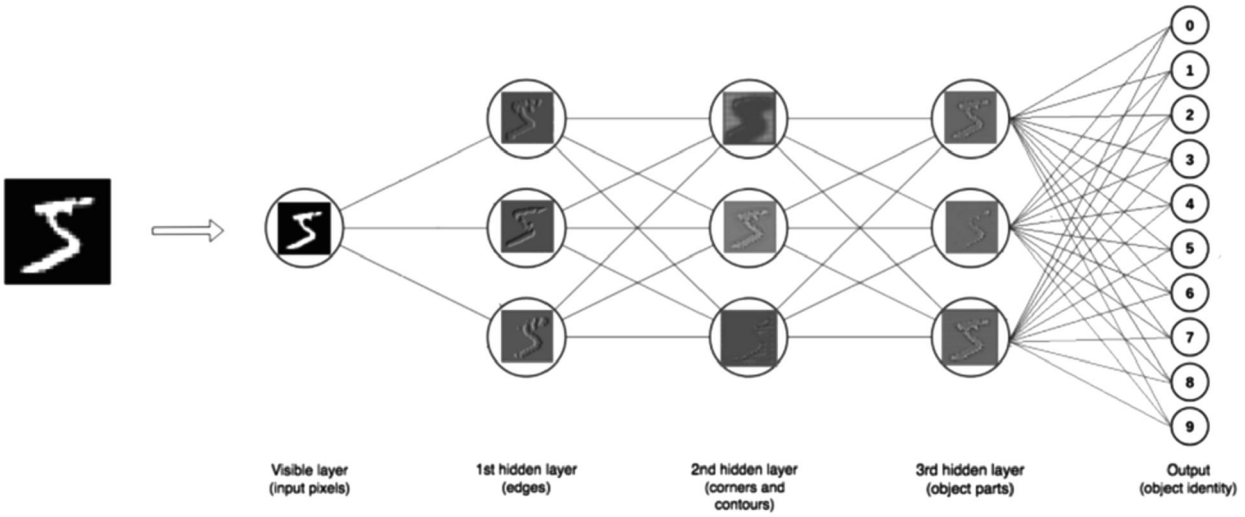
Neural networks provide an abstract representation of the data at each stage of the network which are designed to detect specific features of the network. When considering convolutional neural networks, which are used to study images, when we look at hidden layers closer to the output of a deep network, the hidden layers have highly interpretable representations, such as faces, clothing, etc. However, when we look at the first layers of the network, they are detecting very basic features such as corners, curves, and so on.



These abstract representations quickly become too complex to comprehend, and to this day the workings of neural networks to produce highly complex abstractions are still seen as somewhat magical and is a topic of research in the deep learning community.



An example of a neural network with multiple hidden layers classifying an image of a human face.



An example of a neural network with multiple hidden layers classifying written digits from the MNIST dataset.

3.5.2 Layers

Q6. Discuss about Layers in Neural Networks.

Ans : **(Imp.)**

The fundamental data structure in neural networks is the layer, to which you were introduced a layer is a data-processing module that takes as input one or more tensors and that outputs one or more tensors. Some layers are stateless, but more frequently layers have a state: the layer's weights, one or several tensors learned with stochastic gradient descent, which together contain the network's knowledge.

Different layers are appropriate for different tensor formats and different types of data processing. For instance, simple vector data, stored in 2D tensors of shape (samples, features), is often processed by densely connected layers, also called fully connected or dense layers (the Dense class in Keras). Sequence data, stored in 3D tensors of shape (samples, timesteps, features), is typically processed by recurrent layers such as an LSTM layer. Image data, stored in 4D tensors, is usually processed by 2D convolution layers (Conv2D).

We can think of layers as the LEGO bricks of deep learning, a metaphor that is made explicit by frameworks like Keras. Building deep-learning models in Keras is done by clipping together compatible layers to form useful data-transformation pipelines. The notion of layer compatibility here refers specifically to the fact that every layer will only accept input tensors of a certain shape and will return output tensors of a certain shape.

Consider the following example:

```
from keras import layers
layer = layers.Dense(32, input_shape=(784,))
```

We're creating a layer that will only accept as input 2D tensors where the first dimension is 784 (axis 0, the batch dimension, is unspecified, and thus any value would be accepted). This layer will return a tensor where the first dimension has been transformed to be 32.

Thus, this layer can only be connected to a downstream layer that expects 32-dimensional vectors as its input. When using Keras, you don't have to worry about compatibility, because the layers you add to your models are dynamically built to match the shape of the incoming layer. For instance, suppose you write the following:

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(layers.Dense(32, input_shape=(784,)))
model.add(layers.Dense(32))
```

The second layer didn't receive an input shape argument—instead, it automatically inferred its input shape as being the output shape of the layer that came before.

3.5.3 Models

Q7. Discuss about Neural Network Models.

Ans : **(Imp.)**

The Neural Networks Model

Neural networks are simple models of the way the nervous system operates. The basic units are neurons, which are typically organized into layers, as shown in the following figure.

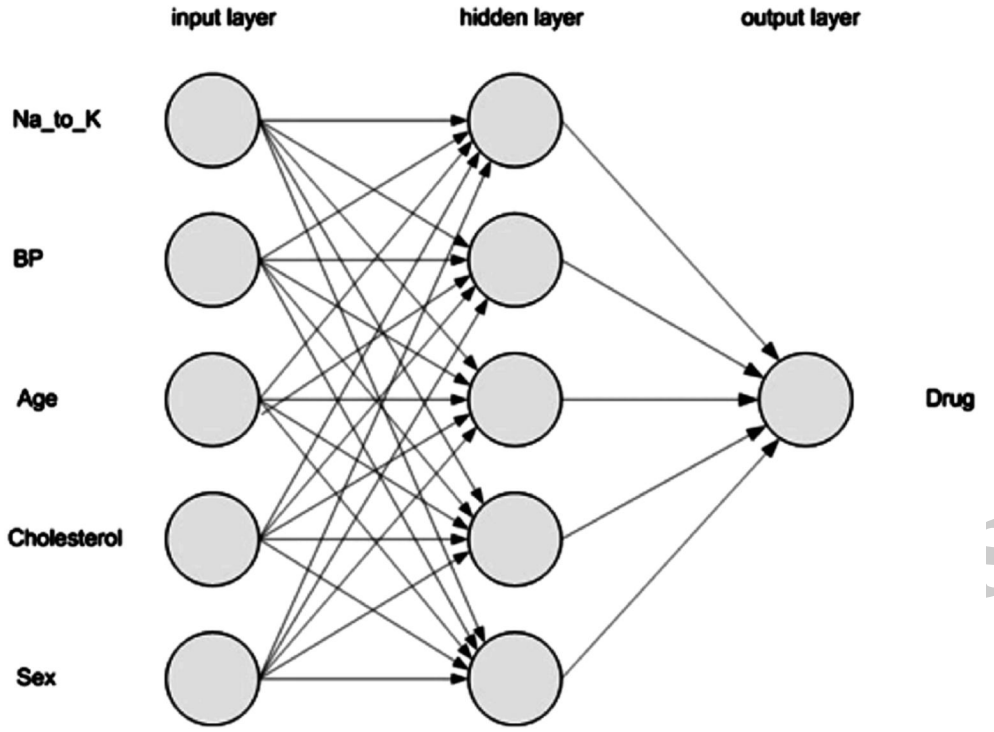


Fig.: Structure of a Neural Network

A neural network is a simplified model of the way the human brain processes information. It works by simulating a large number of interconnected processing units that resemble abstract versions of neurons.

The processing units are arranged in layers. There are typically three parts in a neural network: an input layer, with units representing the input fields; one or more hidden layers; and an output layer, with a unit or units representing the target field(s). The units are connected with are propagated from each neuron to every neuron in the next layer. Eventually, a result is delivered from the output layer.

The network learns by examining individual records, generating a prediction for each record, and making adjustments to the weights whenever it makes an incorrect prediction. This process is repeated many times, and the network continues to improve its predictions until one or more of the stopping criteria have been met.

Initially, all weights are random, and the answers that come out of the net are probably nonsensical. The network learns through training. Examples for which the output is known are repeatedly presented to the network, and the answers it gives are compared to the known outcomes. Information from this comparison is passed back through the network, gradually changing the weights. As training progresses, the network becomes increasingly accurate in replicating the known outcomes. Once trained, the network can be applied to future cases where the outcome is unknown.

Models: networks of layers

A deep-learning model is a directed, acyclic graph of layers. The most common instance is a linear stack of layers, mapping a single input to a single output.

But as you move forward, you'll be exposed to a much broader variety of network topologies. Some common ones include the following:

- Two-branch networks
- Multihead networks
- Inception blocks

The topology of a network defines a hypothesis space we defined machine learning as "searching for useful representations of some input data, within a predefined space of possibilities, using guidance from a feedback signal." By choosing a network topology, you constrain your space of possibilities (hypothesis space) to a specific series of tensor operations, mapping input data to output data. What you'll then be searching for is a good set of values for the weight tensors involved in these tensor operations.

Picking the right network architecture is more an art than a science; and although there are some best practices and principles you can rely on, only practice can help you become a proper neural-network architect. The next few chapters will both teach you explicit principles for building neural networks and help you develop intuition as to what works or doesn't work for specific problems.

3.5.4 Loss functions and optimizers

Q8. Explain in detail about Loss functions and Optimizers.

Ans : **(Imp.)**

Loss functions and optimizers keys to configuring the learning process.

Once the network architecture is defined, you still have to choose two more things:

- **Loss function (objective function):** The quantity that will be minimized during training. It represents a measure of success for the task at hand.
- **Optimizer :** Determines how the network will be updated based on the loss function. It implements a specific variant of stochastic gradient descent (SGD).

A neural network that has multiple outputs may have multiple loss functions (one per output). But the gradient-descent process must be based on a single scalar loss value; so, for multiloss networks, all losses are combined (via averaging) into a single scalar quantity.

Choosing the right objective function for the right problem is extremely important: your network will take any shortcut it can, to minimize the loss; so if the objective doesn't fully correlate with success for the task at hand, your network will end up doing things you may not have wanted. Imagine a stupid, omnipotent AI trained via SGD, with this poorly chosen objective function: "maximizing the average well-being of all humans alive." To make its job easier, this AI might choose to kill all humans except a few and focus on the well-being of the remaining ones because average well-being isn't affected by how many humans are left. That might not be what you intended! Just remember that all neural networks you build will be just as ruthless in lowering their loss function, so choose the objective wisely, or you'll have to face unintended side effects.

Fortunately, when it comes to common problems such as classification, regression, and sequence prediction, there are simple guidelines you can follow to choose the correct loss. For instance, you'll use binary cross entropy for a two-class classification problem, categorical cross entropy for a many-class classification problem, mean-squared error for a regression problem, connectionist temporal classification (CTC) for a sequence-learning problem, and so on. Only when you're working on truly new research problems will you have to develop your own objective functions. In the next few chapters, we'll detail explicitly which loss functions to choose for a wide range of common tasks.

Back Propogation and Optimisation Function: Error J(w) is a function of internal parameters of model i.e weights and bias. For accurate predictions, one needs to minimize the calculated error. In a neural network, this is done using back propagation. The current error is typically propagated backwards to a previous layer, where it is used to modify the weights and bias in such a way that the error is minimized. The weights are modified using a function called Optimization Function.

Optimisation functions usually calculate the gradient i.e. the partial derivative of loss function with respect to weights, and the weights are modified in the opposite direction of the calculated gradient. This cycle is repeated until we reach the minima of loss function.

Thus, the components of a neural network model i.e the activation function, loss function and optimization algorithm play a very important role in efficiently and effectively training a Model and produce accurate results. Different tasks require a different set of such functions to give the most optimum results.

Loss Functions

Thus, loss functions are helpful to train a neural network. Given an input and a target, they calculate the loss, i.e difference between output and target variable. Loss functions fall under four major category:

Regressive loss functions

They are used in case of regressive problems, that is when the target variable is continuous. Most widely used regressive loss function is Mean Square Error. Other loss functions are:

1. **Absolute error :** measures the mean absolute value of the element-wise difference between input.
2. **Smooth Absolute Error :** a smooth version of Abs Criterion.

Classification loss functions

The output variable in classification problem is usually a probability value f(x), called the score for the input x. Generally, the magnitude of the score represents the confidence of our prediction. The target variable y, is a binary variable, 1 for true and -1 for false.

On an example (x,y), the margin is defined as yf(x). The margin is a measure of how correct we are. Most classification losses mainly aim to maximize the margin. Some classification algorithms are :

1. Binary Cross Entropy
2. Negative Log Likelihood
3. Margin Classifier
4. Soft Margin Classifier

Embedding loss functions

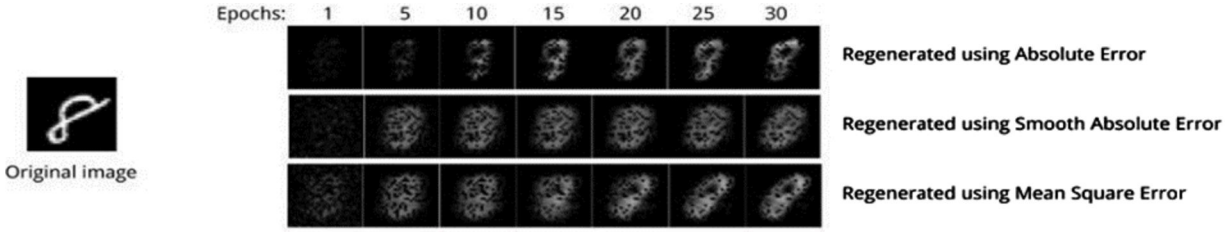
It deals with problems where we have to measure whether two inputs are similar or dissimilar. Some examples are:

1. **L1 Hinge Error :** Calculates the L1 distance between two inputs.
2. **Cosine Error :** Cosine distance between two inputs.

Visualising Loss Functions

We performed the task to reconstruct an image using a type of neural network called Autoencoders. Different results were obtained for the same task by using different Loss Functions, while everything else in the neural network architecture remained constant. Thus, the difference in result represents the properties of the different loss functions employed. A very simple data set, MNIST data set was used for this purpose. Three loss functions were used to reconstruct images.

- Absolute Loss Function
- Mean Square Loss Function
- Smooth Absolute Loss Function



Loss function for Mean Square Error $loss(x, y) = \frac{1}{n} \sum | x_i - y_i |^2$

Loss function for Absolute Error $loss(x, y) = \frac{1}{n} \sum | x_i - y_i |$

Loss function for Smooth Absolute Error $loss(x, y) = \frac{1}{n} \sum \begin{cases} 0.5 * (x_i - y_i)^2, \text{ if } | x_i - y_i | < 1 \\ | x_i - y_i | - 0.5, \text{stotherwise} \end{cases}$

While the Absolute error just calculated the mean absolute value between of the pixel-wise difference, Mean Square error uses mean squared error. Thus it was more sensitive to outliers and pushed pixel value towards 1 (in our case, white as can be seen in image after first epoch itself).

Smooth L1 error can be thought of as a smooth version of the Absolute error. It uses a squared term if the squared element-wise error falls below 1 and L1 distance otherwise. It is less sensitive to outliers than the Mean Squared Error and, in some cases, prevents exploding gradients.

Short Question and Answers

1. Discuss about Gradient-based optimization?

Ans :

Gradient descent is an optimization algorithm that's used when training deep learning models. It's based on a convex function and updates its parameters iteratively to minimize a given function to its local minimum.

Gradient Descent

$$\Theta_j = \Theta_j - \alpha \frac{\partial}{\partial \Theta_j} J(\Theta_0, \Theta_1)$$

Learning Rate

The notation used in the above Formula is given below,

- ± is the learning rate,
- J is the cost function, and
- ô is the parameter to be updated.

The gradient represents the partial derivative of J (cost function) with respect to $\hat{o}_j$

Note that, as we reach closer to the global minima, the slope or the gradient of the curve becomes less and less steep, which results in a smaller valueof derivative, which in turn reduces the step size or learning rate automatically.

It is the most basic but most used optimizer that directly uses the derivative of the loss function and learning rate to reduce the loss function and tries to reach the global minimum.

The Gradient Descent Optimization algorithm has many applications including:

- Linear Regression,
- Classification Algorithms,

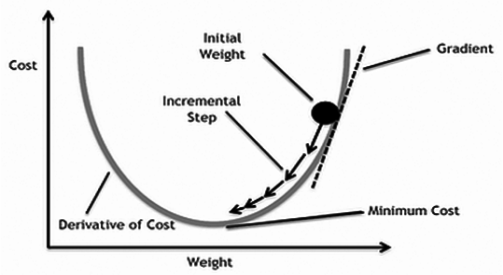
Backpropagation in Neural Networks, etc.

Repeat until convergence {

$$\theta_j \leftarrow Q_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

}

The above-described equation calculates the gradient of the cost function J(¸) with respect to the network parameters ¸ for the entire training dataset:



2. What Is the Difference Between Batch Gradient Descent and Stochastic Gradient Descent?

Ans :

Batch Gradient Descent	Stochastic Gradient Descent
The batch gradient computes the gradient using the entire dataset. It takes time to converge because the volume of data is huge, and weights update slowly.	The stochastic gradient computes the gradient using a single sample. It converges much faster than the batch gradient because it updates weight more frequently

3. What Do You Understand by Backpropagation?

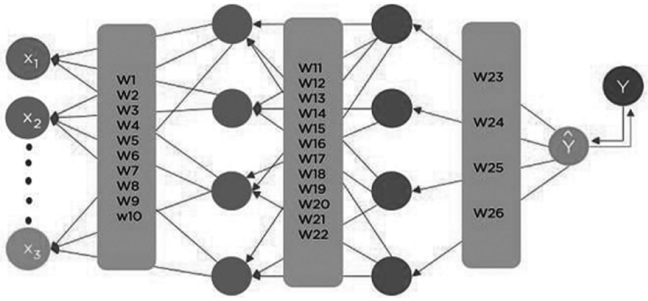
Ans :

Backpropagation is a technique to improve the performance of the network. It backpropagates the error and updates the weights to reduce the error.

A training algorithm which is used for a multilayer neural network is known as backpropagation. In backpropagation, the error is moved from the end of the network to all weights, thus allowing efficient computing of the gradient. It is typically divided into several steps, such as-

- Forward propagation of the training data so that the output is generated.
- By using the target value and the output value, error derivative can be computed. (with respect to output activation)
- We then propagate for computing the derivative of the error (with respect to output activation) and continue to do so for all of the hidden layers.
- By using the previously calculated derivatives, we can calculate error derivatives with respect to the weights.
- Update the weights.

- Neural Network technique to minimize the cost function
- Helps to improve the performance of the network
- Backpropagates the error and updates the weights to reduce the error



4. What is a Neural Network?

Ans :

Neural Networks replicate the way humans learn, inspired by how the neurons in our brains fire, only much simpler.

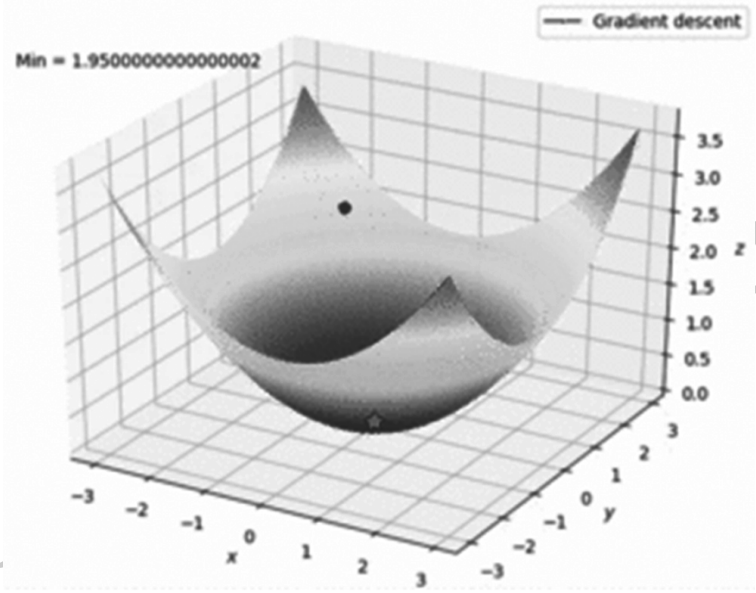
- **Neural Network:** The most common Neural Networks consist of three network layers:

- **An input layer:** A hidden layer (this is the most important layer where feature extraction takes place, and adjustments are made to train faster and function better)
- **An output layer:** Each sheet contains neurons called "nodes," performing various operations. Neural Networks are used in deep learning algorithms like CNN, RNN, GAN, etc.

5. What is gradient descent?What are the steps for using a gradient descent algorithm?

Ans :

Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient.



- **Stochastic Gradient Descent:** Uses only a single training example to calculate the gradient and update parameters.
- **Batch Gradient Descent:** Calculate the gradients for the whole dataset and perform just one update at each iteration.
- **Mini-batch Gradient Descent:** Mini-batch gradient is a variation of stochastic gradient descent where instead of single training example, mini-batch of samples is used. It's one of the most popular optimization algorithms.

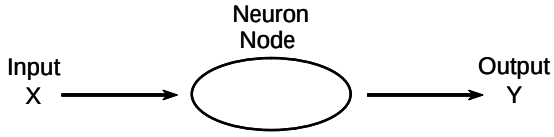
The steps involved in using a gradient descent algorithm are as follows-

- -Initialize a random weight and bias
- -Pass an input through the network and get the value from the output layer
- -Calculate if there is an error between the actual value and the predicted value
- -Go to each neuron which is contributing to the error and change its respective value so that the error is reduced
- -Reiterate the steps until the best weights of the network are found.

6. Write a note on Anatomy of a neural network.

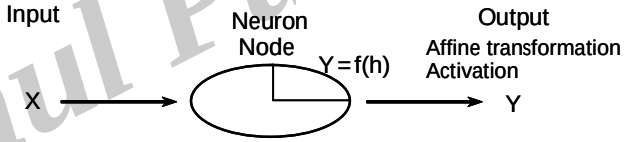
Ans :

Artificial neural networks are one of the main tools used in machine learning. As the "neural" part of their name suggests, they are brain-inspired systems which are intended to replicate the way that we humans learn. Neural networks consist of input and output layers, as well as (in most cases) a hidden layer consisting of units that transform the input into something that the output layer can use. They are excellent tools for finding patterns which are far too complex or numerous for a human programmer to extract and teach the machine to recognize.



While vanilla neural networks (also called "perceptrons") have been around since the 1940s, it is only in the last several decades where they have become a major part of artificial intelligence. This is due to the arrival of a technique called backpropagation (which we discussed in the previous tutorial), which allows networks to adjust their neuron weights in situations where the outcome doesn't match what the creator is hoping for — like a network designed to recognize dogs, which misidentifies a cat, for example.

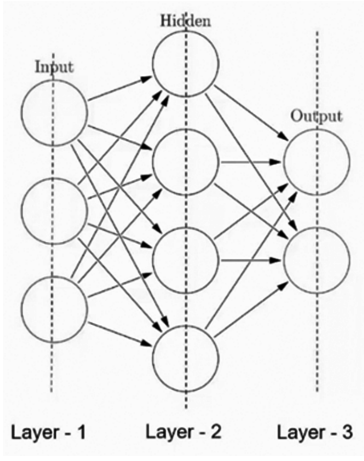
The fact that neural networks make use of affine transformations in order to concatenate input features together that converge at a specific node in the network. This concatenated input is then passed through an activation function, which evaluates the signal response and determines whether the neuron should be activated given the current inputs.as this can be an important factor in obtaining a functional network. So far we have only talked about sigmoid as an activation function but there are several other choices, and this is still an active area of research in the machine learning literature.



7. What are Layers in a Neural Network?

Ans :

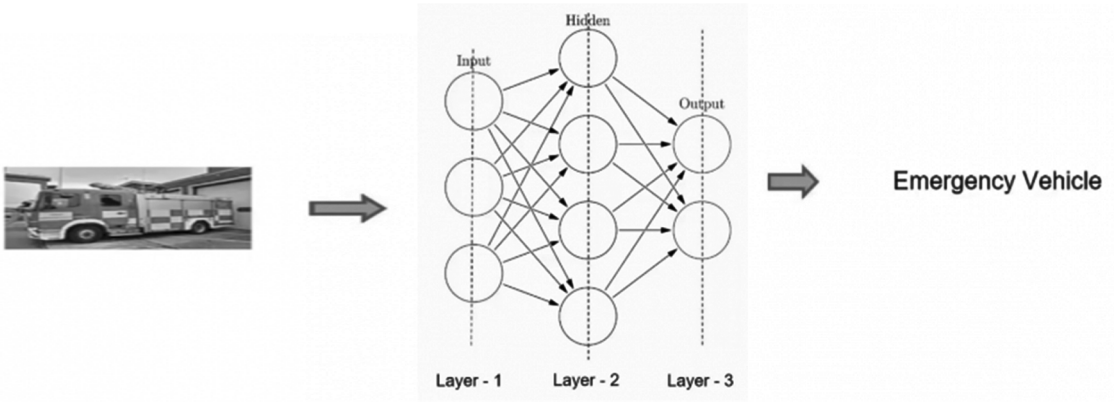
A neural network is made up of vertically stacked components called Layers. Each dotted line in the image represents a layer. There are three types of layers in a Neural Network.



Input Layer: First is the input layer. This layer will accept the data and pass it to the rest of the network.

Hidden Layer: The second type of layer is called the hidden layer. Hidden layers are either one or more in number for a neural network. In the above case, the number is 1. Hidden layers are the ones that are actually responsible for the excellent performance and complexity of neural networks. They perform multiple functions at the same time such as data transformation, automatic feature creation, etc.

Output Layer: The last type of layer is the output layer. The output layer holds the result or the output of the problem. Raw images get passed to the input layer and we receive output in the output layer.



In this case, we are providing an image of a vehicle and this output layer will provide an output whether it is an emergency or non-emergency vehicle, after passing through the input,hidden layers and finally output layers.

8. List out neural network models?

Ans :

There are many different types of artificial neural networks, varying in complexity. They share the intended goal of mirroring the function of the human brain to solve complex problems or tasks. The structure of each type of artificial neural network in some way mirrors neurons and synapses. However, they differ in terms of complexity, use cases, and structure. Differences also include how artificial neurons are modelled within each type of artificial neural network, and the connections between each node. Other differences include how the data may flow through the artificial neural network, and the density of the nodes.

5 Examples of the different types of artificial neural network include:

- Feedforward artificial neural networks
- Perceptron and Multilayer Perceptron neural networks
- Radial basis function artificial neural networks
- Recurrent neural networks
- Modular neural networks

9. What is loss function? and list out them.

Ans :

A loss function is a function that compares the target and predicted output values; measures how well the neural network models the training data. When training, we aim to minimize this loss between the predicted and target outputs**.**

Types of Loss Functions

In supervised learning, there are two main types of loss functions — these correlate to the 2 major types of neural networks: regression and classification loss functions

1. **Regression Loss Functions —** used in regression neural networks; given an input value, the model predicts a corresponding output value (rather than pre-selected labels).

   Ex. Mean Squared Error, Mean Absolute Error

2. **Classification Loss Functions —** used in classification neural networks; given an input, the neural network produces a vector of probabilities of the input belonging to various pre-set categories — can then select the category with the highest probability of belonging.

   Ex. Binary Cross-Entropy, Categorical Cross-Entropy

10. How can hyperparameters be trained in neural networks?

Ans :

Hyperparameters can be trained using four components as shown below:

- **Batch Size:** This is used to denote the size of the input chunk. Batch sizes can be varied and cut into sub-batches based on the requirement.
- **Epochs:** An epoch denotes the number of times the training data is visible to the neural network so that it can train. Since the process is iterative, the number of epochs will vary based on the data.
- **Momentum:** Momentum is used to understand the next consecutive steps that occur with the current data being executed at hand. It is used to avoid oscillations when training.
- **Learning Rate:** Learning rate is used as a parameter to denote the time required for the network to update the parameters and learn.

Choose the Correct Answer

1. Different learning method does not include: [d]
   (a) Memorization (b) Analogy
   (c) Deduction (d) Introduction
2. Which of the following is the model used for learning? [d]
   (a) Decision trees (b) Neural networks
   (c) Propositional and FOL rules (d) All
3. Automated vehicle is an example of _____. [a]
   (a) Supervised learning (b) Unsupervised learning
   (c) Active learning (d) Reinforcement learning
4. Following is an example of active learning: [a]
   (a) News recommendation system (b) Dust cleaning machine
   (c) Automated vehicle (d) None of them
5. Which of the following is not an application of learning? [d]
   (a) Data mining (b) WWW
   (c) Speech recognition (d) None of them
6. Which of the following is the component of learning system? [d]
   (a) Goal (b) Model
   (c) Learning rules (d) All of them
7. Following is also called as exploratory learning: [c]
   (a) Supervised learning (b) Active learning
   (c) Unsupervised learning (d) Reinforcement learning
8. How many types of learning are available in machine learning? [c]
   (a) 1 (b) 2
   (c) 3 (d) 4
9. Artificial neural network is used for [d]
   (a) Classification (b) Clustering
   (c) Pattern recognition (d) All of them
10. A Neural Network can answer [b]
   (a) For Loop questions (b) what-if questions
   (c) IF-The-Else Analysis Questions (d) None of the mentioned

Fill in the blanks

1. _____ computes the output volume by computing dot product between all filters and image patch
2. _____ is/are the ways to represent uncertainty
3. The purpose of Axon is _____.
4. When the cell is said to be fired _____?
5. What is the objective of a pattern storage task is _____.
6. The BAM is a _____ associative pattern-marching network that encodes binary or bipolar patterns using Hebbian learning rule
7. CAM stands for _____.
8. In the _____ associative memory network, the training input vector and training output vector are the same.
9. The _____ is used to control the amount of weight adjustment at each step of training.
10. Madaline stands for _____.

ANSWERS

1. Convolution Layer
2. Fuzzy logic, Entropy and Probability
3. transmission
4. if potential of body reaches a steady threshold values
5. to store and recall
6. Recurrent hetero
7. Content Addressable Memories
8. auto
9. learning rate
10. Multiple Adaptive Linear Neuron

One Mark Answers

1. What are activation functions?

Ans :

Activation functions are entities in Deep Learning that are used to translate inputs into a usable output parameter.

2. What are activation functions?

Ans :

Activation functions are entities in Deep Learning that are used to translate inputs into a usable output parameter.

3. Why is Fourier transform used in Deep Learning?

Ans :

Fourier transform is an effective package used for analyzing and managing large amounts of data present in a database.

4. What are the elements in TensorFlow that are programmable?

Ans :

In TensorFlow, users can program three elements:

- Constants
- Variables
- Placeholders

5. What is CNTK used for?

Ans :

The Microsoft Cognitive Toolkit (CNTK) is a powerful, open source library that can be used to create machine learning prediction models.

6. What are the main difficulties when training RNNs

Ans :

The two main difficulties when training RNNs are unstable gradients (exploding or vanishing) and a very limited short-term memory. These problems both get worse when dealing with long sequences.

7. What's the difference between Traditional Feedforward Networks and Recurrent Neural Networks?

Ans :

Traditional feedforward neural networks take in a fixed amount of input data all at the same time and produce a fixed amount of output each time.

Recurrent neural networks do not consume all the input data at once. Instead, they take them in one at a time and in a sequence.

8. What is the basic concept of Recurrent Neural Network?

Ans :

Use previous inputs to find the next output according to the training set.

9. For what RNN is used and achieve the best results?

Ans :

Handwriting and speech recognition.

10. One of the RNN´s issue is 'Exploding Gradients'. What is that?

Ans :

When the algorithm assigns a stupidly high importance to the weights, without much reason.

UNIT IV

Introduction to Keras, Keras, TensorFlow, Theano, and CNTK Recurrent neural networks: A recurrent layer in Keras, Understanding the LSTM and GRU layers

4.1 Introduction to Keras

Q1. Define Keras. Discuss its featurs and benefits?

Ans :

Deep learning is a branch of artificial intelligence concerned with solving highly complex problems by emulating the working of the human brain. In deep learning, we use neural networks which use multiple operators placed in nodes to help break down the problem into smaller parts, which are each solved individually. But neural networks can be really hard to implement. This problem is taken care of by Keras, a deep learning framework.

Meaning

Keras is a high-level, deep learning API developed by Google for implementing neural networks. It is written in Python and is used to make the implementation of neural networks easy. It also supports multiple backend neural network computation.

Keras is relatively easy to learn and work with because it provides a python frontend with a high level of abstraction while having the option of multiple back-ends for computation purposes. This makes Keras slower than other deep learning frameworks, but extremely beginner-friendly.

Keras allows you to switch between different back ends. The frameworks supported by Keras are:

- Tensorflow
- Theano
- PlaidML
- MXNet
- CNTK (Microsoft Cognitive Toolkit)

Out of these five frameworks, TensorFlow has adopted Keras as its official high-level API. Keras is embedded in TensorFlow and can be used to perform deep learning fast as it provides inbuilt modules for all neural network computations. At the same time, computation involving tensors, computation graphs, sessions, etc can be custom made using the TensorFlow Core API, which gives you total flexibility and control over your application and lets you implement your ideas in a relatively short time.

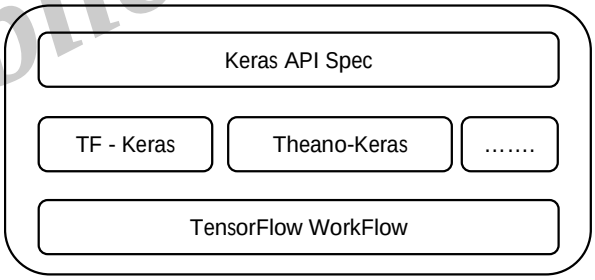


Fig.: Keras Backend

Features

Keras leverages various optimization techniques to make high level neural network API easier and more performant. It supports the following features.

- Consistent, simple and extensible API.
- Minimal structure - easy to achieve the result without any frills.
- It supports multiple platforms and backends.
- It is user friendly framework which runs on both CPU and GPU.
- Highly scalability of computation.

Benefits

Keras is highly powerful and dynamic framework and comes up with the following advantages

- Larger community support.
- Easy to test.
- Keras neural networks are written in Python which makes things simpler.
- Keras supports both convolution and recurrent networks.
- Deep learning models are discrete components, so that, you can combine into many ways.

Q2. Explain about keras applications and its building model?

Ans :

Keras runs on top of open source machine libraries like TensorFlow, Theano or Cognitive Toolkit (CNTK). Theano is a python library used for fast numerical computation tasks. TensorFlow is the most famous symbolic math library used for creating neural networks and deep learning models. TensorFlow is very flexible and the primary benefit is distributed computing. CNTK is deep learning framework developed by Microsoft. It uses libraries such as Python, C#, C++ or standalone machine learning toolkits. Theano and TensorFlow are very powerful libraries but difficult to understand for creating neural networks.

Keras is based on minimal structure that provides a clean and easy way to create deep learning models based on TensorFlow or Theano. Keras is designed to quickly define deep learning models. Well, Keras is an optimal choice for deep learning applications.

Keras is a deep-learning framework for Python that provides a convenient way to define and train almost any kind of deep-learning model. Keras was initially developed for researchers, with the aim of enabling fast experimentation.

Keras has the following key features

- It allows the same code to run seamlessly on CPU or GPU.
- It has a user-friendly API that makes it easy to quickly prototype deep-learning models.
- It has built-in support for convolutional networks (for computer vision), recurrent networks (for sequence processing), and any combination of both.
- It supports arbitrary network architectures: multi-input or multi-output models, layer sharing, model sharing, and so on. This means Keras is appropriate for building essentially any deep-learning model, from a generative adversarial network to a neural Turing machine.
- Keras is distributed under the permissive MIT license, which means it can be freely used in commercial projects. It's compatible with any version of Python from 2.7 to 3.6.

Keras has well over 200,000 users, ranging from academic researchers and engineers at both startups and large companies to graduate students and hobbyists. Keras is used at Google, Netflix, Uber, CERN, Yelp, Square, and hundreds of startups working on a wide range of problems. Keras is also a popular framework on Kaggle, the machine-learning competition website, where almost every recent deep-learning competition has been won using Keras models.

Need of Keras

- Keras is an API that was made to be easy to learn for people. Keras was made to be simple. It offers consistent & simple APIs, reduces the actions required to implement common code, and explains user error clearly.
- Prototyping time in Keras is less. This means that your ideas can be implemented and deployed in a shorter time. Keras also provides a variety of deployment options depending on user needs.
- Languages with a high level of abstraction and inbuilt features are slow and building custom features in then can be hard. But Keras runs

on top of TensorFlow and is relatively fast. Keras is also deeply integrated with TensorFlow, so you can create customized workflows with ease.

- The research community for Keras is vast and highly developed. The documentation and help available are far more extensive than other deep learning frameworks.
- Keras is used commercially by many companies like Netflix, Uber, Square, Yelp, etc which have deployed products in the public domain which are built using Keras.

Apart from this, Keras has features such as :

- It runs smoothly on both CPU and GPU.
- It supports almost all neural network models.
- It is modular in nature, which makes it expressive, flexible, and apt for innovative research.

Building Model of aKeras

The below diagram shows the basic steps involved in building a model in Keras:

Fig.: Building a model

1. **Define a network:** In this step, you define the different layers in our model and the connections between them. Keras has two main types of models: Sequential and Functional models. You choose which type of model you want and then define the dataflow between them.
2. **Compile a network:** To compile code means to convert it in a form suitable for the machine to understand. In Keras, the model.compile() method performs this function. To compile the model, we define the loss function which calculates the losses in our model, the optimizer which reduces the loss, and the metrics which is used to find the accuracy of our model.
3. **Fit the network:** Using this, we fit our model to our data after compiling. This is used to train the model on our data.
4. **Evaluate the network:** After fitting our model, we need to evaluate the error in our model.
5. **Make Predictions:** We use model.predict() to make predictions using our model on new data.

Applications

- Keras is used for creating deep models which can be productized on smartphones.
- Keras is also used for distributed training of deep learning models.
- Keras is used by companies such as Netflix, Yelp, Uber, etc.
- Keras is also extensively used in deep learning competitions to create and deploy working models, which are fast in a short amount of time.

4.2 TensorFlow

Q3. Explain in detail about Tensor Flow and its use cases?

(OR)

Explain in detail about components and its applications?

Ans:

TensorFlow is a popular framework of machine learning and deep learning. It is a free and open-source library which is released on 9 November 2015 and developed by Google Brain Team. It is entirely based on Python programming language and use for numerical computation and data flow, which makes machine learning faster and easier.

TensorFlow can train and run the deep neural networks for image recognition, handwritten digit classification, recurrent neural network, word embedding, natural language processing, video detection, and many more. TensorFlow is run on multiple CPUs or GPUs and also mobile operating systems.

The word TensorFlow is made by two words, i.e., Tensor and Flow:

1. Tensor is a multidimensional array
2. Flow is used to define the flow of data in operation.

TensorFlow is used to define the flow of data in operation on a multidimensional array or Tensor.

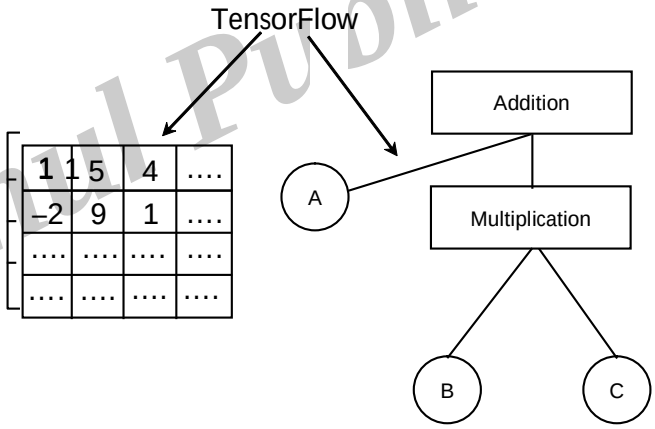


Fig.: Tensor Flow

History of TensorFlow

Many years ago, deep learning started to exceed all other machine learning algorithms when giving extensive data. Google has seen it could use these deep neural networks to upgrade its services:

- Google search engine
- Gmail
- Photo

They build a framework called TensorFlow to permit researchers and developers to work together in an AI model. Once it approved and scaled, it allows lots of people to use it.

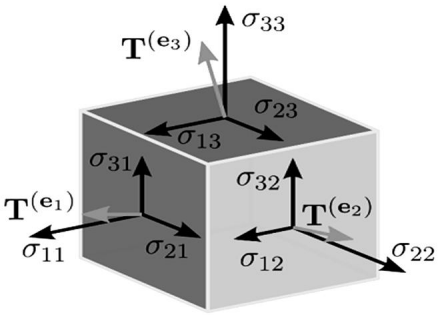
It was first released in 2015, while the first stable version was coming in 2017. It is an open- source platform under Apache Open Source License. We can use it, modify it, and reorganize the revised version for free without paying anything to Google.

Components of TensorFlow

i) Tensor

The name TensorFlow is derived from its core framework, "**Tensor**." A tensor is a vector or a matrix of n-dimensional that represents all type of data. All values in a tensor hold similar data type with a known shape. The shape of the data is the dimension of the matrix or an array.

A tensor can be generated from the input data or the result of a computation. In TensorFlow, all operations are conducted inside a graph. The group is a set of calculation that takes place successively. Each transaction is called an op node are connected.



ii) II Graphs

TensorFlow makes use of a graph framework. The chart gathers and describes all the computations done during the training.

Advantages

- It was fixed to run on multiple CPUs or GPUs and mobile operating systems.
- The portability of the graph allows to conserve the computations for current or later use. The graph can be saved because it can be executed in the future.
- All the computation in the graph is done by connecting tensors together.

  Consider the following expression a = (b+c)*(c+2)

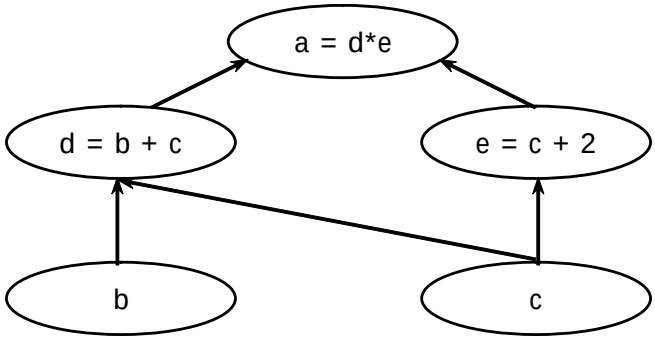
  We can break the functions into components given below:

  d = b+c

  e = c+2

  a = d*e

  Now, we can represent these operations graphically below:



iii) Session

A session can execute the operation from the graph. To feed the graph with the value of a tensor, we need to open a session. Inside a session, we must run an operator to create an output.

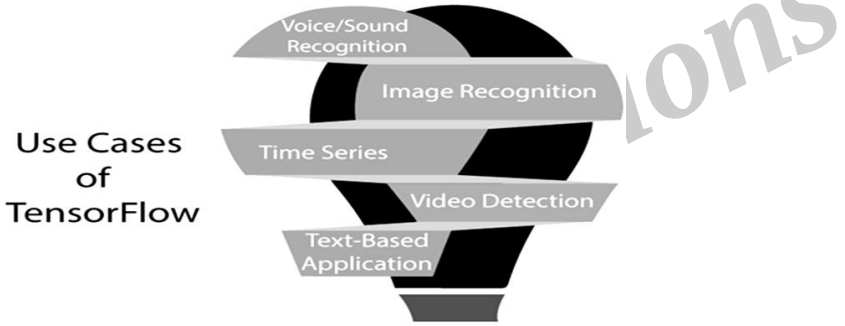
Popular of TensorFlow

TensorFlow is the better library for all because it is accessible to everyone. TensorFlow library integrates different API to create a scale deep learning architecture like CNN (Convolutional Neural Network) or RNN (Recurrent Neural Network).

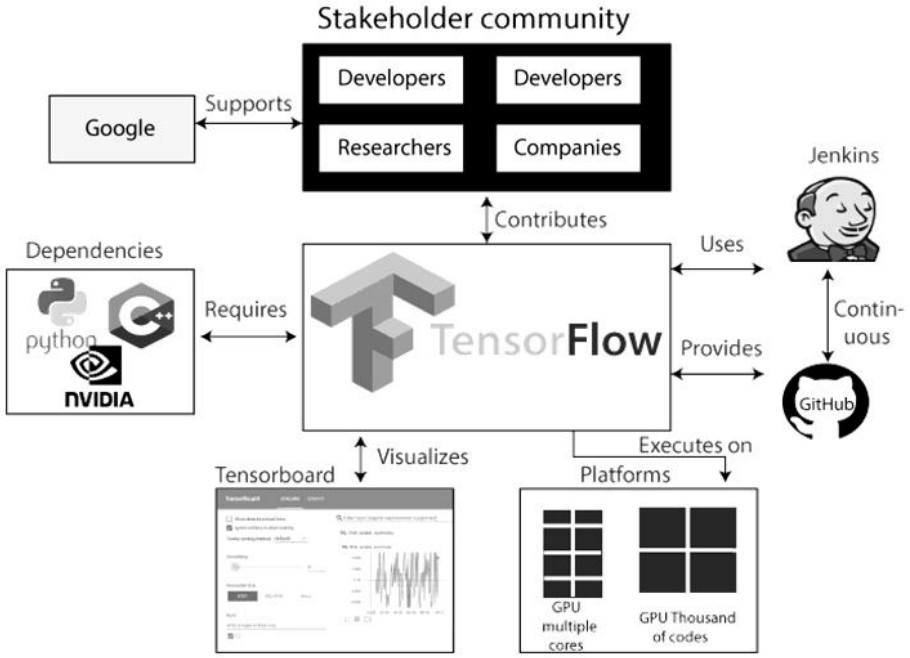
TensorFlow is based on graph computation; it can allow the developer to create the construction of the neural network with Tensorboard. This tool helps debug our program. It runs on CPU (Central Processing Unit) and GPU (Graphical Processing Unit).

TensorFlow attracts the most considerable popularity on GitHub compare to the other deep learning framework.

Use Cases/Applications of TensorFlow



TensorFlow provides amazing functionalities and services when compared to other popular deep learning frameworks. TensorFlow is used to create a large-scale **neural network** with many layers.



It is mainly used for deep learning or machine learning problems such as **Classification**, **Perception**, **Understanding**, **Discovering Prediction**, and **Creation**.

1. Voice/Sound Recognition

Voice and sound recognition applications are the most-known use cases of deep-learning. If the neural networks have proper input data feed, neural networks are capable of understanding audio signals.

For example:

Voice recognition is used in the Internet of Things, automotive, security, and UX/UI.

Sentiment Analysis is mostly used in customer relationship management **(CRM)**.

Flaw Detection (engine noise) is mostly used in automotive and Aviation.

Voice search is mostly used in customer relationship management (CRM)

2. Image Recognition

Image recognition is the first application that made deep learning and machine learning popular. Telecom, Social Media, and handset manufacturers mostly use image recognition. It is also used for face recognition, image search, motion detection, machine vision, and photo clustering.

For example, image recognition is used to recognize and identify people and objects in from of images. Image recognition is used to understand the context and content of any image.

For object recognition, TensorFlow helps to classify and identify arbitrary objects within larger images.

This is also used in engineering application to identify shape for modeling purpose (**3d** reconstruction from **2d** image) and by Facebook for photo tagging.

For example, deep learning uses TensorFlow for analyzing thousands of photos of cats. So a deep learning algorithm can learn to identify a cat because this algorithm is used to find general features of objects, animals, or people.

3. Time Series

Deep learning is using Time Series algorithms for examining the time series data to extract meaningful statistics. For example, it has used the time series to predict the stock market.

A recommendation is the most common use case for Time Series. **Amazon**, **Google**, **Facebook**, and **Netflix** are using deep learning for the suggestion. So, the deep learning algorithm is used to analyze customer activity and compare it to millions of other users to determine what the customer may like to purchase or watch.

For example, it can be used to recommend us TV shows or movies that people like based on TV shows or movies we already watched.

4. Video Detection

The deep learning algorithm is used for video detection. It is used for motion detection, real-time threat detection in gaming, security, airports, and UI/UX field.

For example, NASA is developing a deep learning network for object clustering of asteroids and orbit classification. So, it can classify and predict NEOs (**Near Earth Objects**).

5. Text-Based Applications

Text-based application is also a popular deep learning algorithm. Sentimental analysis, social media, threat detection, and fraud detection, are the example of Text-based applications.

For example, Google Translate supports over 100 languages.

Some **companies** who are *currently using TensorFlow* are Google, AirBnb, eBay, Intel, DropBox, Deep Mind, Airbus, CEVA, Snapchat, SAP, Uber, Twitter, Coca-Cola, and IBM.

Q4. What are the Features of TensorFlow?

Ans :

Features of TensorFlow

TensorFlow has an interactive **multiplatform** programming interface which is scalable and reliable compared to other deep learning libraries which are available.

These features of TensorFlow will tell us about the popularity of TensorFlow.

1. Responsive Construct

We can visualize each part of the graph, which is not an option while using **Numpy** or **SciKit**. To develop a deep learning application, firstly, there are two or three components that are required to create a deep learning application and need a programming language.

2. Flexible

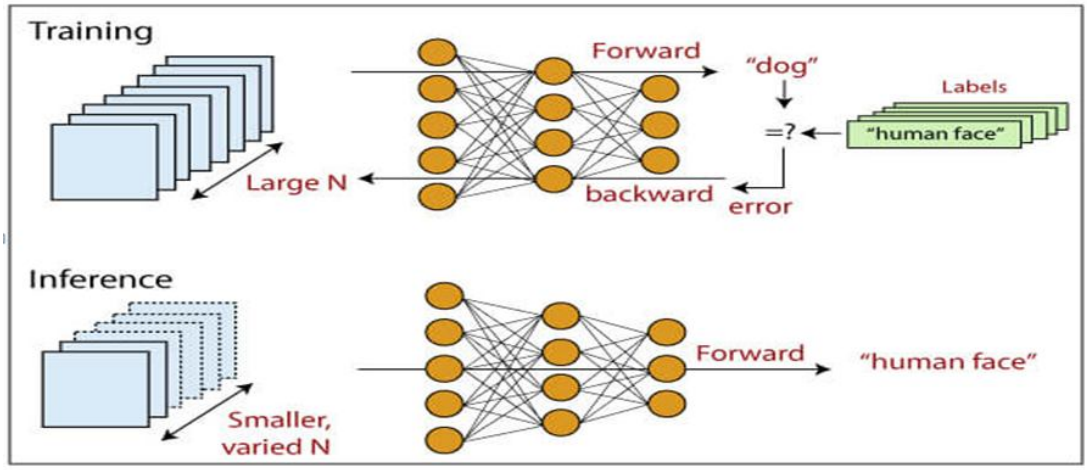
It is one of the essential TensorFlow Features according to its operability. It has modularity and parts of it which we want to make standalone.

3. Easily Trainable

It is easily trainable on CPU and for GPU in distributed computing.

4. Parallel Neural Network Training

TensorFlow offers to the pipeline in the sense that we can train multiple neural networks and various **GPUs**, which makes the models very efficient on large-scale systems.



5. Large Community

Google has developed it, and there already is a large team of software engineers who work on stability improvements continuously.

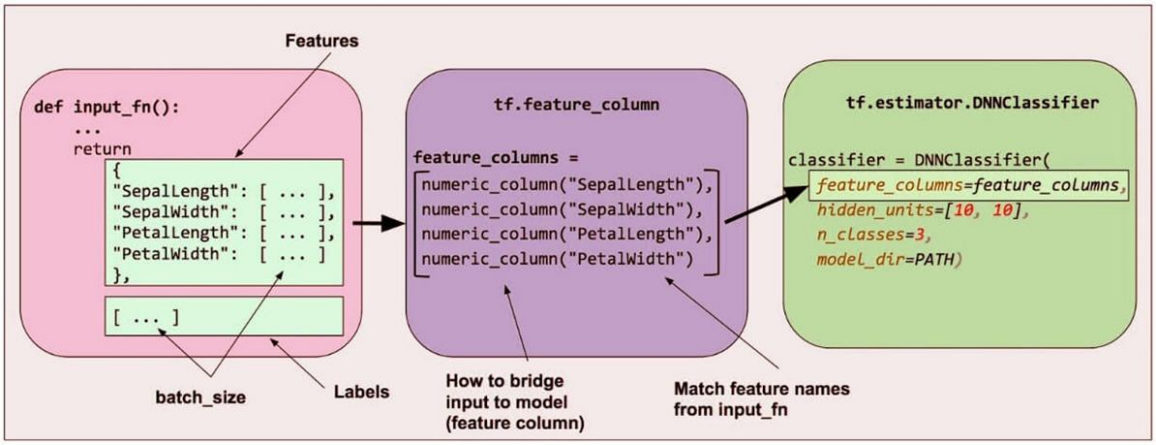
6. Open Source

The best thing about the machine learning library is that it is open source so anyone can use it as much as they have internet connectivity. So, people can manipulate the library and come up with a fantastic variety of useful products. And it has become another **DIY** community which has a massive forum for people getting started with it and those who find it hard to use it.

7. Feature Columns

TensorFlow has feature columns which could be thought of as intermediates between raw data and estimators; accordingly, **bridging** input data with our model.

The feature below describes how the feature column is implemented.



8. Availability of Statistical Distributions

This library provides distributions functions including Bernoulli, Beta, Chi2, Uniform, Gamma, which are essential, especially where considering probabilistic approaches such as Bayesian models.

9. Layered Components

TensorFlow produces layered operations of weight and biases from the function such as tf.contrib.layers and also provides batch normalization, convolution layer, and dropout layer. So tf.contrib.layers.optimizers have optimizers such as Adagrad, SGD, Momentum which are often used to solve optimization problems for numerical analysis.

10. Visualizer (With TensorBoard)

We can inspect a different representation of a model and make the changed necessary while debugging it with the help of TensorBoard.

11. Event Logger (With TensorBoard)

It is just like UNIX, where we use **tail - f** to monitor the output of tasks at the cmd. It checks, logging events and summaries from the graph and production with the TensorBoard.

4.3 THEANO

Q5. What is Theano library in deep learning with suitable examples?

Ans :

Theano is a Python library also known as the grandfather of deep learning libraries is popular among researchers. It is a compiler for manipulating and analyzing mathematical expressions, particularly matrix-valued expressions. Computations in Theano are written in a NumPy-like syntax and built to run quickly on either CPU or GPU architectures. Theano python library has garnered attention in the machine learning community as it lets you build wrapper libraries or deep learning or neural network models efficiently at high speeds.

Theano for Deep Learning

- A smart deep learning framework that lets you perform data-intensive computations faster than a CPU or a GPU with efficient native libraries like BLAS.
- Evaluating expressions is much faster because of the dynamic C code it generates.
- Creates Symbolic graphs for computing gradients automatically.
- Provides stable means to optimize and evaluate unstable expressions.

Installing of Theano using Anaconda in windows:

To install theano package with conda run:

conda install -c anaconda theano

or

conda install theano (Preferable)

To install theano package with command prompt:

pip install theano

In this way, we can install theano in windows.

Adding Two scalars using Theano

A simple function to add two scalars:

Example

```
# Import libraries
import theano
from theano import tensor
# Creating two floating-point scalars
x = tensor.dscalar()
y = tensor.dscalar()
# Creating addition expression
z = x + y
# Convert the expression into a callable object that takes (x,y) values as input and computes a value for z
fun = theano.function([x, y], z)
# Pass 11.6 to 'x', 1.1 to 'y', and evaluate 'z'
fun(11.6, 1.1)
```

Output

```
array(12.7)
```

Adding of Two Matrices using Theano:

A simple function to add two matrices:

Example

```
# Import libraries
import theano
from theano import tensor
# Creating two floating-point scalars
a = tensor.dmatrix()
b = tensor.dmatrix()
# Creating addition expression
c = a + b
# Convert the expression into a callable object that takes (a,b) values as input and computes a value for c
fun = theano.function([a, b], c)
# Calling function
fun([[1, 2], [3, 4]], [[1, 2], [3, 4]])
```

Output

```
array([[2., 4.],
       [6., 8.]])
```

To Compute Gradient in Derivatives in Theano

Let's create a function to find out the derivative of some expression y with respect to its parameter x. To do this we will use the macro tt.grad. For instance, we can compute the gradient of 2x ^ 3 with respect to x. Note that: d(2x ^ 3)/dx = 6x ^ 2.

Example

```
# Import libraries
import theano
from theano import tensor
from theano import pp
# Creating a scalar
x = tensor.dscalar('x')
# Creating 'y' expression
y = (2 * x ** 3)
# Computing derivative
derivative = tensor.grad(y, x)
# Print out the gradient
pp(derivative)
# Convert the expression into a callable object that takes 'x' as input and computes a value for 'derivative'
fun = theano.function([x], derivative)
# Calling function
fun(3)
```

Output

```
array(54.)
```

eval() Function in Theano

The eval() function is used to return the actual value of theano variable instead of its name. If we try to just print the variable x, we only print its name. But if we use eval(), we get the actual square matrix that it is initialised to.

Example

```
# Import library
import theano
# Creating a theano variable 'x' with value 10
x = theano.shared(10,'x')
#This will just print the variable x
X
```

Output

```
x
# Eval function
x.eval() #This will print its actual value
```

Output

```
array(10)
```

4.4 What is Microsoft Cognitive Toolkit (CNTK)?

Q6. Discuss in detail about CNTK?

Ans :

Microsoft Cognitive Toolkit (CNTK), formerly known as Computational Network Toolkit, is a free, easy-to-use, open-source, commercial-grade toolkit that enables us to train deep learning algorithms to learn like the human brain. It enables us to create some popular deep learning systems like feed-forward neural network time series predic-tion systems and Convolutional neural network (CNN) image classifiers.

CNTK allows the user to easily realize and combine popular model types such as feed-forward DNNs, convolutional neural networks (CNNs) and recurrent neural networks (RNNs/LSTMs).

For optimal performance, its framework functions are written in C++. Although we can call its function using C++, but the most commonly used approach for the same is to use a Python program.

Microsoft Research developed CNTK, a deep learning framework that builds a neural network as a series of computational steps via a direct graph. CNTK supports interfaces such as Python and C++ and is used for handwriting, speech recognition, and facial recognition.

CNTK's Features

Following are some of the features and capabilities offered in the latest version of Microsoft CNTK:

Built-in components

- CNTK has highly optimised built-in components that can handle multi-dimensional dense or sparse data from Python, C++ or BrainScript.
- We can implement CNN, FNN, RNN, Batch Normalisation and Sequence-to-Sequence with attention.
- It provides us the functionality to add new user-defined core-components on the GPU from Python.

- It also provides automatic hyperparameter tuning.
- We can implement Reinforcement learning, Generative Adversarial Networks (GANs), Supervised as well as Unsupervised learning.
- For massive datasets, CNTK has built-in optimised readers.

Usage of resources efficiently

- CNTK provides us parallelism with high accuracy on multiple GPUs/machines via 1-bit SGD.
- To fit the largest models in GPU memory, it provides memory sharing and other built-in methods.

Express our own networks easily

- CNTK has full APIs for defining your own network, learners, readers, training and evaluation from Python, C++, and BrainScript.
- Using CNTK, we can easily evaluate models with Python, C++, C# or BrainScript.
- It provides both high-level as well as low-level APIs.
- Based on our data, it can automatically shape the inference.
- It has fully optimised symbolic Recurrent Neural Network (RNN) loops.

Measuring model performance

- CNTK provides various components to measure the performance of neural networks you build.
- Generates log data from your model and the associated optimiser, which we can use to monitor the training process.

Version 1.0 vs Version 2.0

Following table compares CNTK Version 1.0 and 2.0:

Version 1.0	Version 2.0
It was released in 2016.	It is a significant rewrite of the 1.0 Version and was released in June 2017.
It used a proprietary scripting language called BrainScript.	Its framework functions can be called using C++, Python. We can easily load our modules in C# or Java. BrainScript is also supported by Version 2.0.
It runs on both Windows and Linux systems but not directly on Mac OS.	It also runs on both Windows (Win 8.1, Win 10, Server 2012 R2 and later) and Linux systems but not directly on Mac OS.

Important Highlights of Version 2.7

Version 2.7 is the last main released version of Microsoft Cognitive Toolkit. It has full support for ONNX 1.4.1. Following are some important highlights of this last released version of CNTK.

- Full support for ONNX 1.4.1.
- Support for CUDA 10 for both Windows and Linux systems.
- It supports advance Recurrent Neural Networks (RNN) loop in ONNX export.
- It can export more than 2GB models in ONNX format.
- It supports FP16 in BrainScript scripting language's training action.

4.5 Recurrent Neural Networks

Q7. Explain in detail about Recurrent neural networks (RNN's)? and its applications

(OR)

Explain in detail about Recurrent neural networks (RNN's)? and working of RNN's

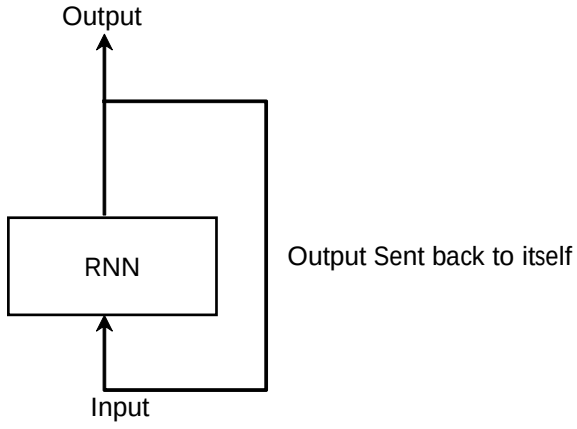
Ans : **(Imp.)**

A recurrent neural network (RNN) is a kind of artificial neural network mainly used in speech recognition and natural language processing (NLP). RNN is used in deep learning and in the development of models that imitate the activity of neurons in the human brain.

Recurrent Networks are designed to recognize patterns in sequences of data, such as text, genomes, handwriting, the spoken word, and numerical time series data emanating from sensors, stock markets, and government agencies.

A recurrent neural network looks similar to a traditional neural network except that a memory-state is added to the neurons. The computation is to include a simple memory.

The recurrent neural network is a type of deep learning-oriented algorithm, which follows a sequential approach. In neural networks, we always assume that each input and output is dependent on all other layers. These types of neural networks are called recurrent because they sequentially perform mathematical computations.



Application of RNN

RNN has multiple uses when it comes to predicting the future. In the financial industry, RNN can help predict stock prices or the sign of the stock market direction (i.e., positive or negative).

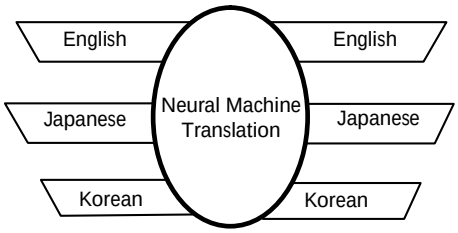
RNN is used for an autonomous car as it can avoid a car accident by anticipating the route of the vehicle.

RNN is widely used in image captioning, text analysis, machine translation, and sentiment analysis. For example, one should use a movie review to understanding the feeling the spectator perceived after watching the movie. Automating this task is very useful when the movie company can not have more time to review, consolidate, label, and analyze the reviews. The machine can do the job with a higher level of accuracy.

Following are the application of RNN:

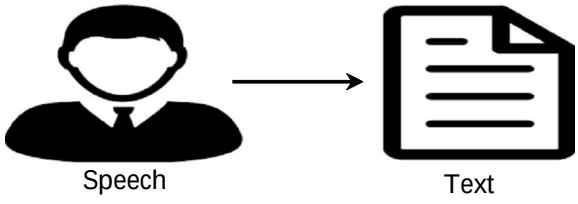
1. Machine Translation

We make use of Recurrent Neural Networks in the translation engines to translate the text from one to another language. They do this with the combination of other models like LSTM (Long short-term memory).



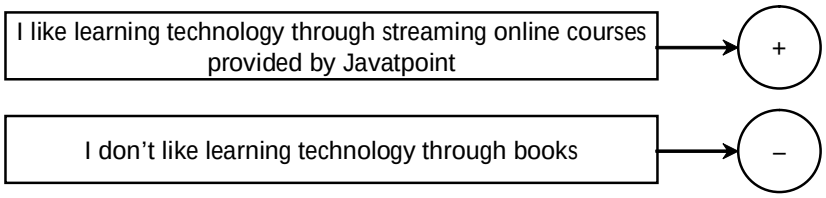
2. Speech Recognition

Recurrent Neural Networks has replaced the traditional speech recognition models that made use of Hidden Markov Models. These Recurrent Neural Networks, along with LSTMs, are better poised at classifying speeches and converting them into text without loss of context.



3. Sentiment Analysis

We make use of sentiment analysis to positivity, negativity, or the neutrality of the sentence. Therefore, RNNs are most adept at handling data sequentially to find sentiments of the sentence.



4. Automatic Image Tagger

RNNs, in conjunction with convolutional neural networks, can detect the images and provide their descriptions in the form of tags. For example, a picture of a fox jumping over the fence is better explained appropriately using RNNs.

A White Dog Jumping into the water

Limitations of RNN

RNN is supposed to carry the information in time. However, it is quite challenging to propagate all this information when the time step is too long. When a network has too many deep layers, it becomes untrainable. This problem is called: vanishing gradient problem.

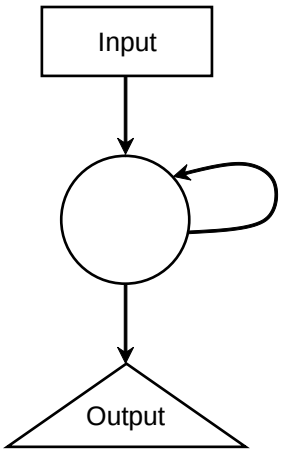
If we remember, the neural network updates the weight use of the gradient descent algorithm. The gradient grows smaller when the network progress down to lower layers.

The gradient stays constant, meaning there is no space for improvement. The model learns from a change in its gradient; this change affects the network's output. If the difference in the gradient is too small (i.e., the weight change a little), the system can't learn anything and so the output. Therefore, a system facing a vanishing gradient problem cannot converge towards the right solution.

The recurrent neural will perform the following

The recurrent network first performs the conversion of independent activations into dependent ones. It also assigns the same weight and bias to all the layers, which reduces the complexity of RNN of parameters. And it provides a standard platform for memorization of the previous outputs by providing previous output as an input to the next layer.

These three layers having the same weights and bias, combine into a single recurrent unit.



For calculating the current state-

$$h_t = f(h_{t-1}, X_t)$$

Where h_t = current state

H_{t-1} = previous state

X_t = input state

To apply the activation function tanh, we have-

$$ht = \tanh (W_{hh} h_t\text{-}1 + W_{xh} X_t)$$

Where:

W_{hh} = weight of recurrent neuron and,

W_{xh} = weight of the input neuron

The formula for calculating output:

$$Y_t = W_{hy} h_t$$

Training through RNN

- The network takes a single time-step of the input.
- We can calculate the current state through the current input and the previous state.
- Now, the current state through h_{t-1} for the next state.
- There is n number of steps, and in the end, all the information can be joined.
- After completion of all the steps, the final step is for calculating the output.
- At last, we compute the error by calculating the difference between actual output and the predicted output.
- The error is backpropagated to the network to adjust the weights and produce a better outcome.

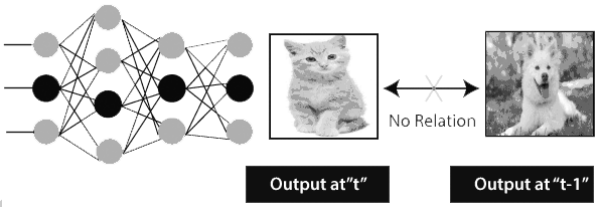
Working of RNN in TensorFlow

Recurrent Neural Networks have vast applications in image classification and video recognition, machine translation, and music composition.

Consider an image classification use-case where we have trained the neural network to classify images of some animals.

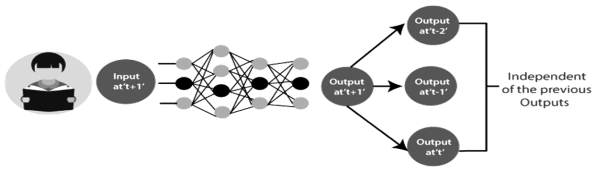
So, let's feed an image of a cat or a dog; the network provides an output with the corresponding label to the picture of a cat or a dog.

See the below diagram



Here, the first output being a cat will not influence the previous output, which is a dog. This means that output at a time 't' is autonomous of output at the time 't-1?.

Consider the scenario where we will require the use of the last obtained output:



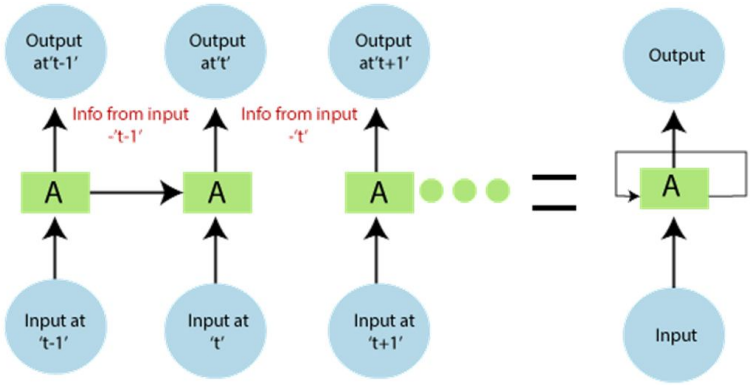
The concept is the same as reading a book. With every page we move forward into, we need the understanding of previous pages to make complete sense of the information in most of the cases.

With the help of the feed-forward network, the new output at the time 't+1? has no relation with outputs at either time t, t-1, t-2.

So, the feed-forward network cannot be used when predicting a word in a sentence as it will have no absolute relation with the previous set of words.

But, with the help of Recurrent Neural Net-works, this challenge can be overcome.

See the following diagram



In the above diagram, we have specific inputs at-1? which is fed into the network. These inputs will lead to parallel outputs at time 't-1' as well.

In the next timestamp, information from the previous input't-1? is provided along with input at't? to provide the output at 't eventually.'

This process repeats itself, to ensure that the latest inputs are aware and can use the information from the previous timestamp is obtained.

The recurrent network is a type of artificial neural network which is designed to recognize patterns in sequences of data. Like, text, genomes, handwriting, the spoken word, numerical times series data from sensors, stock markets, and government agencies.

4.5.1 A Recurrent Layer in Keras

Q8. Write about recurrent layer in Keras

Ans: **(Imp.)**

Recurrent Layers

RNN:

keras.engine.base
_layer.wrapped_fn()

The RNN layer act as a base class for the recurrent layers.

Arguments

- **cell :** It can be defined as an instance of RNN cell, which is a class that constitutes:
- **A call:** (input_at_t, states_at_t) method that returns (output_at_t, states_at_t_plus_1). It may optionally take a constant argument, which is explained below more briefly in the section "Note on passing external constants".
- **A state_size:** attribute can be simply defined as a single integer (state integer) or a list/tuple of integers (one size per state). In case of a single integer, it acts as a size of the recurrent state that is mandatory to be similar to the size of the output cell.
- **An output_size:** attribute, which can be referred to as a single integer or a TensorShape that epitomizes the shape of output. In case of a backward-compatible reason when the attribute is unavailable for the cell, there may be a chance that the value may get inferred by its initial element on the state_size.

- **Also, there may be a possibility** where the cell is a list of RNN cell instances; then, in that case, the cell gets stacked one after another in the RNN, leading to an efficient implemen-tation of the stacked RNN.
- **return_sequences:** It is a Boolean that depicts the last output to be returned either in the output sequence or the full sequence.
- **return_states:** It is also Boolean that depicts for the last state if it should be returned in addition to the output.
- **go_backwards:** It is Boolean, which is by default False. In case if it is set to True, then it backwardly processes the sequence of input and reverts back with the reversed sequence.
- **stateful:** It is Boolean, which is by default False. If stateful is set to True, then for each sample in the batch at the i^{th} index, the last state will be utilized as the initial state for the sample of the i^{th} index in the following batch.
- **unroll:** It is Boolean (False by default). If in case it is true, then either it will unroll the network, or it will utilize a symbolic loop. The RNN can speed up on unrolling even if it is memory-intensive as it is much more suitable for shorter sequences.
- **input_dim:** It is an integer that depicts the dimensionality of the input. The input_shape argument will be utilized when this layer will be used as an initial layer in the model.
- **input_length:** It describes the length of the input sequences, which is specified when it is constant. It is used when we first want to connect it to the Flatten and then to the Dense layers upstream as it helps to compute the output shape of the dense layer. If the recurrent layer is not the initial layer in the model, then you will have to specify the length of the input at the level of the first layer via input_shape

Input shape

It is a 3D tensor of shape (batch_size, timesteps, input_dim).

Output shape

- **If the return_state:** a list of tensors, then the first tensor will be the output and the remaining will be the last states, each of shape (batch_size, units) like for example; For RNN and GRU, the number of state of tensors is 1 and for LSTM is 2.
- **If the return_sequence:** 3D, then the shape of a tensor will be (batch_size, timesteps, units), else if it is a 2D, then the shape will be (batch_size, units).

Masking

Masking is supported by this layer to input the data with several numbers of timesteps. The **Embedding** layer is utilized with the **mask_zero** parameter, which is set to **True**, for introducing masks to the data.

Note on using statefulness in RNNs

If you set the RNN layer as 'stateful', then it means states that are computed in a single batch for the samples are used again as initial states for the next batch samples. It means that one-to-one mapping is done in between the samples in distinct consecutive batches.

For enabling statefulness, you need to specify stateful=True inside the constructor layer followed by specifying a fixed batch size for the model, which is done by passing if sequential model: batch_input_shape=(…) to the initial (first) layer in the model, else for any functional model consisting 1 or more Input layers: batch_shape=(…) to all the first layers in the model. The expected shape of inputs includes the batch size to be a tuple of integers, for example (32, 10, 100) and specify shuffle = False while calling fit().

Also, you need to call **.reset_states()** either on a specified layer or on the entire model, if you are willing to reset the states of your model.

Note on specifying the initial states of RNNs

The initial state of RNN layers can be symbolically specified by calling them with **initial_state** keyword argument, such that its value must be a tensor or list of tensors depicting the initial states of the RNN layer.

The initial state of RNN layers can be numerically specified by calling **reset_states** with **states** keyword argument, such that its value must either be a numpy array or a list of arrays depicting the initial states of the RNN layers.

Note on passing external constants to RNNs

The external constants can be pass on the cell by utilizing the constants keyword argument of RNN.__call__ and RNN.call method for which it necessitates the cell.call method to accept the same keyword arguments constants. These constants are utilized for conditioning the cell transformation on additional static inputs (that does not change over time).

4.5.2 Understanding the LSTM and GRU layers

Q9. Explain in detail about LSTM layers?

Ans : **(Imp.)**

LSTM

Keras LSTM stands for the Long short-term memory layer, which Hochreiter created in 1997. This layer uses available constraints and runtime hardware to gain the most optimized performance where we can choose the various implementation that is pure tensorflow or cuDNN based.

keras.layers.LSTM(units, activation ='tanh', recurrent_activation= 'sigmoid', use_bias= True, kernel_initializer=' glorot_uniform', recurrent_initializer=' orthogonal', bias_initializer= 'zeros', unit_forget_bias=Tr ue, kernel_regularizer =None, recurrent_regularizer=None, bias_regularizer= None, activity_regularizer= None, kernel_constraint= None, recurrent_constraint = None, bias_constraint= None, dropout =0.0, recurrent_dropout=0.0, implementation = 2, return_sequences = False, return_state = False, go_backwards = False, stateful = False, unroll = False)

Arguments

- **Units:** It refers to a positive integer that represents the output space dimensionality.
- **Activation:** It can be defined as an activation function to be used, which is a hyperbolic tangent (tanh) by default. If None is passed then it means nothing has been applied (i.e. "linear" activation a(x) = x).
- **recurrent_activation:** It is an activation function that is utilized for the recurrent step and is by default, hard sigmoid **(hard _sigmoid)**. If **None** is passed then it means nothing has been applied (i.e. "linear" activation a(x) = x).
- **use_bias:** It refers to Boolean that depicts for the layer whether to use a bias vector or not.
- **kernel_initializer:** It refers to an initializer for the **kernel** weights matrix that is utilized to linearly transform the inputs.
- **recurrent_initializer:** It refers to an initializer for the **recurrent_kernel** weights matrix that is supposed to be used while linearly transforming the recurrent states.
- **bias_initializer:** It indicates an initializer for bias vector.
- **unit_forget_bias:** It indicates a Boolean, and if set to True, 1 will be added to the bias of the forget gate at the initialization. Also, it will enforce the **bias_initializer="zeros"**.
- **kernel_regularizer:** It refers to a regularizer function, which is being applied to the **kernel** weights matrix.
- **recurrent_regularizer:** It refers to a regularizer function that is applied to the **recurrent_kernel** weight matrix.
- **bias_regularizer:** It refers to the regularizer function, which is being implemented on the bias vector.
- **activity_regularizer:** It refers to the regularizer function that is applied to the activation (output of the layer).
- **kernel_constraint:** It refers to a constraint function executed on the **kernel**
- **bias_constraint:** It refers to a constraint function, which is being applied to the bias vector.
- **recurrent_constraint:** It is that constraint function that is applied to the **recurrent _kernel** weights matrix.

- **dropout:** It is a float in between 0, and 1that depicts the total number of the fraction of units to be dropped to linearly transform the input.
- **recurrent_dropout:** It is a float in between 0, and 1that depicts the total number of the fraction of units to be dropped to linearly transform the recurrent state.
- **implementation:** It is an implementation mode, which is either 1 or 2. In mode 1, operations will be structure as a large number of smaller dot products and additions, whereas in mode 2, it will batch them as a few large operations. These modes will showcase different performance profiles over distinct hardware and applications.
- **return_sequences:** It refers to a Boolean that depicts for the last output to be returned either in the output sequence or the full sequence.
- **return_states:** It refers to also Boolean that depicts for the last state if it should be returned in addition to the output.
- **go_backwards:** It can be defined as a Boolean, which is by default False. In case, if it is true, then it backwardly processes the input sequence and reverts back the reversed sequence.
- **stateful:** It can be understood as Boolean, which is by default False. If it is True, then for each sample in the batch at the i^{th} index, the last state will be utilized as the initial state for the sample of the i^{th} index in the following batch.
- **unroll:** It indicates to a Boolean (False by default). If in case it is true, then either it will unroll the network, or it will utilize a symbolic loop. The RNN can speed up on unrolling even if it is memory-intensive as it is much more suitable for shorter sequences.

4.5.2.1 GRU

Q10. Write a short note on GRU (Gated Recurrent Unit) with suitable illustrations.

Ans : **(Imp.)**

GRU

Introduction to KerasGRU

Keras GRU abbreviation is gated recurrence unit which was introduced in 2014. It is very simpler and very similar to the LSTM. It is nothing but the LSTM without using an output gate. They are performing similarly to the LSTM for most of the tasks, but it will perform better on multiple tasks by using a smaller dataset and using data that was less frequently used in the keras network.

Key Takeaways

- The keras GRU is known as a gated recurrent unit or it is an RNN architecture that was similar to the units of LSTM.
- It will comprise the update gate and reset gate instead of using an input. The reset gate will combine new input from memory.

Keras GRU

By using keras and TensorFlow we are building the neural network which was very easy to build. We can easily build the neural network by using libraries of keras and TensorFlow. Basically, GRU is an improved version of the RNN model. It is more efficient than the RNN model which was very simple. It is an improved version of the RNN model. This model is very useful and efficient compared to the RNN model.

The model contains two gates first is reset and the second is updated. We can also replace the simple RNN layer with the bidirectional gru layer. The GRU model will comprise the reset gate and update the gate instead of an input. It will forget the gate of the LTSM. The reset gate will determine how we are combining the new input by using previous memory and the output gate will define how much memory we are keeping around.

Keras GRU Layers

As per the available constraint and hardware, the GRU layer is choosing different implementations

for maximizing the performance. If the gru is available and all the arguments of the layer are meeting the requirement of the cuDNN kernel. This layer will use the cuDNN implementation. The below example shows how kerasgru uses the layer as follows.

Code

```
tf.keras.layers.GRU(
units,
activation,
return_state = False,
go_backwards = False,
stateful = False,
unroll = False,
time_major = False,
reset_after = True,
recurrent_activation = "sigmoid",
use_bias = True,
kernel_initializer = "glorot_uniform",
recurrent_initializer = "orthogonal",
bias_initializer = "zeros",
bias_constraint = None,
dropout = 0.0,
recurrent_dropout = 0.0,
return_sequences = False,
)
```

Output:

```
> > >
> > >   tf.keras.layers.GRU(
. . .       units,
. . .       activation,
. . .     return_state = False,
. . .       go_backwards - False,
. . .       stateful = False,
. . .       unroll = False,
. . .       time_major = False,
. . .       reset_after = True,
. . .       recurrent_activation = "sigmoid",
. . .       usebias = True,
. . .       kernel_initializer = "glorot_uniform"
```

```
. . .        recurrent_initializer = “orthogonal”,
. . .        bias_initlalizer = “zeros”,
. . .        bias_constraint = None,
. . .        dropout_0.0,
. . .        recurrent_dropout = 0.0,
. . .        return_sequences = False,
. . .     )
```

Basically, there are two variants available for GRU implementation. Default is based on the v3 and it contains the reset gate which was applied to the matrix multiplication. The other is based on the original and it contains the order reserved.

Code:

```
ip=tf.random.normal([22,12,6])
lay =tf.keras.layers.GRU(6)
op =lay(ip)
print(op.shape)
```

Output

The below variant of gru layer is compatible with the GPU only. So it will contain separate biases.

Code

```
lay =tf.keras.layers.GRU(4,return_sequences= True,return_state= True)
whole_sequence_output,final_state=lay(ip)
print(whole_sequence_output.shape)
print(final_state.shape)
```

Output

```
> > >
> > >
> > >
> > > lay = tf.keras.layers.GRU (4, return_sequences = True, return_state = True)
> > > whole_sequence_output, final_state = lay (ip)
> > > print(whole_sequence_output.shape)
(22, 12, 4)
> > > print (final_state.shape)
(22, 4)
> > >
> > >
```

Keras GRU contains the below arguments which we need to define while implementing the gru layer.

- activation
- units
- recurrent_activation
- use_bias
- kernel_initializer
- recurrent_initializer
- bias_regularizer
- kernel_regularizer
- recurrent_regularizer
- go_backwards
- stateful
- bias_initializer
- activity_regularizer
- kernel_constraint
- recurrent_constraint
- bias_constraint
- dropout
- recurrent_dropout
- return_sequences
- return_state
- time_major
- reset_after

The kerasgru layer also contains the call arguments i.e. mask, input training, and initial state. We can use all these arguments at the time of defining it.

Keras GRU Methods

Given below are the methods mentioned:

1. get_dropout_mask_for_cell

This method will get mask dropout from the cells RNN input. This method is creating mask based context into the cached mask. If suppose a new mask is generated it will update the cache in the cell. This method will contain count, input, and training arguments. The below example shows get_dropout_mask_for_cell method.

Code:

```
importtensorflow.compat.v2 astf

from kerasimport backend
```

```
classgru:
def get_dropout_mask_for_cell(self, inputs, training, count=1):
ifself.dropout==0:
return None
```

Output:

```
>>>
>>>
>>> import tensorflow.compat.v2 as tf
>>> from tensorflow.tools.docs import doc_controls
>>> from keras import backend
>>> class gru:
... def get_dropout_mask_for_cell (self, inputs, training, count-1);
```

2. get_recurrent_dropout_mask_for_cell

This method will get the recurrent mask dropout from the RNN cell. It will create mask based context which was existing in the cached mask. This method will contain count, input, and training arguments. Below example shows get_recurrent _dropout _mask_for_cell method.

Code:

```
importtensorflow.compat.v2 astf
from kerasimport backend
classgru:
def get_recurrent_dropout_mask_for_cell(self, inputs, training, count=1):
```

Output:

```
>>>
>>>
>>> import tensorflow.compat.v2 as tf
>>> from keras import backend
>>> class gru:
... def get_recurrent_dropout_mask_for_cell (self, inputs, training, count=1);
```

3. reset_dropout_mask

This method is used in resetting the dropout mask. This method is important in the RNN layer for invoking the call method so we can clear the cached method by calling the call method. The below example shows get reset_dropout_mask method.

Code:

```
importtensorflow.compat.v2 astf
from kerasimport backend
classgru:
def reset_dropout_mask(self):
```

Output:

```
> > >
> > >
> > >  import tensorflow.compat.v2 as tf
> > >  from keras import backend
> > >  class gru:
. . .     def reset_dropout_mask (self):
```

4. reset_recurrent_dropout_mask

This method is used in resetting the recurrent dropout mask. This method is important in the RNN layer for invoking the call method so we can clear the cached mask method. The below example shows get reset_recurrent_dropout_mask method.

Code:

```
importtensorflow.compat.v2 astf
from kerasimport backend
classgru:
def reset_recurrent_dropout_mask(self):
```

Output:

```
> > >
> > >
> > >  import tensorflow.compat.v2 as tf
> > >  from keras import backend
> > >  class gru:
. . .     def reset_recurrent_dropout_mask (self):
```

5. reset_states

This method is used to reset the recorded states from the RNN layer. We can use the RNN layer which was constructed from stateful arguments. Numpy array contains the initial state value. The below example shows reset_states method.

Code:

```
importtensorflow.compat.v2 astf
from kerasimport backend
classgru:
def reset_states(states = None):
```

Output:

```
> > >
> > >
> > >  import tensorflow.compat.v2 as tf
> > >  from keras import backend
> > >  class gru:
. . .     def reset_states (states = None):
. . .
```

Keras GRU Network

The GRU unit doesn't need to use a memory unit for controlling the flow of information unit as LSTM. It will make use of hidden states without any control. GRU contains parameters for training the faster and generalizing the large data. The gru network is very similar to the LSTM except it will contain two gates update and reset gate. The reset gate will determine how we can combine the new input with the previous memory. The below example shows how kerasgru works as follows. In the below example, we are creating the model.

Code

```
mod =keras.Sequential()
mod.add(layers.GRU(64,input_shape=(32,32)))
mod.add(layers.BatchNormalization())
mod.add(layers.Dense(15))
print(mod.summary())
```

Output

```
> > >
> > >     import transforflowas tf
> > >     from tensorflow import keras
> > >     from trensorflow.keras import layers
> > >
> > >     mod = keras.sequential( )
> > >     mod.add(layers.GRU(64. Input_Shape = (32.  32)))
> > >     mod.add(layers.BatchNormalization())
> > >     mod.add(layers.Dense(15))
> > >     print(mod.summary())
```

Layer (type)	Output Shape	Parma #
gru_2 (GRU)	(None, 64)	18816
batch_normalization (BatchN ormaliztion)	(None, 64)	256
dense (Dense)	(None, 15)	975

Total params : 20,047
Trainable paramas : 19,919
Non-trainable params : 128

```
None
> > >
```

Model: "sequential"

In the below example, we are using the fit method to define the kerasgru network model.

Code

```
model.fit(
x_train,y_train, …
)
```

Output

```
> > >
> > >
> > > model.fit(
. . .         X_trai, y_train, validation_data = (x_validate, y_validate), batch_size=64, epochs = 18
. . . )
```

Now we are testing the kerasgru network model. We are using for loop for the same.

Code

```
foriinrange(10):
result =tf.argmax()
```

Output

```
> > >
> > >
> > >
> > >       for i in range(10):
. . .         reslult = tf.argmax(model.predict(tf.expand_dims(x_test[i], 0)), axis = 1)
. . .
```

Examples of Keras GRU

Given below are the examples mentioned:

Example #1

In the below example, we are using the layer.

Code

```
importtensorflowastf
from tensorflowimportkeras
ip=tf.random.normal([42,22,12])
lay =tf.keras.layers.GRU(24)
op =lay(ip)
print(op.shape)
```

Output

```
> > >
> > >
> > >    import tensorflow as tf
> > >    from tensorflow import keras
> > >    ip = tf.random.normal ([42, 22, 12])
> > >    lay = tf.keras.layers.GRU (24)
> > >    OP = lay(ip)
> > >    print (OP.shape)
(42, 24)
> > >
> > >
> > >
```

Example #2

In the below example we are importing the keras module.

Code

```
importtensorflowastf
from tensorflowimportkeras
lay =tf.keras.layers.GRU()
whole_sequence_output,final_state=lay(ip)
print(whole_sequence_output.shape)
print(final_state.shape)
```

Output

```
> > >
> > >
> > >  import tensorflow as tf
> > >  from tensorflow import keras
> > >  lay = tf.keras.layers.GRU (4, return_sequences = True, return_state = True)
> > >  whole_sequence_output, final_state = lay (ip)
> > >  print(whole_sequence_output.shape)
(42, 22, 4)
> > >  print (final_state.shape)
(42, 4)
> > >
> > >
```

Short Question and Answers

1. Write short note on Keras?

Ans :

Keras is a high-level, deep learning API developed by Google for implementing neural networks. It is written in Python and is used to make the implementation of neural networks easy. It also supports multiple backend neural network computation.

Keras is relatively easy to learn and work with because it provides a python frontend with a high level of abstraction while having the option of multiple back-ends for computation purposes. This makes Keras slower than other deep learning frameworks, but extremely beginner-friendly.

Keras allows you to switch between different back ends. The frameworks supported by Keras are:

- TensorFlow
- Theano
- PlaidML
- MXNet
- CNTK (Microsoft Cognitive Toolkit)

Out of these five frameworks, TensorFlow has adopted Keras as its official high-level API. Keras is embedded in TensorFlow and can be used to perform deep learning fast as it provides inbuilt modules for all neural network computations. At the same time, computation involving tensors, computation graphs, sessionsetc. can be custom made using the TensorFlow Core API, which gives you total flexibility and control over your application and lets you implement your ideas in a relatively short time.

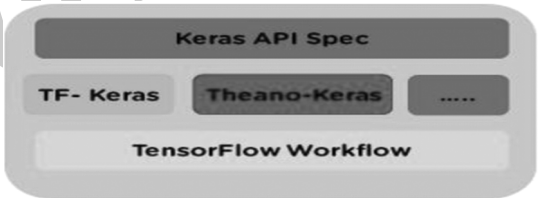


Fig.: Keras Backend

2. Write short note on Keras features and benefits?

Ans :

Features

Keras leverages various optimization techniques to make high level neural network API easier and more performant. It supports the following features “

- Consistent, simple and extensible API.
- Minimal structure - easy to achieve the result without any frills.
- It supports multiple platforms and backends.
- It is user friendly framework which runs on both CPU and GPU.
- Highly scalability of computation.

Benefits

Keras is highly powerful and dynamic framework and comes up with the following advantages

- Larger community support.
- Easy to test.
- Keras neural networks are written in Python which makes things simpler.
- Keras supports both convolution and recurrent networks.
- Deep learning models are discrete components, so that, you can combine into many ways.

3. Building Model of a Keras:

Ans :

The below diagram shows the basic steps involved in building a model in Keras:

Fig. : Building a Model

(i) Define a network: In this step, you define the different layers in our model and the connections between them. Keras has two main types of models: Sequential and Functional models. You choose which type of model you want and then define the dataflow between them.

(ii) Compile a network: To compile code means to convert it in a form suitable for the machine to understand. In Keras, the model.compile() method performs this function. To compile the model, we define the loss function which calculates the losses in our model, the optimizer which reduces the loss, and the metrics which is used to find the accuracy of our model.

(iii) Fit the network: Using this, we fit our model to our data after compiling. This is used to train the model on our data.

(iv) Evaluate the network: After fitting our model, we need to evaluate the error in our model.

(v) Make Predictions: We use model.predict() to make predictions using our model on new data.

4. Discuss about Applications of Keras

Ans :

- Keras is used for creating deep models which can be productized on smartphones.
- Keras is also used for distributed training of deep learning models.
- Keras is used by companies such as Netflix, Yelp, Uber, etc.
- Keras is also extensively used in deep learning competitions to create and deploy working models, which are fast in a short amount of time.

5. What are the Features of TensorFlow?

Ans :

TensorFlow has an interactive multiplatform programming interface which is scalable and reliable compared to other deep learning libraries which are available.

These features of TensorFlow will tell us about the popularity of TensorFlow.

(i) Responsive Construct: We can visualize each part of the graph, which is not an option while using Numpy or SciKit. To develop a deep learning application, firstly, there are two or three components that are required to create a deep learning application and need a programming language.

(ii) Flexible: It is one of the essential TensorFlow Features according to its operability. It has modularity and parts of it which we want to make standalone.

(iii) Easily Trainable: It is easily trainable on CPU and for GPU in distributed computing.

(iv) Parallel Neural Network Training: TensorFlow offers to the pipeline in the sense that we can train multiple neural networks and various GPUs, which makes the models very efficient on large-scale systems.

(v) Large Community: Google has developed it, and there already is a large team of software engineers who work on stability improvements continuously.

(vi) Open Source: The best thing about the machine learning library is that it is open source so anyone can use it as much as they have internet connectivity. So, people can manipulate the library and come up with a fantastic variety of useful products. And it has become another DIY community which has a massive forum for people getting started with it and those who find it hard to use it.

6. Write short note on CNTK?

Ans :

Microsoft Cognitive Toolkit (CNTK), formerly known as Computational Network Toolkit, is a free, easy-to-use, open-source, commercial-grade toolkit that enables us to train deep learning algorithms to learn like the human brain. It enables us to create some popular deep learning systems like feed-forward neural network time series prediction systems and Convolutional neural network (CNN) image classifiers.

CNTK allows the user to easily realize and combine popular model types such as feed-forward DNNs, convolutional neural networks (CNNs) and recurrent neural networks (RNNs/LSTMs).

For optimal performance, its framework functions are written in C++. Although we can call its function using C++, but the most commonly used approach for the same is to use a Python program.

Microsoft Research developed CNTK, a deep learning framework that builds a neural network as a series of computational steps via a direct graph. CNTK supports interfaces such as Python and C++ and is used for handwriting, speech recognition, and facial recognition.

7. What are the features of CNTK?

Ans :

Following are some of the features and capabilities offered in the latest version of Microsoft CNTK:

Built-in Components

- CNTK has highly optimised built-in components that can handle multi-dimensional dense or sparse data from Python, C++ or BrainScript.
- We can implement CNN, FNN, RNN, Batch Normalisation and Sequence-to-Sequence with attention.
- It provides us the functionality to add new user-defined core-components on the GPU from Python.
- It also provides automatic hyperparameter tuning.
- We can implement Reinforcement learning, Generative Adversarial Networks (GANs), Supervised as well as Unsupervised learning.
- For massive datasets, CNTK has built-in optimised readers.

Usage of resources efficiently

- CNTK provides us parallelism with high accuracy on multiple GPUs/machines via 1-bit SGD.
- To fit the largest models in GPU memory, it provides memory sharing and other built-in methods.

8. Explain in detail about Recurrent neural networks (RNN's)?

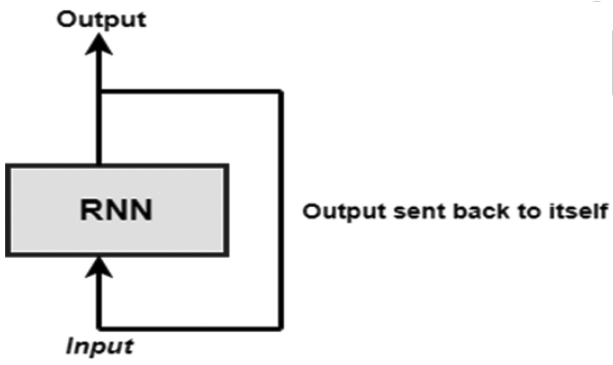
Ans :

A recurrent neural network (RNN) is a kind of artificial neural network mainly used in speech recognition and natural language processing (NLP). RNN is used in deep learning and in the development of models that imitate the activity of neurons in the human brain.

Recurrent Networks are designed to recognize patterns in sequences of data, such as text, genomes, handwriting, the spoken word, and numerical time series data emanating from sensors, stock markets, and government agencies.

A recurrent neural network looks similar to a traditional neural network except that a memory-state is added to the neurons. The computation is to include a simple memory.

The recurrent neural network is a type of deep learning-oriented algorithm, which follows a sequential approach. In neural networks, we always assume that each input and output is dependent on all other layers. These types of neural networks are called recurrent because they sequentially perform mathematical computations.



9. What are the Applications of RNN?

Ans :

Following are the application of RNN:

(i) Machine Translation: We make use of Recurrent Neural Networks in the translation engines to translate the text from one to another language. They do this with the combination of other models like LSTM (Long short-term memory).

(ii) Speech Recognition: Recurrent Neural Networks has replaced the traditional speech recognition models that made use of Hidden Markov Models. These Recurrent Neural Networks, along with LSTMs, are better poised at classifying speeches and converting them into text without loss of context.

(iii) Sentiment Analysis: We make use of sentiment analysis to positivity, negativity, or the neutrality of the sentence. Therefore, RNNs are most adept at handling data sequentially to find sentiments of the sentence.

(iv) Automatic Image Tagger: RNNs, in conjunction with convolutional neural networks, can detect the images and provide their descriptions in the form of tags. For example, a picture of a fox jumping over the fence is better explained appropriately using RNNs.

10. Differentiate between TensorFlow and Keras?

Ans :

	Keras	**TensorFlow**
API Level	High	High and Low
Architecture	Simple, concise, readable	Not easy to use
Datasets	Smaller datasets	Large datasets, high performance
Debugging	Simple network, so debugging is not often needed	Difficult to conduct debugging
Does It Have Trained Models?	Yes	Yes
Popularity	Most popular	Second most popular
Speed	Slow, low performance	Fast, high-performance
Written In	Python	C++, CUDA, Python

Choose the Correct Answer

1. _____ is a high level API built on TensorFlow. [b]
   (a) PyBrain (b) Keras
   (c) PyTorch (d) Theano
2. Is keras a library? [a]
   (a) Yes (b) No
   (c) Can be yes or no (d) Can not say
3. Who invented keras? [d]
   (a) Michael Berthold (b) Adam Paszke
   (c) Sam Gross (d) François Chollet
4. _____ is a regularization technique for neural network models proposed by Srivastava, it is a technique where randomly selected neurons are ignored during training. [c]
   (a) Callout (b) Digout
   (c) Dropout (d) Knimeout
5. What is true about Keras? [d]
   (a) Keras is an API designed for human beings, not machines.
   (b) Keras follows best practices for reducing cognitive load
   (c) It provides clear and actionable feedback upon user error
   (d) All of the above
6. What are advanced activation functions in keras? [c]
   (a) LeakyReLU (b) PReLU
   (c) Both A and B (d) None of them
7. Which of the following are correct initializers in keras? [d]
   (a) keras.initializers.Initializer() (b) keras.initializers.Zeros()
   (c) keras.initializers.Ones() (d) All of the above
8. A _____ requires shape of the input (input_shape) to understand the structure of the input data. [a]
   (a) Keras layer (b) Keras Module
   (c) Keras Model (d) Keras Time
9. Which of the following returns all the layers of the model as list? [b]
   (a) model.inputs (b) model.layers
   (c) model.outputs (d) model.get_weights
10. Keras is a _____. [b]
   (a) Data science library (b) Neural network library
   (c) Data testing library (d) None of them

Fill in the blanks

1. Keras is written in _____.
2. Keras is a _____.
3. Keras developed by _____.
4. Keras support _____ and _____ neural networks
5. How many backend engines does Keras consist is _____.
6. TensorFlow is a product of _____.
7. TensorFlow helps us to perform _____. and _____.
8. Which kind of library Theano is _____.
9. tf.keras.backend.set_floatx(value) will return _____.
10. What is the return value of the epsilon function is _____.

ANSWERS

1. Python
2. Neural network library
3. François Chollet
4. Convolutional, recurrent
5. Three
6. Google
7. Data automation, Model tracking
8. Mathematical operation
9. Returns the default float type, as a string
10. It returns the fuzz factor

One Mark Answers

1. Who is the Creator of Keras?

Ans :

François Chollet, He is currently working as an AI Researcher at Google.

2. List out layers in keras?

Ans :

- Core Layers
- Convolutional Layers
- Pooling Layers
- Locally-connected Layers
- Recurrent Layers
- Embedding Layers
- Merge Layers
- Advanced Activations Layers
- Normalization Layers
- Noise layers

3. What is the significance of using the Fourier transform in Deep Learning tasks?

Ans :

The Fourier transform function efficiently analyzes, maintains, and manages large datasets. You can use it to generate real-time array data that is helpful for processing multiple signals.

4. What are some of the uses of Autoencoders in Deep Learning?

Ans:

- Autoencoders are used to convert black and white images into colored images.
- Autoencoder helps to extract features and hidden patterns in the data.
- It is also used to reduce the dimensionality of data.
- It can also be used to remove noises from images.

5. What is Data Augmentation in Deep Learning?

Ans :

Data Augmentation is the process of creating new data by enhancing the size and quality of training datasets to ensure better models can be built using them. There are different techniques to augment data such as numerical data augmentation, image augmentation, GAN-based augmentation, and text augmentation.

6. Explain two ways to deal with the vanishing gradient problem in a deep neural network.

Ans :

- Use the ReLU activation function instead of the sigmoid function
- Initialize neural networks using Xavier initialization that works with tanh activation.

7. What APIs does Keras have?

Ans :

Keras has two main APIs: the Sequential API and the Functional API. The Sequential API is used for creating simple models, while the Functional API is used for creating more complex models.

8. How many ways can we initialize weights in Keras?

Ans :

There are three ways that we can initialize weights in Keras:

(i) Initializing all weights to the same value

(ii) Initializing weights randomly

(iii) Initializing weights using a pre-trained model

9. Why is there a need for keras?

Ans :

Keras is an API designed for human beings, not machines. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.

10. What is keras dropout?

Ans :

Dropout is a regularization technique for neural network models proposed by Srivastava, it is a technique where randomly selected neurons are ignored during training.

FACULTY OF SCIENCE

B.Sc. VI-Semester(CBCS) Examination

MODEL PAPER - I

DEEP LEARNING

Time : 3 Hours] **[Max. Marks : 80**

PART - A (8 × 4 = 32 Marks)

Note : Answer any Eight questions. All questions carry equal marks.

ANSWERS

1. Differences between ML and DL. **(Unit-I, SQA-3)**
2. Applications of Deep Learning. **(Unit-I, SQA-4)**
3. Write short note on Deep Learning Algorithms. **(Unit-I, SQA-6)**
4. Write about Broadcasting in Tensors with suitable examples. **(Unit-II, SQA-1)**
5. Explain about Reshaping a Tensor with suitable examples. **(Unit-II, SQA-3)**
6. What are the main differences between AI, Machine Learning, and Deep Learning? **(Unit-II, SQA-6)**
7. Discuss about Gradient-based optimization. **(Unit-III, SQA-1)**
8. What Do You Understand by Backpropagation? **(Unit-III, SQA-3)**
9. What is a Neural Network? **(Unit-III, SQA-4)**
10. Write short note on Keras. **(Unit-IV, SQA-1)**
11. Building Model of a Keras. **(Unit-IV, SQA-3)**
12. Discuss about Applications of Keras. **(Unit-IV, SQA-4)**

PART - B (4 × 12 = 48 Marks)

Note : Answer all the questions. All questions carry equal marks.

13. (a) What is Deep Learning? Explain in detail about Deep Learning. **(Unit-I, Q.No.1)**

OR

(b) What are the Real-World Examples of Tensors? Give with examples. **(Unit-I, Q.No.11)**

14. (a) Define Tensor? What Does Element-Wise Operations with examples. **(Unit-II, Q.No.1)**

OR

(b) Discuss about Geometric interpretation of tensor operations. **(Unit-II, Q.No.5)**

15. (a) Discuss about Gradient-based optimization. **(Unit-III, Q.No.1)**

OR

(b) Write a note on Anatomy of a neural network. **(Unit-III, Q.No.5)**

16. (a) Write short note on Keras features and benefits. **(Unit-IV, Q.No.1)**

OR

(b) What are the Features of TensorFlow? **(Unit-IV, Q.No.4)**

FACULTY OF SCIENCE

B.Sc. VI-Semester(CBCS) Examination

MODEL PAPER - II

DEEP LEARNING

Time : 3 Hours] **[Max. Marks : 80**

PART - A (8 × 4 = 32 Marks)

Note : Answer any Eight questions. All questions carry equal marks.

ANSWERS

1. What do you understand by Boltzmann Machine? **(Unit-I, SQA-10)**
2. What is the use of Deep learning in today's age, and how is it adding data scientists? **(Unit-I, SQA-9)**
3. Discuss about Data representations for neural networks. **(Unit-I, SQA-8)**
4. Discuss about Tensor dot with suitable examples. **(Unit-II, SQA-2)**
5. Write about Broadcasting in Tensors with suitable examples. **(Unit-II, SQA-1)**
6. Discuss about geometric interpretation of deep learning. **(Unit-II, SQA-4)**
7. What is gradient descent? What are the steps for using a gradient descent algorithm? **(Unit-III, SQA-5)**
8. Write a note on Anatomy of a neural network in Deep Learning. **(Unit-III, SQA-6)**
9. What are Layers in a Neural Network? **(Unit-III, SQA-7)**
10. Differentiate between TensorFlow and Keras. **(Unit-IV, SQA-10)**
11. What are the Features of TensorFlow? **(Unit-IV, SQA-5)**
12. Discuss about Applications of Keras. **(Unit-IV, SQA-4)**

PART - B (4 × 12 = 48 Marks)

Note : Answer all the questions. All questions carry equal marks.

13. (a) Explain the history (Evolution) of Deep Learning. **(Unit-I, Q.No.2)**

OR

(b) Write short note and examples on :

i) Image Data **(Unit-I, Q.No.14)**

ii) Video Data **(Unit-I, Q.No.15)**

14. (a) Write about Broadcasting in Tensors with suitable examples. **(Unit-II, Q.No.2)**

OR

(b) Explain about Reshaping a Tensor with suitable examples. **(Unit-II, Q.No.4)**

15. (a) Write a note on Derivative of a tensor operation. **(Unit-III, Q.No.2)**

OR

(b) Explain about Stochastic Gradient Descent with suitable examples. **(Unit-III, Q.No.3)**

16. (a) Explain about keras applications and its building model. **(Unit-IV, Q.No.2)**

OR

(b) Discuss in detail about CNTK. **(Unit-IV, Q.No.6)**

FACULTY OF SCIENCE

B.Sc. VI-Semester(CBCS) Examination

MODEL PAPER - III

DEEP LEARNING

Time : 3 Hours] **[Max. Marks : 80**

PART - A (8 × 4 = 32 Marks)

Note : Answer any Eight questions. All questions carry equal marks.

ANSWERS

1. Write a note on Deep Learning. **(Unit-I, SQA-1)**
2. Explain about Simple neural network architecture. **(Unit-I, SQA-7)**
3. Write short note on Deep Learning Algorithms. **(Unit-I, SQA-6)**
4. How can hyperparameters be trained in neural networks? **(Unit-II, SQA-7)**
5. What are some of the examples of supervised learning and unsupervised learning algorithms in Deep Learning? **(Unit-II, SQA-8)**
6. Explain about Reshaping a Tensor with suitable examples. **(Unit-II, SQA-3)**
7. What is a Neural Network? **(Unit-III, SQA-4)**
8. List out neural network models. **(Unit-III, SQA-8)**
9. What is loss function? and list out them. **(Unit-III, SQA-9)**
10. What are the Features of TensorFlow? **(Unit-IV, SQA-5)**
11. Write short note on CNTK. **(Unit-IV, SQA-6)**
12. Explain about Recurrent neural networks (RNN's). **(Unit-IV, SQA-8)**

PART - B (4 × 12 = 48 Marks)

Note : Answer all the questions. All questions carry equal marks.

13. (a) What is Neural Networks? Explain in detail about Neural Networks and it types. **(Unit-I, Q.No.6)**

OR

(b) Explain and List out Deep Learning Algorithms. **(Unit-I, Q.No.5)**

14. (a) Discuss about Tensor dot with suitable examples. **(Unit-II, Q.No.3)**

OR

(b) Explain about Reshaping a Tensor with suitable examples. **(Unit-II, Q.No.4)**

15. (a) Explain in detail about The Backpropagation algorithm. **(Unit-III, Q.No.4)**

OR

(b) Discuss about Layers in Neural networks. **(Unit-III, Q.No.6)**

16. (a) What is Theano library in deep learning with suitable examples? **(Unit-IV, Q.No.5)**

OR

(b) Explain in detail about Recurrent neural networks (RNN's). and working of RNN's. **(Unit-IV, Q.No.7)**