1.1 HISTORY OF DEEP LEARNING

History of Deep Learning

- The history of deep learning dates back to 1943. During this period Warren McCulloch and Walter Pitts created a computer model based on Neural Networks of the human brain.
- Warren McCulloch and Walter Piits used a combination of mathematics and algorithms which they called as Threshold Logic to copy the thought process.
- Since then deep learning evolved steadily. After 1960, there were two significant breaks in its development.
- Henry J. Kelly developed the basics of a continuous Back Propagation Model in 1960.
- In 1962, Stuart Dreyfus came up with a simpler version based only on the chain-rule.
- The concept of back-propagation existed in the early 1960s but became useful in 1985.

Developing Deep Learning Algorithms

- The development of deep-learning algorithms began in 1965. Alexy Grigorievich Lvakhnenko and Valentine Grigo'evich LaPa used models with polynomial (complicated equations) activation functions, which were analysed statistically.
- During 1970's, lack of funding made a setback into the development of AI and deep learning. In spite of that individuals carried on the research without funding through those difficult years.
- Convolution neural networks were first used by Kunihiko Fukushima who designed the neural networks with multiple pooling and convolutional layers.
- Kunihiko Fukushima developed an artificial neural network, called Neo cognifrom in 1979. It used a multilayered and hierarchical design.
- The multilayered and hierarchical design allowed the computer to learn to recognise visual patterns.
- The networks resembled modern version and were trained with a reinforcement strategy of recurring activation in multiple layers, and it gained strength over time.

The Fortran Code for Back Propagation

- In 1970, back propagation, was developed and it used errors into training deep learning models.
- Seppo Linnainmaa made back propagation popular by using Fortran code for back propagation.
- It was developed in 1970. But this concept was applied to neural networks in 1985.

- Hinton and Rumelhart, Williams demonstrated back propagation in a neural network. And it provided interesting distribution representations.
- Yann LeCun explained the first practical demonstration of back propagation at Bell Labs in 1989 by combining convolutional neural networks with back propagation to read handwritten digits.
- The combination of convolutional neural networks with back propagation system was used to read the numbers of handwritten checks.
- During 1985-90, there was no progress in AI. And that effected research for neural networks and deep learning.
- In 1995 Vladimir Vapnik and Dana cortes developed the support vector machine which is a system for mapping and recognizing similar data.
- Long Short-term memory (or LSTM) was developed in 1997 by Schmidhuber and Sepp Hochreiter for recurrent neural networks.
- The next significant deep learning advancement was in 1999 when computers adopted the speed of the GPu processing.
- Faster processing meant increased computational speeds of 1000 times over a 10-year span.
- During this era, neural networks began competing with support vector machines.
- Neural networks offered better results using the same data, though slow to a support vector machine.

Deep Learning from the 2000's and Beyond

- In 2000 came out the Vanishing Gradient problem. Here features (lessons) formed in lower layers were not being learned by the upper layers, since no learning signal reached these layers were discovered.
- This problem was particularly of only gradient based learning methods.
- This problem turned out to be certain activation functions which condensed their input and reduced the output range in a chaotic fashion. This led to large areas of input mapped over an extremely small range.
- In 2001, a research report compiled by META Group came up with the challenges came out the Vanishing Gradient problem. Here features and opportunities of the three dimensional data growth.
- This report marked the creation of Big Data and described the increasing volume and speed of data as increasing the range of data sources and types.
- An AI Professor at Stanford University, Fei-Fei-Li launched image Net in 2009. He assembled a free data base of more than 14 million labeled images.
- These images were the inputs to train neural nets.

- The speed of GPUS increased significantly by 2011. And it made possible to train convolutional neural networks without the need of layer by layer pre-training.
- Deep learning has significant advantages into efficiency and speed.

The Cat Experiment

- In 2012, Google Brain released the results of a free-spirited project called the Cat Experiment. That explored the difficulties of unsupervised learning.
- Deep learning deploys supervised learning, which means the convolutional neural net is trained using labelled data like the images from imageNet.
- This experiment used a neural net which was spread over, 1,000 computers where ten million unlabeled images were taken randomly from YouTube, as inputs to the training software.
- From here onwards, unsupervised learning remains a significant goal in the field of deep learning. 2018 and years beyond will mark the evolution of artificial intelligence which will be dependent on deep learning.
- Deep learning is still in growth-phase and in constant need of creative ideas to evolve further.

1.2 DEEP LEARNING SUCCESS STORIES

Advantages of Deep learning

- Deep learning is more useful than its classical ML counterparts.
- This gives a massive opportunity for businesses looking for the technology to deliver high-performance outcomes. We mention the advantages of DL that led for this massive growth.

1. Feature Generation Automation

- DL algorithms can generate new features from among a limited number located in the training dataset without additional human intervention.
- This implies that deep learning can perform complex tasks that often require extensive feature engineering.
- For businesses, it is faster application that deliver superior accuracy.

2. Works Well with Unstructured Data

- DL has the ability to work with unstructured data. Majority of business data is unstructured, and hence it is relevant in the business content.
- Text, images and voice are some of the most common data formats that businesses use.

Classical ML algorithms are limited in their ability to analyse unstructured data, and here is where deep learning promises to make the most impact.

Training DL networks with unstructured data and appropriate labelling can help businesses optimise virtually every function from marketing and sales to finance.

3. Better Self-Learning Capabilities

- The multiple layers in deep neural networks allow models to become more efficient at learning complex features and performing more intensive computational tasks, i.e., execute many complex operations simultaneously.
- It has the ability to make sense of inputs like images, sounds, and video like a human would, that involve unstructured datasets.
- This is due to DL algorithms' ability to learn from its own errors.
- It can verify the accuracy of its predictions/outputs and make necessary adjustments.
- D. L's performance is directly proportional to the volume of the training datasets. Hence, the larger the datasets, the more accuracy.

4. Supports Parallel and Distributed Algorithms

- Parallel and Distributed Algorithms allow deep learning models to train much faster. Models can be trained using local training (i.e. using one machine to train the model), with GPUs, or a combination of both.
- In data parallelism, data or model is being distributed across multiple machines, and hence training is more effective.
- Parallel and distributed algorithms allow deep learning models to be trained at scale. Depending on the volume of your training dataset and GPU computing power, one can use as few as two or three computers to over 20 computers to complete the training within a day.

5. Cost effectiveness

- While training deep learning models can be cost-intensive, once trained, it can help business cut down on unnecessary expenditure.
- In industries such as manufacturing such as manufacturing, consulting or even retail, the cost of an inaccurate prediction or product defect is massive. This often outweighs the costs of training deep learning models.

6. Advanced Analytics

- D.L., when applied to data science, can offer better and more effective processing models.

- Its ability to learn unsupervised allows continuous improvement in accuracy and outcomes. It also offers data scientists with more reliable and concise analysis results.
- The technology powers most prediction software today with applications ranging from marketing to sales, HR, finance, and more

7. Scalability

- D. L. is highly scalable due to its ability to process massive amounts of data and perform a lot of computation in a cost-and time-effective manner. This directly impacts productivity and modularity and portability, (trained models can be used across a range of problems)
- For instance, Google's Cloud AI platform prediction allows you to run your deep neural network at scale on the cloud.
- In addition to better model organization and versioning, one can also leverage Google's cloud infrastructure to scale batch prediction. This improves efficiency by automatically scaling the number of nodes in use.

1.3 MULTILAYER PERCEPTRONS (MLPs)

- Multi-layer perceptrons is connected dense layers, and that transforms any input dimension to the desired dimension. A multilayer perceptron is a neural network that has multiple layers. To create a neural network, we combine neurons together so that the outputs of some neurons are inputs of other neurons.
- Multilayer perceptron are sometimes, referred to as "Vanilla" neural networks, especially when they have a single hideen layer.
- A multi-layer perceptron has one input layer and for each input, there is one neuron (or node), it has one output layer with a single node for each output and it can have any number of hidden layers and each hidden layer can have any number of nodes.
- We show a diagram of a multi-layer Perceptron (MLP) :

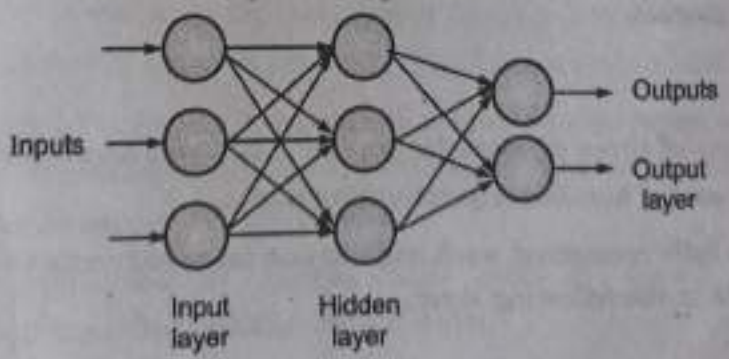


Fig. 1.3.1

- In the above diagram of multi-layer perceptron, there are three inputs and thus three input nodes and the hidden layer has three nodes.
- The output layer gives two outputs, hence there are two output nodes. The nodes in the input layer take input and forward it for further process.
- In the above diagram the nodes in the input layer forwards their output to each of the three nodes in the hidden layer, and in the same way, the hidden layer processes the information and passes it to the output layer.

Activation function

- Except for the input nodes, each node is a neuron that uses a non linear 'activation function'. MLP utilises a chain rule' based 'supervised learning' technique called back propagation for training. Its multiple layers and non-linear activation distinguishes MLP from a linear 'perceptron'. It can distinguish data that is not linearly separable.
- In MLPs some neurons use a nonlinear activation function to model the frequency of 'action potentials', or firing, of biological neurons.
- The two common activation functions are both sigmoids, and are given as :

$$y(x_i) = \tanh(x_i) \text{ and}$$

$$y(x_i) = (1 + e^{-x_i})^{-1} = \frac{1}{1 + e^{-x_i}}$$

- The first is a 'hyperbolic tangent' that ranges from – 1 to 1, while the other is 'logistic function' which is similar in shape but ranges from 0 to 1.
- Here y_i is the output of the i^{th} node (i.e. neuron) and x_i is the weighted sum of the input connections.
- Alternative activation functions have also been mentioned, including 'the rectifier and softplus' functions.
- In recent developments of 'deep learning' the 'Rectified Linear unit (RcLU) is frequently used. It is one of the possible ways to overcome the numerical problems related to the sigmoids.

Layers

- The MLP consists of three or more layers (an input and an output layer with one or more hidden layers) of non-linearly-activating nodes.
- Since MLPs are fully connected, each node in one layer connects with a certain weight W_{ij} to every node in the following layer.

☞ Learning

- Learning occurs in the perceptron by changing connection weights after each piece of data is processed. It is based on the amount of error in the output compared to the expected result.
- This is 'supervised learning' and it is carried out through back propagation. The back-propagation is a generalisation of the 'least mean square algorithm' in the linear perceptron.
- We denote the degree of error in an output node j in the n^{th} data point by $e_j(n) = d_j(n) - y_j(n)$, where $d_j(n)$ is the desired target value for n^{th} data point at node j, and $y_j(n)$ is the value produced by the perceptron at node j when the n^{th} data point is given as an input.
- The node weights are adjusted based on corrections that minimise the error in the entire output for the n^{th} data point, given by

$$E(n) = \frac{1}{2} \sum_{\substack{\text{output} \\ \text{node j}}} e_j^2 (n)$$

- We use gradient descent to note the change in each weight W_{ij} as :

$$\Delta W_{ij}(n) = -\eta \frac{\partial E(n)}{\partial V_j(n)} y_i(n),$$

Where $y_i(n)$ is the output of the previous neuron i, and η is the 'learning rate'.

- The learning rate ensures that the weights, quickly converge to a response without oscillations. $\frac{\partial E(n)}{\partial V_j(n)}$ denotes the partial derivative of the error E(n) according to the weighted sum $V_j(n)$ of the input connections of neuron i.

☞ Applications

(i) MLPs are useful in research for their ability to solve problems, they give approximate solutions for extremely complex problems like 'fitness approximation'.

(ii) MLPs can be used to create mathematical models by regression analysis. MLPs make good classifier algorithms because 'classification' is a particular case of regression when the response variable is categorical.

(iii) MLPs find applications in diverse fields such as 'speech recognition', 'image recognition', and 'machine translation' software.

1.4 SIGMOID NEURONS

There are three major types of neurons that are used in practice and they introduce 'non-collinearity' in their computations.

The first of these is the 'Sigmoid Neuron', and that uses the function

$$f(x) = \frac{1}{1 + e^{-\lambda x}} : \lambda \text{ is steepness parameter.}$$

The function indicates that when the logit is very small, the output of a logistic neuron is very close to zero, but when the logit is very large, the output of the logistic neuron is close to 1. In between these two extremities, the neuron assumes an 'S-shape', as shown in the Fig. 1.4.1.

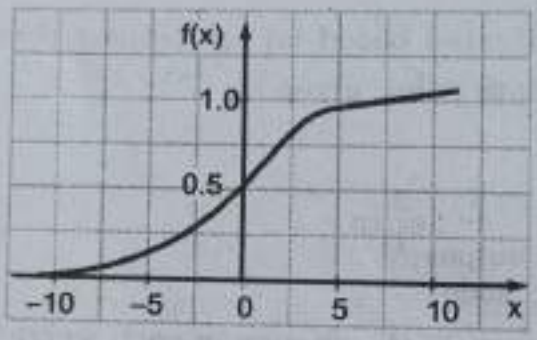


Fig. 1.4.1 : The output of a sigmoid neuron as x-varies

Note :

(I) Tanh neurons use a similar kind of S-shaped non-linearity, but instead of ranging from 0 to 1, the output tanh neurons ranges from – 1 to 1.

Here, we use $f(x) = \tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

The resulting relationship between the output and logit x is, shown in the Fig. 1.4.2.

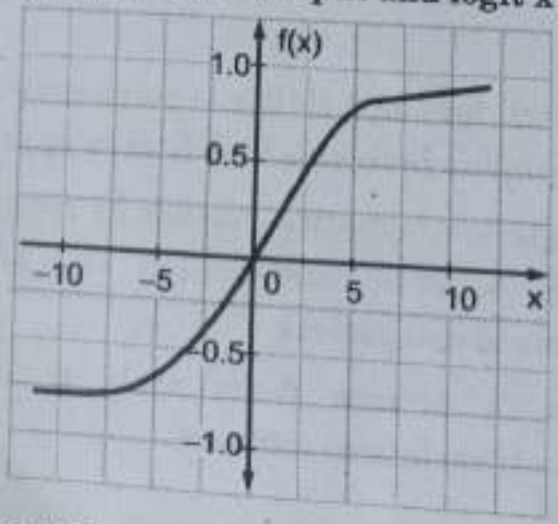


Fig. 1.4.2 : The output of a tanh neuron as x-varies

(II) Sigmoid functions are so called because their graphs are 'S-shaped'.
Sigmoids are odd, monotonic functions of one-variable.

1.5 GRADIENT DESCENT

- Gradient Descent is an optimization algorithm and it is used to train **machine learning models** and **neural networks.**
- Training the data helps these models learn over time and the cost function within gradient descent gauges its accuracy with each iteration of parameter updates.
- The model will continue to adjust its parameter so that possible error will be minimum. The happens when function become, near to zero.
- Once machine learning models are optimized for accuracy, they become powerful tools for computer science application.

1.5.1 Working of Gradient Descent

GQ. How gradient descent works ?

- Let us revise the concept of linear regression.
- The equation $y = mx + c$, represents a line with 'm ' as slope and 'c' as intercept on Y-axis.
- We can plot a scatter – diagram and find the line of best fit. It can calculate the error between the actual output and the predicted output, using the mean - squared formula.
- The gradient descent algorithm behaves similarly, but it is based on a convex function, such as :

  Point of convergence, i.e. where the cost function is at its minimum
- The starting point is chosen arbitrarily to evaluate the performance.
- From that starting point, we calculate derivative (or slope) and draw tangent line and note the steepness of the slope.
- The slope will yield the updates of the parameters i.e. the weight and bias.
- The slope at the starting point will be steeper, but as new parameters are generated, the steepness reduces till it reaches, the lowest point on the curve, known as the point of convergence.
- The aim of gradient descent is to minimize the cost function or the error between predicated and actual value.
- To do this, we require two data points : a direction and a learning rate.

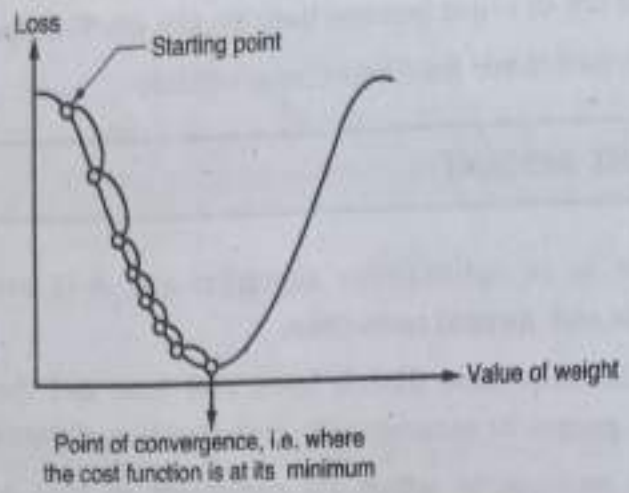
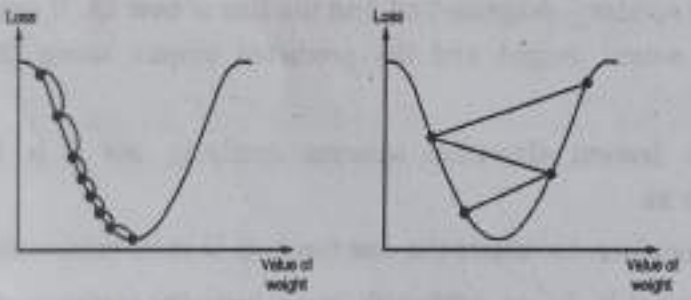


Fig. 1.5.1 : Gradient descent algorithm

- Using these factors, we can calculate partial derivatives of future iterations, and thereby we get local or global minimum i.e. point of convergence.
- **Learning rate :** This is also called as step size or alpha. This is the size of the steps to reach to the minimum. This is typically a small value and it is evaluate and updated, depending upon the behaviour of the cost function. High learning rates result in larger steps but the risk involved is minimum.
- Conversely, a low learning rate has small step sizes. The advantage of this method is gives more precision. Since the number of iterations are more, it takes more time and computations to reach the minimum.



(a) Small learning rate (b) Large learning rate

Fig. 1.5.2

1.5.2 The Cost (or loss) Function

- This function measures the difference, or error between actual y and predicted y at its current position.
- This provides feedback to the model so that it can adjust parameters to minimize the error and can find local or global minimum. This way model's efficacy is improved.
- It iterates continuously i.e. it moves along the direction of steepest descent, until the cost function is minimum.

- At this point, the model stops learning. Here, we note that there is a slight difference between cost function and cost function and loss function.
- A loss function refers to the error of one training example, while a cost function calculates the average error across an entire training set.

1.5.3 Types of Gradient Descent

There are three types of gradient descent learning algorithms :

(i) batch-gradient descent

(ii) stochastic-gradient descent and

(iii) mini-batch gradient descent

1.5.3(A) Batch Gradient Descent

- Batch gradient descent sums the error at each point in a training set, updating the model only after all training examples have been evaluated.
- This process is referred to as a training epoch.
- While this batching provides computation efficiency, it can have long processing time for large training datasets. It needs actually to store all of the data into memory. Batch descent method produces a stable error gradient and convergence, but the convergence point is not most ideal.

1.5.3(B) Stochastic Gradient Descent

- Stochastic gradient descent (SGD) runs a training epoch for each example in the dataset.
- It updates each training examples parameter one at a time.
- Since it is only one training example, they are easier to keep in memory.
- These frequent updates can offer more detail and also speed, it results in losses in computational efficiency when compared to batch gradient descent.
- Its frequent updates create noisy gradients, but this helps to find global minimum.

1.5.3(C) Mini-batch Gradient Descent

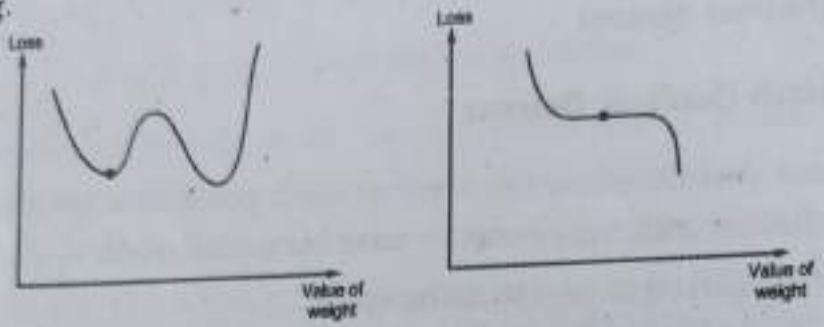
- Mini-batch gradient descent combines concepts from both batch gradient and stochastic gradient descent.
- It splits the training dataset into small batch sizes and performs updates on each of those batches. This approach makes a baloma between the computational efficiency of batch gradient descent and the speed of stochastic gradient descent.

1.5.4 Challenges with Gradient Descent

Gradient descent is the most common approach for optimization problems, but it has its own set of challenges.

For example : Local minima and saddle points

- For, convex problem, local minimum can be found easily. But for non-convex problems, gradient descent has to struggle to find global minimum.
- Note that when the slope of the cost function is at zero (or close to zero), the model stops learning.



(a) (b)

Fig. 1.5.3 : Value of weight

- At local minima and saddle points the slope of the cost function is zero. But the slope of cost function, increases on either side of the current point.
- Noisy gradients can help the gradient escape local minimum and saddle points.

1.5.5 Vanishing and Exploding Gradients

We can also come across two other problems when the model is trained with gradient descent and back propagation :

(i) **Vanishing gradient :** This occurs when the gradient is too small. As we go backwards during backpropagation, the gradient continues to become smaller, causing the earlier layers in the network to learn more slowly than later layers.

When this happens, the weight parameters update until they become insignificant - i.e. (almost) 0. This results in an algorithm which stops learning.

(ii) **Exploding gradients :** This happens when the gradient is too large. This creates an unstable model.

In this case, the model weights will grow too large.

One solution to this issue is to leverage a dimensionality reduction technique, and this helps to minimize the complexity within the model.

1.5.6 Method of Resolving the Vanishing Oradiert Problem

There are various methods that help in overcoming the vanishing gradient problems :

(1) Multi-level hierarchy

(2) The long-short term memory

(3) Residual neural network

(4) ReLU

1.5.6(A) Multi-level Hierarchy

It is a simple method, it trains one level at a time and fine - tunes the level by backpropagatio. So that every layer learns a compressed observation which goes ahead for the next level.

1.5.6(B) Long-Short Term Memory (LSTM)

- A simple LSTM helps the gradient size to remain constant.
- The activation function we use in the LSTM often works as an identify function.
- Hence in gradient back-propagation the size of the gradient does not vanish.
- From the Fig. 1.5.4 the effective weight of the gradient is equal to the forget gate activation.

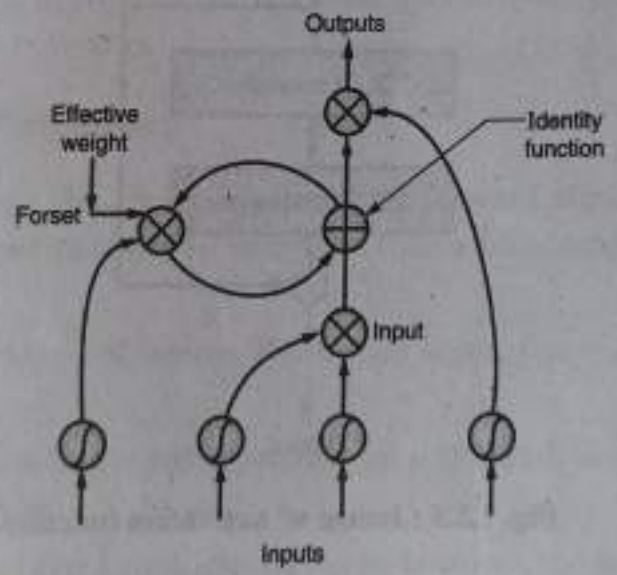


Fig. 1.5.4 : Image of gradient back-propagation

- Hence, if the **forget gate** is on (i.e.) activation close to 1.01, then the gradient does not vanish.
- Therefore, LSTM is one of the best options to deal with long-range dependences. Especially, in the recurrent neural network.

1.5.7 Residual Neural Network

- Residual connections in the neural network make the model learn well and the batch normalization feature makes sure that gradients will not vanish.
- These batch normalization features are obtained by the skip connection.
- The skip or bypass connection is useful in any network to bypass data from a few layers. Using these connection, information can be transferred from layer in to layer n + t.
- We connect the activation function of layer n to the activation function of n + t.
- This causes the gradient to pass between the layers without any modification in size. We exhibit the image :

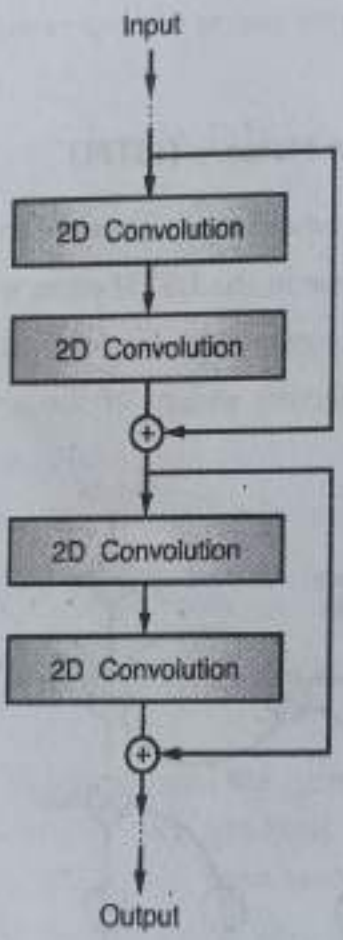


Fig. 1.5.5 : Image of Activation function

1.5.8 Rectified Linear Unit (ReLU) Activation Function

- ReLU is an activation function, like a sigmoid and tanh activation function but better than them.
- The basic functions for ReLU is given by,

$$f(x) = \max(0, x)$$

- Here gradient is one when the output of the function is > 0.

1.6 FEEDFORWARD NEURAL NETWORKS

1.6.1 Feed Forward Neural Networks (FFNN)

- A feed forward neural network is an artificial neural network where in connections between the nodes do not form a cycle.
- As such it is different from it's descendent : random neural network. The feed forward neural network was the first and simplest type of neural network device.
- In an artificial feed forward network (FFN), information always moves in one direction, i.e. it never goes backwards.
- A (FNN) is an artificial neural network (ANN). FFNN is a biologically inspired classification algorithm. It contains organised layers, consisting of a number of simple neuron – like processing units.
- Every unit in a layer is connected with all the units in the previous layer. These connections are not all equal. Each connection may have a different weight.
- The weights on these connections encode the knowledge of a network. Often the units in a neural network are called 'nodes'.
- Data enters at the inputs and passes through the network, layer by layer, until it arrives at the outputs.
- When it acts as a classifier, there is no feedback between layers. Hence they are called as feed forward neural networks.

1.6.2 Multi-layer Perceptron

- A multi-layer perceptron (MLP) is a class of feed forward algorithm neural network. The terms multiple-layer perceptron consists of networks compose of multiple layers of perceptron.
- MLP consists of three types of layers- the initial layer, the output layer and hidden layer.
- The input layer receives the input signal to be processed. And the classification is performed by output layer.
- The (number of) hidden layers that are placed in between the input and output layers carry the computational work of MLP.
- Similar to feed forward network (FFN) in MLP the data flows in the forward direction form input to output layer. The neurons in the MLP are trained with the back propagation algorithm. The major use of MLP are pattern classification and recognition.
- Multilayer perceptron and CNN are two fundamental concepts in machine learning.

1.6.3 Convolution Neural Network (CNN)

- A CNN is a types of artificial neural network used in major recognition. CNNs are a category of neural networks that have proven very effective in areas such as image recognition and classification.
- Convolutional networks have been successful in identifying faces, objects and traffic signs apart from powering vision in robots and self-driving cars.
- CNNs are important tools for most machine learning practitioners today.
- CNN is a deep learning neural network designed for processing structured arrays of data such as images.

1.6.4 Recurrent Neural Network (RNN)

- Recurrent neural network is a class of artificial neural network where connections between nodes form a directed graph along a temporal sequence. This makes it to show temporal dynamic behaviour.
- It uses sequential data or time series data. These data are commonly used for ordinal or temporal problems, such as languages translation, natural language processing, speech recognition and image captioning, they are incorporated into popular applications such as voice search and Google translation.
- Like CNNs, recurrent neural networks utilise training data by learning. They are distinguished by their 'memory' as they take information from prior inputs to influence the current input and output.
- While traditional deep neural networks assume that inputs and outputs are independent of each other.
- The output of recurrent neural networks depends on the prior elements within the sequence, while future events would be also helpful in determining the output of a given sequence, unidirectional recurrent neural networks cannot account for these events in their predictions.

1.7 REPRESENTATION POWER OF FEEDFORWARD NEURAL NETWORKS

- Single neurons are not nearly expressive enough to solve complicated learning problems. That is why our brain is made up of more than one neuron.
- It is not possible for a single neuron to differentiate handwritten digits. The neurons in the human brain are organised in layers. Information flows from one layer to another until sensory input is converted into conceptual understanding.
- This information is processed by each layer and passed on to the next until, we conclude whether we are looking at a cat or an aeroplane.

- Borrowing from these concepts, we can construct an 'artificial neural network'. A neural network comes about when we start looking up neurons to each other, the input data, and to the output nodes, which correspond to the network's answer to a learning problem.
- The bottom layer of the network pulls in the input data. The top layer of neurons (output nodes) computes our final answer.
- The middle layers of neurons are called 'hidden layers'.
- We note the following points :

(i) When the neural net tries to solve the problem, actually hidden layers come into the picture to resolve the problem.

Oftentimes, taking a look at the activities of hidden layers can tell use a lot about the features of network has automatically learned to extract from the data.

(ii) Often, hidden layers have fewer neurons than the input layer to force the network to learn compressed representation of the original input.

For example, while our eyes obtain raw pixel values from the surroundings, our brain thinks in terms of edges and contours.

It is because the hidden layers of biological neurons in our brain force us to come up with better representations that we perceive.

(iii) It is not that every neuron has its output connected to the inputs of all neurons in the next layer. Actually, selecting which neurons to connect to which other neurons in the next layer is an art that comes from experience.

(iv) The inputs and outputs are 'vectorised' representations.

We can also mathematically express a neural network as a series of vector and matrix operations.

For example, let us consider the input to the i^{th} layer of the network to be a vector $x = [x_1, x_2, \ldots, x_n]$. We want to find a vector

$y = [y_1, y_2, \ldots, y_n]$ produced by the input through the neurons.

- We can express this as a simple matrix multiply if we construct a weight matrix W of size $n \times m$ and a bias vector of size m.
- In this matrix, each column corresponds to a neuron, where the j^{th} element of the column corresponds to the weight of the connection pulling in the j^{th} element of the input.
- Thus, we have $y = f(W^T x + b)$. This reformulation is not that easy when we begin to implement these networks in software.

1.8 THREE CLASSES OF DEEP LEARNING

1.8.1 Transfer Learning

- The reuse of a previously learned model on a new problem is known as transfer learning. Due to its ability to train deep neural networks with a minimal amount of data, it is now quite popular in deep learning.
- This is helpful in the field of data science because most real-world situations do not demand for millions of labelled data points to train complex models.
- In transfer learning, a machine leverages the information gained from a previous assignment to improve prediction about a new task. For example, while training a classifier to determine whether an image contains food, you may use the knowledge you learned to distinguish beverages.
- Through transfer learning, the expertise of a machine learning model that has already been trained is applied to an unrelated but closely related problem. For instance, if you trained a straightforward classifier to determine whether a picture has a backpack, you might use that knowledge to recognize other things, like sunglasses.

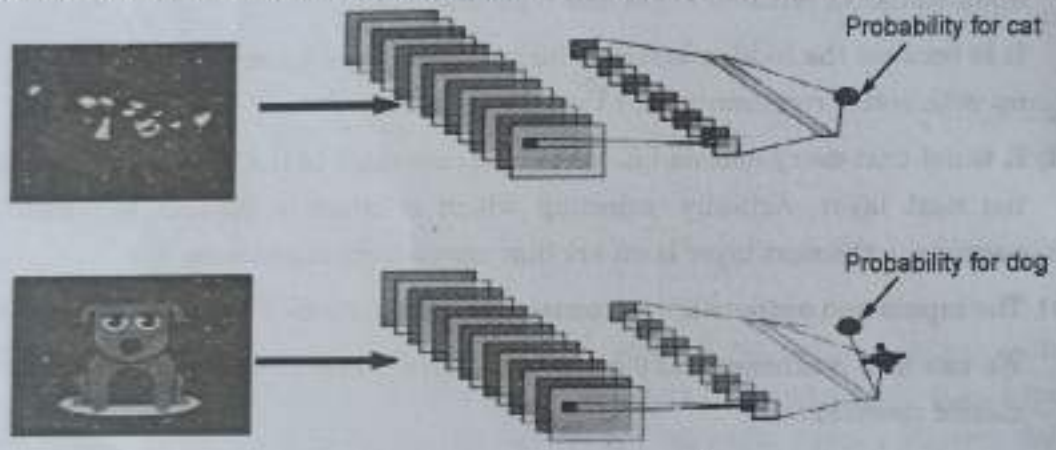


Fig. 1.8.1 : Transfer Learning Example

- With transfer learning, we essentially attempt to apply the knowledge we have gained to new situations to comprehend the concepts more fully.
- Weights are automatically transferred from a network that completed the new "task B" to a network that was conducting "task A."
- Because of the massive amount of CPU power required, transfer learning is typically applied in computer vision and natural language processing tasks like sentiment analysis.

1.8.2 How to Use Transfer Learning ?

- You can use transfer learning on your own predictive modeling problems.
- Two common approaches are as follows:

1. Develop Model Approach
2. Pre-trained Model Approach

1. Develop Model Approach

(a) **Select Source Task :** You must select a related predictive modeling problem with an abundance of data where there is some relationship in the input data, output data, and/or concepts learned during the mapping from input to output data.

(b) **Develop Source Model :** Next, you must develop a skillful model for this first task. The model must be better than a naive model to ensure that some feature learning has been performed.

(c) **Reuse Model :** The model fit on the source task can then be used as the starting point for a model on the second task of interest. This may involve using all or parts of the model, depending on the modeling technique used.

(d) **Tune Model :** Optionally, the model may need to be adapted or refined on the input-output pair data available for the task of interest.

2. Pre-trained Model Approach

(a) **Select Source Model :** A pre-trained source model is chosen from available models. Many research institutions release models on large and challenging datasets that may be included in the pool of candidate models from which to choose from.

(b) **Reuse Model :** The model pre-trained model can then be used as the starting point for a model on the second task of interest. This may involve using all or parts of the model, depending on the modeling technique used.

(c) **Tune Model :** Optionally, the model may need to be adapted or refined on the input-output pair data available for the task of interest.

This second type of transfer learning is common in the field of deep learning.

1.8.3 Types of Transfer Learning

1. Domain Adaptation

- Domain adaptation is usually referred to in scenarios where the marginal probabilities between the source and target domains are different, such as

$$P(X_s) \neq P(X_t).$$

- There is an inherent shift or drift in the data distribution of the source and target domains that requires tweaks to transfer the learning.
- For instance, a corpus of movie reviews labeled as positive or negative would be different from a corpus of product-review sentiments.
- A classifier trained on movie-review sentiment would see a different distribution if utilized to classify product reviews.
- Thus, domain adaptation techniques are utilized in transfer learning in these scenarios.

2. Domain confusion

- Different layers in a deep learning network capture different set of features.
- We can utilize this fact to learn domain-invariant features and improve their transferability across domains.
- Instead of allowing the model to learn any representation, we nudge the representations of both domains to be as similar as possible. This can be achieved by applying certain preprocessing steps directly to the representations themselves.
- The basic idea behind this technique is to add another objective to the source model to encourage similarity by confusing the domain itself, hence *domain confusion*.

3. Multitask learning

- Multitask learning is a slightly different flavor of the transfer learning world. In the case of multitask learning, several tasks are learned simultaneously without distinction between the source and targets.
- In this case, the learner receives information about multiple tasks at once, as compared to transfer learning, where the learner initially has no idea about the target task. This is depicted in the following diagram:

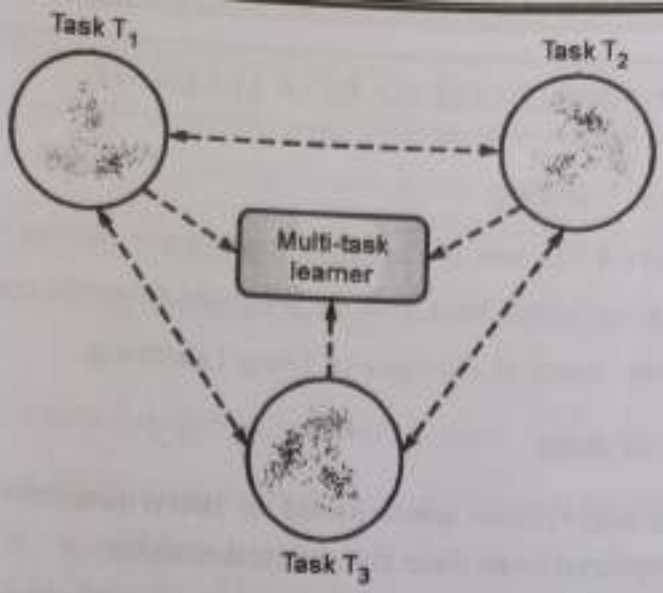


Fig. 1.8.2 : Multitask learning : Learner receives information from all tasks simultaneously

4. One-shot learning

- Deep learning systems are data hungry by nature, such that they need many training examples to learn the weights. This is one of the limiting aspects of deep neural networks, though such is not the case with hum an learning.
- For instance, once a child is shown what an apple looks like, they can easily identify a different variety of apple (with one or a few training examples); this is not the case with ML and deep learning algorithms.
- One-shot learning is a variant of transfer learning where we try to infer the required output based on just one or a few training examples. This is essentially helpful in real-world scenarios where it is not possible to have labelled data for every possible class (if it is a classification task) and in scenarios where new classes can be added often.

5. Zero-shot learning

- Zero-shot learning is another extreme variant of transfer learning, which relies on no labelled examples to learn a task. This might sound unbelievable, especially when learning using examples is what most supervised learning algorithms are about.
- Zero-data learning, or zero-short learning, methods make clever adjustments during the training stage itself to exploit additional information to understand unseen data.
- For example, take *zero-shot learning* as a scenario where three variables are learned, such as the traditional input variable, x, the traditional output variable, y, and the additional random variable that describes the task, T. The model is thus trained to learn the conditional probability distribution of $P(y \mid x, T)$.
- Zero-shot learning comes in handy in scenarios such as machine translation, where we may not even have labels in the target language.

1.9 BASIC TERMINOLOGIES OF DEEP LEARNING

Challenges of Deep learning

In the words of Andrew Ng, one of the most prominent names in Deep Learning :

"I believe deep learning is our best shot at progress towards real AI".

Here, we discuss prominent challenges in Deep Learning.

1. Lots and lots of data

- Deep Learning algorithms are trained to learn progressively using data. Large datasets are required to deliver the desired results.
- The more powerful abstraction you want, the more parameters need to be tuned and more parameters require more data.
- The complexity of a neural network can be expressed through the number of parameters. In case of deep neural networks, this number can be in the range of millions, or billions.
- Let us call this number as p. Since we want the model's ability to generalise, the number of data points is at least (P * P).

2. Over fitting in Neural Networks

- The efficiency of a model is judged by its ability to perform well on an unseen data set and not by its performance on the training data fed to it.
- The model memorises the training examples but does not learn to generalize to new situations and data set.

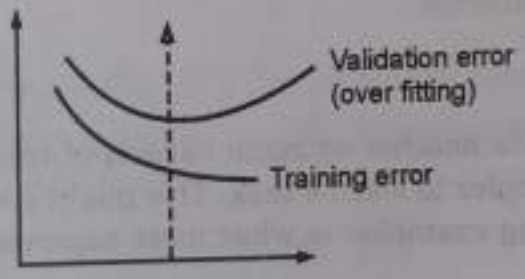


Fig. 1.9.1

3. Hyper parameter Optimisation

- Hyper parameters are the parameter whose value is defined prior to the commencement of the learning process.
- Relying on the default parameters and not performing Hyper parameter Optimisation can have a significant impact on the model performance.
- Changing the value of such parameters by a small amount can make a large change in the performance of the model.

4. Requires high-performance hardware

- Training a dataset for a Deep Learning solution requires a lot of data.
- To ensure better efficiency and less time consumption, data scientists switch to multi-core high performing GPUS and similar processing units. These processing units are costly and consume a lot of power.
- Deploying deep learning solution to the real world becomes a costly and power consuming affair.

5. Neural Networks are essentially a Balck box

- We feed known data to the neural networks and how they are put together.
- But we usually do not understand how they arrive at a particular solution. Neural networks are essentially Black boxes.

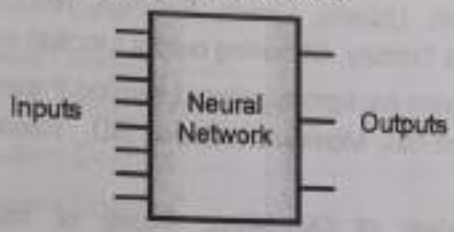


Fig. 1.9.2

- The lack of ability of neural networks for reason on an abstract level makes it difficult to implement high-level cognitive functions. Also, their operation is largely invisible to humans, so verification process does not work.
- But professor Murray shanahan, professor of Cognitive Robotics, has presented a paper which mentions advancements in solving a fore mentioned hurdles.

6. Lack of flexibility and Mulittasking

- Deep learning models, once trained, can deliver tremendously efficient and accurate solution to a specific problem. In the current landscape, the neural network architectures are highly specialized to specific domains of application.
- Google DeepMind's Research Scientist Raia Hadsell summed it up:
- "There is no neural network in the world, and no method right now that can be trained to identify objects and images, play space invaders, and listen to music."
- Most of the systems are good at solving one problem. Even solving a very similar problem requires retraining and reassessment.
- But there are advancements in this aspect using progressive Neural Networks. Also there is significant progress towards Multi Task Learning (MTL).

Chapter Ends ...

❑❑❑

MODULE 2

CHAPTER 2

Training, Optimization and Regularization of Deep Neural Network

Syllabus

Training Feedforward DNN : Multi Layered Feed Forward Neural Network, Learning Factors, Activation functions: Tanh, Logistic, Linear, Softmax, ReLU, Leaky ReLU, Loss functions: Squared Error loss, Cross Entropy, Choosing output function and loss function

Optimization : Learning with backpropagation, Learning Parameters : Gradient Descent (GD), Stochastic and Mini Batch GD, Momentum Based GD, Nesterov Accelerated GD, AdaGrad, Adam, RMSProp

Regularization : Overview of Overfitting, Types of biases, Bias Variance Tradeoff Regularization Methods: L_1, L_2 regularization, Parameter sharing, Dropout, Weight Decay, Batch normalization, Early stopping, Data Augmentation, Adding noise to input and output.

Training Feedforward DNN

2.1 MULTILAYER FEED-FORWARD NEURAL NETWORK

- This network is made up of multiple layers. The architecture of this class consists of one or more intermediary layers called as hidden layers, besides having an input and an output layer.
- The hidden layers perform intermediary computations before directing the input to the output layer.
- The input layer neurons are linked to the hidden layer neurons and the weights on these links are called as **input-hidden layer weights.**
- Again, the hidden layer neurons are linked to the output layer neurons and the corresponding weights are referred to as **hidden-output layer weights.**

2.1.1 Perceptron Training Algorithm for Multiple Output Classes

We mention below the Algorithm of perceptron training for multiple output classes :

- **Step (I) :** Nitialise the weights, biases and learning rate conveniently.
- **Step (II) :** Check for stopping condition; if it is not true, perform steps III-VII.
- **Step (III) :** For each bipolar or binary training vector pair S : t, perform steps IV–VI.
- **Step (IV) :** Set activation function (identity function) of each input unit i = 1 to n : $x_i = s_i$
- **Step (V) :** Calculate the net input as,

$$y_{inj} = b_j + \sum_{i=1}^{n} x_i w_{ij}$$

Then calculate output response of each output unit j = 1 to m : To calculate the output response, apply activation formula over the net input :

$$y_j = f(y_{inj}) = \begin{cases} 1, & \text{if } y_{inj} > \theta \text{ ; } \theta \text{ is threshold valve} \\ 0, & \text{if } -\theta \le y_{inj} \le \theta \\ -1, & \text{if } y_{inj} < -\theta \end{cases}$$

- **Step (VI) :** If $t_j \neq y_j$, then make adjustment in weights and bias for j = 1 to m and i = 1 to n.

i.e. $w_{ij}\,(new) = w_{ij}\,(old) + \alpha\, t_j\, x_i$

$b_j\,(new) = b_j\,(old) + \alpha\, t_j$

Otherwise we have

w_{ij} (new) = w_{ij} (old)

b_j (new) = b_j (old)

- **Step (VII) :** If there is no change in weights then stop the training process, otherwise begin from step III. We exhibit the architecture for the above algorithm :

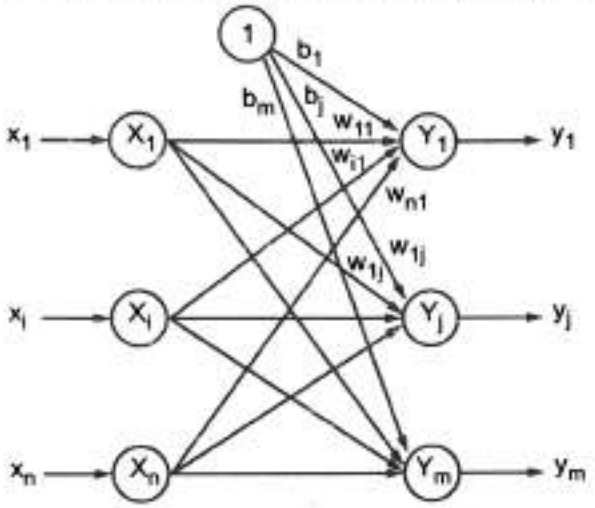


Fig. 2.1.1 : Network architecture for perceptron network for several output classes

2.2 LEARNING FACTORS

(i) Learning rate α determines the size of the weight adjustments made at each iteration and hence influences the rate of convergence.

(ii) Poor choice of the rate can result in a failure in convergence.

It is convenient to keep the rate constant through all the iterations for best results. If the learning rate α is too large, the search path will oscillate and converges more slowly than a direct descent. The range of α from 10^{-3} to 0.9 is optimistic and has been used successfully for several back-propagation algorithmic experiments. If the rate is too small, the descent will progress in small steps, increasing the time to converge. Jacobs has suggested the use of adaptive coefficient where the value of the learning rate is the function of error derivative on successive updates.

2.2.1 Calculation of Error

Consider any r^{th} output neuron.

Let 'O' be the output for which the target output is 'T'.

The **error norm** in output for the r^{th} output neuron is given by

$$E_r^1 = \frac{1}{2} e_r^2 = \frac{1}{2}(T - 0)^2 \quad ...(i)$$

Where 'e_r' is the error in the r^{th} neuron. Error may be positive or negative. To consider only positive values, we consider the square of the error, i.e., we consider only absolute valve.

For the **first training pattern,** the Euclidean norm of error E^1 is given by

$$E^1 = \frac{1}{2} \sum_{r=1}^{n} (T_{or} - D_{or})^2 \quad ...(ii)$$

If we consider error function for all the training patterns, we get

$$E(V, W) = \sum_{j=1}^{n \text{ set}} E^j (V, W, I), \text{ where}$$

E is the error function depending on the m (1 + n) weights of W and V. Based on error signal, neural network modifies its synaptic connections to improve the system performance.

2.3 ACTIVATION FUNCTION

RQ. Explain common activation functions used in neural network.

(Ref - Q. 1(b), May 2011, 5 Marks)

(1) A model of the behaviour of a neuron can be presented as shown in Fig. 2.3.1. Here x_1, x_2, ..., x_n are the n-inputs to the artificial neuron. w_1, w_2, ..., w_n are the weights attached to the input links.

(2) **Biological** neuron receives all inputs through the dendrites, sums them and produces an output. If the sum is greater than a threshold value. The input signals are passed on to the cell body through the synapse which may accelerate or retard an arriving signal.

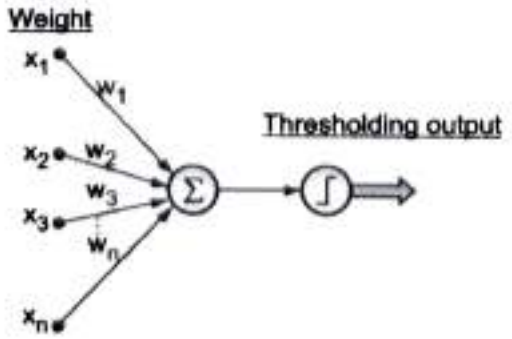


Fig. 2.3.1 : Simple model of an artificial neuron

(3) Here weights model the acceleration or retardation of the input signals. In short, weights are multiplicative factors of the inputs.

(4) The total input (say I) received by the soma (body) of the artificial neuron is

$$I = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n \quad \ldots(i)$$

$$= \sum_{i=1}^{n} w_i x_i$$

(5) This sum is passed on to a non-linear filter ϕ called Activation function, or Transfer function, or Squash Function which releases the outputs.

$$\text{i.e. } y = f(I) \quad \ldots(ii)$$

2.3.1 Derivation of Activation Functions

RQ. What are the important properties of activation function used in neural networks ?
(Ref - Q. 1(c), May 2016, 5 Marks)

RQ. List the different activation functions used in neural network.
(Ref - Q. 1(d), May 2017, 5 Marks)

RQ. With mathematical list four different activation functions used in neurons.
(Ref - Q. 1(e), May 2019, 5 Marks)

The obtain exact output, the activation function is applied over the net input of an ANN.

There are several activation functions. Here we consider a few :

(I) **Linear function :** It is defined as

$$f(x) = x \quad \text{for all } x$$

Here input = output

(1) Linear function

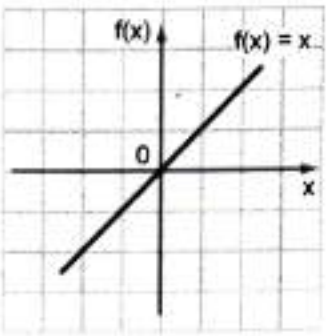


Fig. 2.3.2 : Linear function

(2)

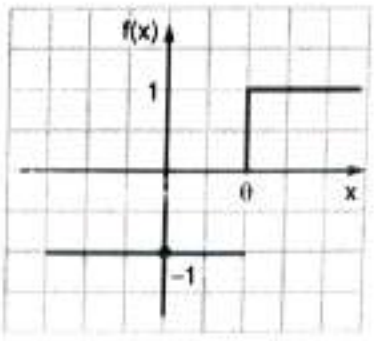


Fig. 2.3.3 : Linear function

(II) **Bipolar Step function :** The function is defined as :

$$f(x) = \begin{cases} 1, & \text{if } x \geq \theta \\ -1, & \text{if } x < \theta \end{cases}$$

Where θ is threshold value. The function is also used in single-layer nets to convert the net input to an bipolar output (+ 1, – 1).

(III) **Binary step function :** The function is defined as :

$$f(x) = \begin{cases} 1, & \text{if } x \geq \theta \\ 0, & \text{if } x < \theta \end{cases}$$

This function is used in single-layer nets to convert the net input to an output that is binary (0 or 1).

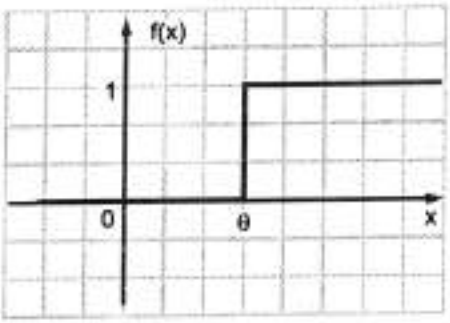


Fig. 2.3.4 : Binary step function

(IV) **Sigmoidal function :** This function is used in back-propagation nets. They are of two types :

(i) **Binary sigmoidal function :** It is also called as unipolar sigmoid function or a logistic sigmoid function, and defined as $f(x) = \frac{1}{1 + e^{-\lambda x}}$, where λ is steepness parameter

The derivative of f(x) is :

$$f'(x) = \frac{-1}{[1 + e^{-\lambda x}]^2} \cdot (-\lambda e^{-\lambda x}) = \frac{\lambda (e^{-\lambda x})}{[1 + e^{-\lambda x}]^2} = \frac{\lambda [1 + e^{-\lambda x} - 1]}{[1 + e^{-\lambda x}]^2}$$

$$= \lambda \left\{ \frac{1}{(1 + e^{-\lambda x})} - \frac{1}{(1 + e^{-\lambda x})^2} \right\}$$

$$= \lambda \left[\frac{1}{(1 + e^{-\lambda x})} \left\{ 1 - \frac{1}{(1 + e^{-\lambda x})} \right\} \right] = \lambda [f(x) (1 - f(x))] = \lambda f(x) [1 - f(x)]$$

Range of this sigmoid function is 0 to 1 [$\because$ denominator is always greater than or equal to 1]

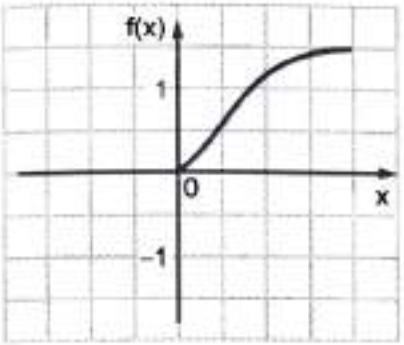


Fig. 2.3.5(i) Binary sigmoidal function

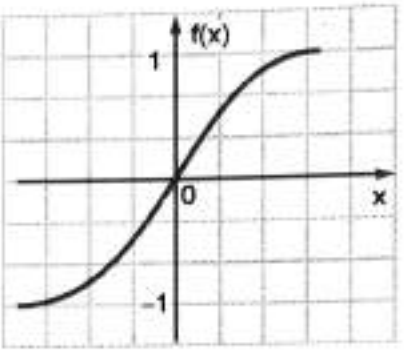


Fig. 2.3.5(ii) Bipolar sigmoidal function

(ii) Bipolar sigmoid function :

The function is given by

$$f(x) = \frac{2}{1+e^{-\lambda x}} - 1 = \frac{2-1-e^{-\lambda x}}{1+e^{-\lambda x}} = \frac{1-e^{-\lambda x}}{1+e^{-\lambda x}}$$

where again, λ is steepness parameter and the range is between – 1 and + 1.

Derivative of f(x) is :

$$f'(x) = \frac{[1+e^{-\lambda x}]\,[\lambda e^{-\lambda x}] - (1-e^{-\lambda x})(-\lambda e^{-\lambda x})}{(1+e^{-\lambda x})^2}$$

$$= \frac{\lambda e^{-\lambda x}[1+e^{-\lambda x}+1-e^{-\lambda x}]}{(1+e^{-\lambda x})^2} = \frac{2\lambda e^{-\lambda x}}{(1+e^{-\lambda x})^2}$$

$$= \frac{\lambda}{2}\left[\frac{4e^{-\lambda x}}{(1+e^{-\lambda x})^2}\right] = \frac{\lambda}{2}\left[\frac{(1+e^{-\lambda x})^2-(1-e^{-\lambda x})^2}{(1+e^{-\lambda x})^2}\right]$$

$$= \frac{\lambda}{2}\left[1-\left(\frac{1-e^{-\lambda x}}{1+e^{-\lambda x}}\right)^2\right] = \frac{\lambda}{2}[1-f(x)^2]$$

$$= \frac{\lambda}{2}[(1+f(x))(1-f(x))] \qquad [\because\ a^2-b^2=(a+b)(a-b)]$$

Remark

(1) Here, $f(x) = \frac{1-e^{-\lambda x}}{1+e^{-\lambda x}} = \tanh\left(\frac{\lambda x}{2}\right)$

$$\therefore\ f'(x) = \frac{\lambda}{2}\operatorname{sech}^2\left(\frac{\lambda x}{2}\right) = \frac{\lambda}{2}\left[1-\tanh^2\left(\frac{\lambda x}{2}\right)\right]$$

$$= \frac{\lambda}{2}\left[\left(1+\tanh\frac{\lambda x}{2}\right)\left(1-\tanh\frac{\lambda x}{2}\right)\right]$$

$$f'(x) = \frac{\lambda}{2}[(1+f(x))(1-f(x))]$$

(2) Sigmoid functions are so-called because their graphs are "S-shaped".

(i) Simple sigmoids defined to be odd, are monotone functions of one variable,

(ii) Hyperbolic sigmoids are subset of simple sigmoids and natural generalisation of the hyperbolic tangent function.

(3) Ramp function

$$\text{It is defined as } f(x) = \begin{cases} 1, & \text{if } x > 1 \\ x, & 0 \le x \le 1 \\ 0, & x < 0 \end{cases}$$

(V) Ramp function

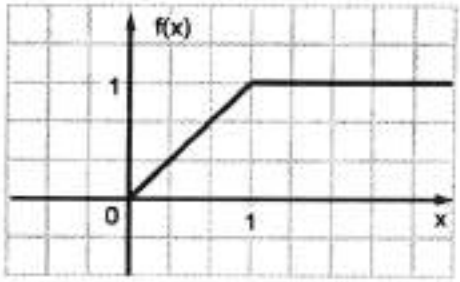


Fig. 2.3.6

(VI)Tanh Activation is an activation function used for neural networks

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \tanh(x)$$

- The tanh function became preferred over the 'sigmoid function' as it gave better performance for multi-layer neural networks.
- But it does not solve the vanishing gradient problem that sigmoid suffered, which was tackled more effectively with the introduction of ReLu activation.

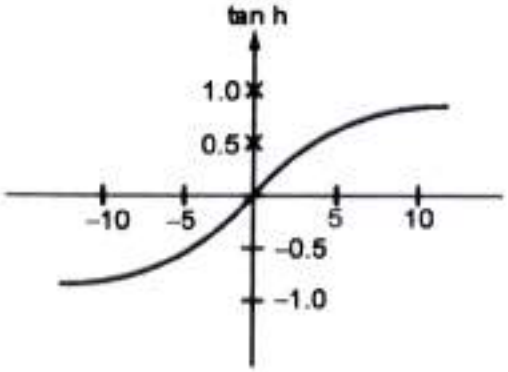


Fig. 2.3.7

(VII) Hard Tanh

Hard tanh is an activation function used for neural networks :

$$f(x) = -1, \quad \text{if } x < -1$$

$$= x \qquad \text{if } -1 \le x \le 1$$

$$= -1 \qquad \text{if } x > 1$$

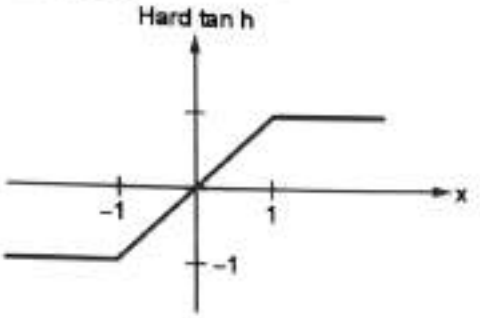


Fig. 2.3.8

(VIII) Softmax

- The softmax function is also known as softargmax or normalised exponential function, it converts a vector of K real numbers into a probability distribution of K possible outcomes.
- It is a generalisation of the logistic function' to multiple dimensions, and is used in 'multinomial logistic regression.'
- The softmax function is often used as the last 'activation function' of a neural network to normalise the output of a network to a probability distribution over predicted output classes.
- The formula is :

$$\sigma\left(\bar{z}\right)_i = \frac{e^{Z_i}}{\sum_{j=1}^{K} e^{Z_j}}$$

Where

σ = Softmax;

$\bar{z}$ = Input vector

e^{Z_i} = Standard exponential function for input vector

K = Number of classes in the multi-class classifier

e^{Z_j} = Standard exponential function for output vector

Softmax is used in neural network

- Softmax assigns decimal probabilities to each class in a multi-class problem.
- Those decimal probabilities must add to 1.0. This additional constraint helps training converge more quickly than it otherwise would.

- Softmax is implemented through a neural network layer just before the output layer.

2.3.2 Illustrative Examples for Logistic Function

Ex. 2.3.1 : Logistic function is given by, $F(v) = \frac{1}{1 + \exp(-av)}$

Show that the derivative of f(v) w.r.t. v is given by, $\frac{df}{dv} = af(v)\,[1 - f(v)]$

What is the value of this derivative at the origin ?

Soln. :

Step (I) : We have, $f(v) = \frac{1}{1 + \exp(-av)}$

Different w.r.t. v ; we get

$$f(v) = \frac{-1}{[1 + \exp(-av)]^2}\,[\exp(-av)(-a)]$$

$$= \frac{a \exp(-av)}{[1 + \exp(-av)]^2} = \frac{a}{[1 + \exp(-av)]} \frac{\exp(-av)}{[1 + \exp(-av)]}$$

$$= a\,f(v)\left[\frac{\exp(-av)}{[1 + \exp(-av)]}\right] = a\,f(v)\left[1 - 1 + \frac{\exp(-av)}{1 + \exp(-av)}\right]$$

$$= a\,f(v)\left[1 - \left\{1 - \frac{\exp(-av)}{1 + \exp(-av)}\right\}\right]$$

...(adding 1, subtracting 1)

$$= a\,f(v)\left[1 - \left\{\frac{1 + \exp(-av) - \exp(-av)}{1 + \exp(-av)}\right\}\right] = a\,f(v)\left[1 - \left\{\frac{1}{1 + \exp(-av)}\right\}\right]$$

$$\therefore f'(v) = a\,f(v)\,[1 - f(v)]$$

Step (II) : As $v \to 0$; $\exp(-av) = \exp(0) = 1$

$$\therefore f(v) = \frac{1}{1+1} = \frac{1}{2}$$

$\therefore$ f'(v) at v = 0 is, $= a \cdot \frac{1}{2}\left[1 - \frac{1}{2}\right] = \frac{a}{4}$

Ex. 2.3.2 : An odd sigmoid function is given by

$f(v) = \frac{1 - \exp(-av)}{1 + \exp(-av)} = v\left(\frac{av}{2}\right)$.

Show that the derivative of f(v) w.r.t. v is given by, $\frac{df}{dv} = \frac{a}{2}[1 - f^2(v)]$.

What is the valve of this derivative at the origin ? Suppose that the slope parameter a is made infinitely large. What is the resulting form of f(v).

☑ **Soln. :**

▸ **Step (I) :** For convenience we take, $f(v) = \tanh\left(\frac{av}{2}\right)$

Differentiating w.r.t. V;

$$\frac{df}{dv} = \operatorname{sech}^2\left(\frac{av}{2}\right) \cdot \frac{a}{2} = \frac{a}{2}\left[1 - \tanh^2\left(\frac{av}{2}\right)\right] \qquad \left[\because \operatorname{sech}^2 x = 1 - \tanh^2 x\right]$$

$$= \frac{a}{2}[1 - f^2(v)]$$

$$\therefore \frac{df}{dv} = \frac{a}{2}[1 - f^2(v)] \qquad \ldots(i)$$

▸ **Step (II) :** We have $f(v) = \tanh\left(\frac{av}{2}\right)$

At $v = 0$, $f(0) = \tanh(0) = 0$. ∴ At origin, from Equation (i), $\frac{df}{dv} = \frac{a}{2}[1 - 0] = \frac{a}{2}$

As $a \to \infty$, $\tan h(\infty) = 1$. ∴ As $a \to \infty$, $f(v) \to 1$.

Ex. 2.3.3 : The algebraic sigmoid function is given by, $f(v) = \frac{v}{\sqrt{1 + v^2}}$

Show that the derivative of f(v) w.r.t. v is given by, $\frac{df}{dv} = \frac{\phi^3(v)}{v^3}$

What is the value of this derivative at origin ?

☑ **Soln. :**

▸ **Step (I) :**

Differentiating $f(v) = \frac{v}{\sqrt{1 + v^2}}$; we get

$$f'(v) = \left[\frac{\sqrt{1 + v^2} \cdot 1 - v \cdot \frac{1 \cdot 2v}{2\sqrt{1 + v^2}}}{(1 + v^2)}\right] = \left[\frac{\sqrt{1 + v^2} - \frac{v^2}{\sqrt{1 + v^2}}}{(1 + v^2)}\right]$$

$$= \left[\frac{(1 + v^2) - v^2}{(1 + v^2)^{3/2}}\right]$$

$$\therefore f'(v) = \frac{1}{(1 + v^2)^{3/2}} \qquad \ldots(i)$$

$$\therefore f'(v) = \frac{1}{v^3}\left[\frac{v^3}{(1 + v^2)^{3/2}}\right] = \frac{1}{v^3}\left[\frac{v}{\sqrt{1 + v^2}}\right]^3 = \frac{1}{v^3}[f(v)]^3 = \frac{f^3(v)}{v^3} \qquad \ldots(ii)$$

▶ **Step (II) :** As $v \to 0$ $\quad f'(v) = \dfrac{1}{(1+v^2)^{3/2}} \to 1.$

$\therefore$ At origin, $f'(v) = 1$.

2.3.3 Re-LU

ReLU is rectified linear unit, is a non-linear activation function, used in multi – layer deep neural networks. This function is represented as :

$$f(x) = \max(0, x),$$

where x = an input value

An output is equal to zero when the input valve is negative and is equal to input value when input value is positive,

$$\text{i.e. } f(x) = \begin{cases} 0, & \text{if } x \le 0 \\ x, & \text{if } x > 0 \end{cases}$$ where x is an input value.

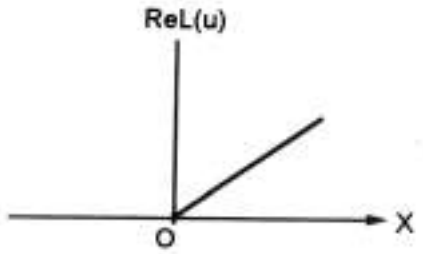


Fig. 2.3.9

- The rectified linear activation function overcomes the vanishing gradient problem, allowing models to learn faster and perform better.
- The rectified linear activation is the default activation when developing multilayer perception and convolutional neural networks.

2.4 LOSS FUNCTION

GQ. Explain loss function. Mention advantages and disadvantages.

- A loss function or cost function (sometimes also called an error function) is a function that maps an event or values of one or more variables onto a real number intuitively representing some 'cost' associated with the event.
- Loss functions are performed at the end of a neural network, comparing the actual and predicted outputs to determine the model's accuracy.

2.4.1 Loss Function for Regression

- Regression involves predicting a specific value that is continuous in nature.
- Estimating the price of a house or predicting stock prices are examples of regression because one works towards building a model that would predict a real-valued quantity.
  - (i) It is the mean of square of residuals all the datapoints in the dataset. Residuals is the difference between the actual and the predicted prediction by the model.
  - (ii) Squaring of residuals is done to convert negative values to positive values.
  - (iii) Squaring also gives more weightage to larger errors when the cost function is far away from its minimal value, squaring the error will penalise the model more and thus helping in reaching the minimal value faster.
- Mean Squared Error (MSE) is given by

$$MSE = \frac{1}{N} \sum_{i=1}^{N} \left(Y_i - \hat{Y}_i \right)^2$$

Advantages

(i) Easy to interpret,

(ii) Always differential because of the square,

(iii) Only one local minima.

Disadvantages

(i) Since error unit is in the square and hence it is not understood properly.

(ii) Not robust to outlier.

2.4.2 Loss Function for Classification

- This is the most common loss function used in classification problems. The cross-entropy loss decreases on the predicted probability converges to the actual label.
- It measures the performance of a classification model whose predicted output is a probability value between 0 to 1.
- 'Cross-entry and mean-squared error' are the two main types of loss functions to use when training neural network models.
- Neural Networks are trained using stochastic gradient descent and require that we choose a loss function while designing and configuring our model.
- Binary Cross-Entropy/Log loss is the most common loss function used in classification problems.

- The cross-entropy loss decreases as the predicted probability converges to the actual label.
- It measures the performance of a classification model whose predicted output is a probability value between 0 and 1.

2.4.3 Classification Loss in Object Detection

- The loss functions of object detection can be categorised as two types : The classification loss and the localisation loss.
- The former is applied to train the classify-head for determining the type of target object, and the latter is used to train another head for regressing a rectangular box to locate target object.
- Object detection is useful in identifying objects in an image or video. Below, the image on the left shows classification, in which the classes Bread and coffee are identical.
- The image on the right illustrates the object detection by surrounding the members of each class-Bread and Coffee-with a boundary box.

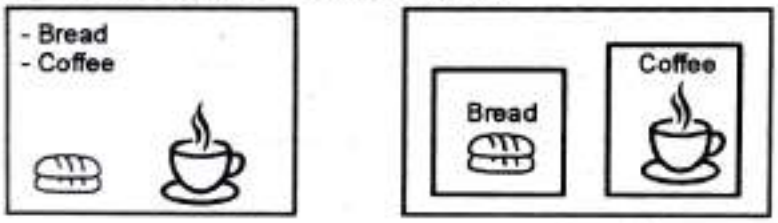


Fig. 2.4.1

- The most popular approaches to computer vision are classification and object detection to identify objects present in an image and specify their position.
- But many use cases for analysing images at a lower level than that. That is where image segmentation comes in.

2.4.4 Image Segmentation

- Any image consists of both useful and useless information, depending on the user's interest.
- Image segmentation separates an image into regions, each with its particular shape and size and border, mentioning for further processing like classification and object detection.
- Image segmentation describes pixels-by-pixel details of an object, making it different from classification and object detection.

2.4.5 Image Segmention Algorithm

- Image segmentation techniques use different algorithms.

Table 2.4.1

Algorithm	Description
Edge detection segmentation	Makes use of discontinuous local features of an image to detect edges and hence define a boundary of the object.
Mask R-CNN	Gives three outputs for each object in the image; its class, bounding box coordinates, and object mask.
Segmentation based on clustering	Divides the pixels of the image into homogenous clusters.
Region-based segmentation.	Separates the objects into different regions based on threshold value (s).

- To solve segmentation problems in a given domain, it is usually necessary to combine algorithms and techniques with specific knowledge of the domain.

2.5 SQUARED ERROR LOSS

Squared-error loss is a widely-used loss function in machine learning and statistics. It measures the average squared difference between the predicted values and the actual target values.

It is especially useful for regression problems, where the aim is to predict continuous numerical values.

Squared error loss or mean squared error loss is calculated by squaring the difference between the value by and the predicted value. We sum the numbers to get a total value and then divide the number y again.

$$\text{Thus, MSE} = \frac{1}{n}\Sigma (y - \hat{y})^2$$

This yields a positive number.

Root mean squared error (RMSE) is given by,

$$\text{RMSE} = \sqrt{\frac{1}{n}\Sigma (y - \hat{y})^2}$$

Mean squared error is used in machine learning for training. We mention below some of them :

1. **Linear regression :** Linear regression is a relationship between a dependent variable and one or more independent variables. It assumes a linear relationship and tries to fit a straight line and that minimizes MSE between the predicted values and the actual target values.

2. **Ridge and Lasso regression :** These are extensions of linear regression that incorporate regularization techniques, L_2 for Ridge, L_1 for Lasso, to prevent overfitting.
3. **Decision trees and random forests :** For the purpose of regression, decision trees and random forests can be trained to minimize the MSE at each node. It helps the model to make better predictions in the feature space.
4. **Neural networks :** Neural networks can be trained with MSE as the loss function. By minimizing MSE, the network learns to predict continuous values that are close to the actual target values.

   MSE is simple, interpretable, and differentiable and that makes it suitable for gradient-based optimization algorithms.

   Thus MSE is a popular choice for training regression models.

2.6 CROSS-ENTROPY

2.6.1 Decision Tree used for Feature-Selection

As the decision –tree building algorithm selects the splits locally, so that the features occurring in decision tree are complementary. In some cases, the full classification of decision trees use only a small part of the feature.

Importance of Feature in Decision Tree

- Feature value is calculated as the decrease in node (weighted) impurity by the probability of reaching that node. The node probability can be calculated by the number of samples that reach the node, divided by the total number of samples.
- The higher the value, the more important is the feature.

2.6.2 Entropy Value in Decision Tree

- Entropy is a measure of disorder or uncertainty. Entropy controls decision tree to split the data. It actually effects how a decision tree draws its boundaries.
- Entropy is a measure of the purity of the sub-split. The entropy of any split can be calculated as : the algorithm calculates the entropy of each feature after every split and as the splitting continues on, it selects the best feature and starts splitting according to it.
- Information gain is the difference between the entropy before and after a decision. Entropy is minimal (i.e. 0) when all examples are +ve or –ve, and is maximal (i.e. 1) when half are positive and half are negative.
- The entropy of a set of sets is the weighted sum of the entropies of the sets. Entropy is calculated for every feature, and the one yielding the minimum value is selected for the split. The mathematical range of entropy is from 0 to 1.

2.6.3 Entropy in Decision Tree

- Entropy helps us to build an appropriate decision tree for selecting the best splitter. Entropy can be defined as a measure of the purity of the sub-split. Entropy always lies between 0 to 1.
- A random variable with only one value a coin that always comes up heads-has no uncertainty and thus its **entropy** is defined as zero, because we gain no information by observing its value.
- A fair coin is equally likely to come up heads or tails, 0 or 1, and we see that this counts as **"1 bit"** of entropy. The roll of a fair four sided die has 2 bits of entropy, because it takes two bits to describe one of four equally probable choices.
- Let us consider an **unfair** coin that comes up heads 99 % of the time. Here, if we guess heads, we shall be wrong only 1 % of the time – so the entropy measure ill be close to zero but positive.
- In general, the entropy of a random variable V with values V_k, probability $P(V_k)$ is defined as, each with

Entropy :
$$H(V) = \sum_k P(V_k) \cdot \log_2 \left[\frac{1}{P(V_k)}\right]$$
$$= - \sum_k P(V_k) \cdot \log_2 [P(V_k)]$$

Now we can check that the entropy of a fair coin is 1 bit :

$$H(\text{fair}) = -[0.5 \log_2(0.5) + 0.5 \log_2(0.5)]$$
$$= -0.5\left[\log_2\left(\frac{1}{2}\right) + \log_2\left(\frac{1}{2}\right)\right]$$
$$= -\frac{1}{2}\left[\log_2\left(\frac{1}{2}\cdot\frac{1}{2}\right)\right] = -\frac{1}{2}[-\log_2(2^2)] = \frac{1}{2}[2\log_2(2)] = 1$$

If the coin is loaded to give 99% heads, we get

$$H(\text{loaded}) = -[0.99 \log_2(0.99) + 0.01 \log_2(0.01)]$$
$$= 0.08 \text{ bits}$$

Again, the entropy of a Boolean random variable that is true wity probability q :

$$B(q) = -[q \log_2 q + (1-q)\log_2(1-q)]$$

In case of decision tree, if a training set contains p positive examples and n negative examples, then the entropy of the goal attribute on the whole set is

$$H(\text{Goal}) = B\left(\frac{p}{p+n}\right)$$

2.6.4 Use of Entropy and Information Gain in Designing Decision Tree

- Entropy provides the information about the target variable. This is exactly why decision trees use entropy and information gain to determine which feature to split their nodes on to get closer to predicting the target variable with each split and also to determine when to stop splitting the tree.
- Generally entropy value is 0.8 Good entropy value is > 0.8. Bad entropy value is difficult to specify. Note that classification table gives more detailed information than the single number of entropy.

2.6.5 Information Gain in Decision Tree

- Information Gain (I.G.) is the reduction in entropy by transforming a dataset and is often used in training decision trees. I.G. is calculated by comparing the entropy of the dataset before and after a transformation.
- Information gain helps to determine the order of attributes in the nodes of a decision tree.
- The main node is referred to as the parent node, where as sub-nodes are known as child nodes. We can use information gain to determine how good is splitting of a node in decision tree.
- Information gain is the main key that is used by decision tree algorithm; To construct 'decision tree', decision tree algorithm will always try to maximize information gain. An attribute with highest information gain will split first.

2.6.6 Entropy and Information Gain

- The information gain is the amount of information gained about a random variable or signal by observing another random variable.
- Entropy is the average rate at which information is provided by a stochastic source of data. Or, it is a measure of the uncertainty associated with a random variable.
- Information gain, like Gini impurity is a metric used to train decision tree. Specifically, these metrics measure the quality of a split.

2.7 CHOOSING OUTPUT FUNCTION AND LOSS FUNCTION

- The choice of a loss function is an important factor when training neural networks for image restoration problems, such as single image super-resolution.
- The loss function should encourage natural and perceptually pleasing results.
- A popular choice for a loss is a pre-trained network, such as VGG, which is used as a feature extractor for computing the difference between restored and reference images.

- But such as an approach has multiple drawbacks
  (i) It is computationally expensive,
  (ii) It requires regularisation and hyper-parameter tuning,
  (iii) It involves a large network trained on an unrelated task.
- It has been noted that there is no single loss function that works best across all applications and across different datasets.
- Hence w e propose to train a set of loss functions that are application specific in nature.
- The actual loss function comprises a series of discriminators that are trained to detect and penalise the presence of application-specific artifacts.
- We show that a single natural image and corresponding distortions are sufficient to train our feature extractor. And that outperforms state-of-the-art loss function. It denoises and JPEG artefact removal.
- An effective loss function need not be a good predictor of perceived image quality, but it should identify the distortions for a given restoration model.

Optimization

2.8 LEARNING WITH BPN LEARNING PARAMETERS

(i) Back-propagation network algorithm is applied to multilayer feed-forward networks consisting of processing elements.

(ii) The networks connected to back-propagation learning algorithm are also called as **back-propagation network (BPN).**

(iii) This algorithm provides a procedure for changing the weight in back-propagation network (BPN) to the given input pattern correctly.

(iv) The basic concept for this weight update is that where the error is propagated back to the hidden unit.

(v) In BPN, weights are calculated during the learning period of the network.

(vi) To update weights, the error must be calculated. There is no direct information of the error at the hidden layer. So we have to develop other techniques to calculate an error at the hidden layer, and this will cause minimisation of the output error.

(vii) Backpropagation is a systematic method of training multilayer artificial neural networks.

It is developed on high mathematical foundation and it has very good application potential.

(viii) Backpropagation learning rule is applicable on any feed forward network architecture.

(ix) Slow rate of convergence and local minima problems are its weaknesses.

(x) The multilayer feedforward (MLFF) network with back propagation (BP) learning is also called as 'multilayer perception' because of its similarity to perceptron networks with more than one layer.

(xi) We carry out BPN as follows :

(a) The feed forward of the input training pattern

(b) The back-propagation of the error and

(c) Updation of weights.

2.8.1 Architecture

A back-propagation neural network (BPN) consist of an input layer, hidden layer and an output layer. The neurons at the hidden and output layers have biases. The bias terms also act as weights. The outputs obtained could be either binary (0, 1) or bipolar (– 1, 1). Refer Fig. 2.8.1.

We Exhibit the Architecture of BPN

The terminologies used in the algorithm are as follows :

x = input training vector$(x_1, x_2, ..., x_n)$

t = target output vector $(t_1, t_2, ... t_k, ..., t_m)$

α = learning rate parameter ;

v_{oj} = bias on j^{th} hidden unit,

w_{ok} = bias on k^{th} output unit ;

Z_j = hidden unit j.

x_j = input unit j,

(since identity activation function is used for input layer, the input and output signals are same).

The net input to Z_j is,

$$Z_{inj} = V_{oj} + \sum_{i=1}^{n} x_i v_{ij}$$

and the output is, $Z_j = f(Z_{inj})$

y_k = output unit k.

The net input to y_k is,

$$y_{ink} = w_{ok} + \sum_{j=1}^{p} Z_j w_{jk} \quad \text{and the output is} \quad y_k = f(y_{ink})$$

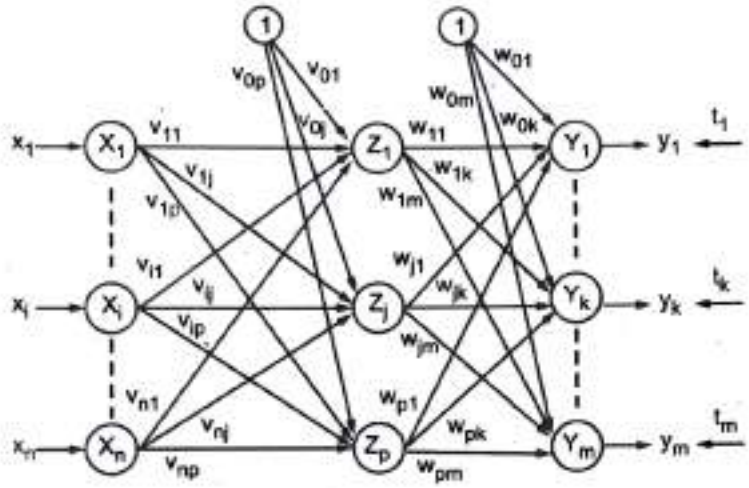


Fig. 2.8.1 : Architecture of a back-propagation network

δ_k = error correction weight adjustment for w_{jk}, which is back-propagated to the hidden units that feed into unit y_k. and

δ_j = error connection weight adjustment for v_{ij} and to the hidden unit Z_j.

2.8.2 Algorithm (Training)

We mention the error-back-propagation learning algorithm :

- **Step (I) :** Small random values for weights and learning rate,
- **Step (II) :** Carry on the steps III-X when stopping condition fails.
- **Step (III) :** Carry on steps III-IX for each training pair.

Phase (I) : Feed-forward phase

- **Step (IV) :** Each input receives input signal xi and sends to hidden unit for i = 1 to n.
- **Step (V) :** Calculate net input

$$Z_{inj} = V_{oj} + \sum_{i=1}^{n} x_i v_{ij}$$

(To calculate output, we apply activation function Z_{inj})(binary or bipolar sigmoidal activation function) : $Z_j = f(Z_{inj})$ and send the output signal to the input of output layer units.

- **Step (VI) :** For each output y_k (k = 1 to m), calculate the net input :

$$y_{ink} = w_{ok} + \sum_{j=1}^{p} Z_j w_{jk}$$

and we apply activation function to evaluate output signal : $y_k = f(y_{ink})$

Phase II + (Back-Propagation of Error)

- **Step (VII) :** We compute error correction term for each output unit y_k (K = 1 to m)

$$\delta_k = (t_k - y_k) f'(y_{ink})$$

Where $f'(y_{ink})$ is derivative of $f(y_{ink})$.

Now, on the basis of the calculated error correction term, update the change in weights and bias :

$$\Delta w_{jk} = \alpha \delta_k z_j \; ; \quad \Delta w_{ok} = \alpha \delta_k$$

The error δ_k is to be sent to the **hidden layer backwards.**

- **Step (VIII) :** For each hidden unit (Z_j, J = 1 to p) ;

$$\delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$$

and to calculate the error term : $\delta_j = \delta_{inj} \cdot f'(Z_{inj})$

Now, we update the change in weights and bias ;

$$\Delta V_{ij} = \alpha \delta_j x_i \; ; \quad \Delta V_{oj} = \alpha \delta_j$$

Phase (III) : Weight and Bias Updation

- **Step (IX) :** Each output unit (y_k, k = 1 to m) updates the bias and weights :

$$w_{jk}(new) = w_{jk}(old) + \Delta w_{jk} \; ;$$

$$w_{ok}(new) = w_{ok}(old) + \Delta w_{ok}$$

Each hidden unit (Z_j, j = 1 to p) update its bias and weights :

$$V_{ij}(new) = V_{ij}(old) + \Delta V_{ij}$$

and $$v_{oj}(new) = V_{oj}(old) + \Delta V_{oj}$$

- **Step (X) :** Check for stopping condition.

2.8.3 Flow-chart for Back-Propagation Network Training

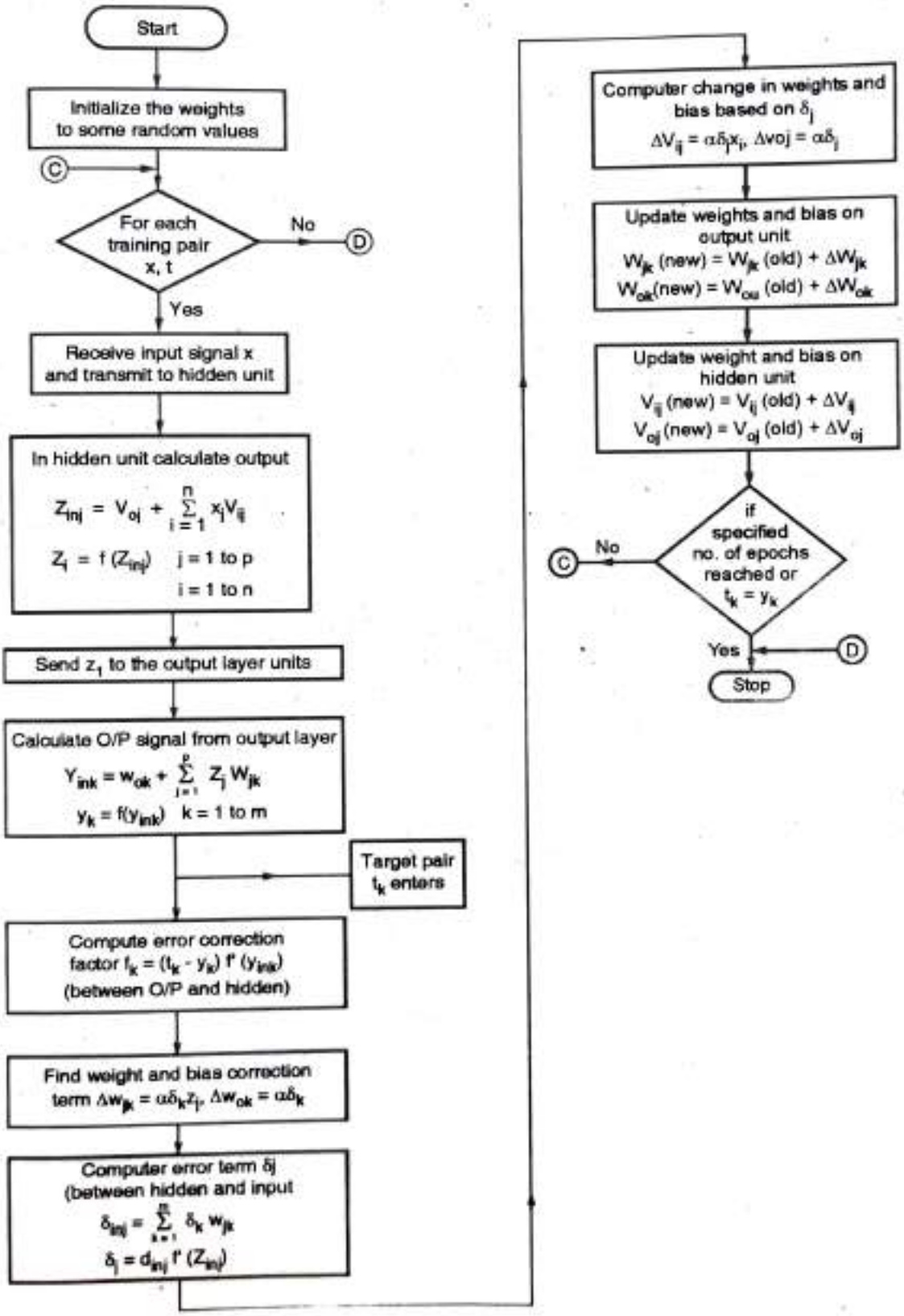


Fig. 2.8.2 : Flow-chart for Back-Propagation Network Training

2.9 GRADIENT DESCENT STOCHASTIC AND MINI-BATCH GRADIENT

- Gradient Descent is an optimization algorithm and it is used to train **machine learning models** and **neural networks.**
- Training the data helps these models learn over time and the cost function within gradient descent gauges its accuracy with each iteration of parameter updates.
- The model will continue to adjust its parameter so that possible error will be minimum. The happens when function become, near to zero.
- Once machine learning models are optimized for accuracy, they become powerful tools for computer science application.

2.9.1 Working of Gradient Descent

GQ. How gradient descent works ?

- Let us revise the concept of linear regression.
- The equation y = mx + c, represents a line with 'm ' as slope and 'c' as intercept on Y-axis.
- We can plot a scatter – diagram and find the line of best fit. It can calculate the error between the actual output and the predicted output, using the mean - squared formula.
- The gradient descent algorithm behaves similarly, but it is based on a convex function, such as :

  Point of convergence, i.e. where the cost function is at its minimum
- The starting point is chosen arbitrarily to evaluate the performance.
- From that starting point, we calculate derivative (or slope) and draw tangent line and note the steepness of the slope.
- The slope will yield the updates of the parameters i.e. the weight and bias.
- The slope at the starting point will be steeper, but as new parameters are generated, the steepness reduces till it reaches, the lowest point on the curve, known as the point of convergence.
- The aim of gradient descent is to minimize the cost function or the error between predicated and actual value.
- To do this, we require two data points : a direction and a learning rate.

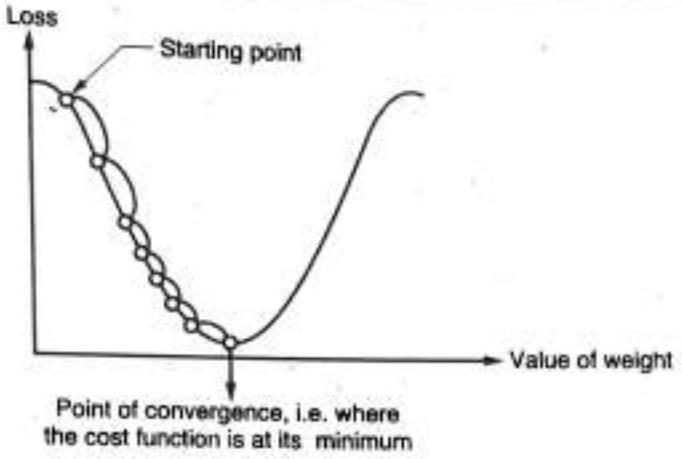
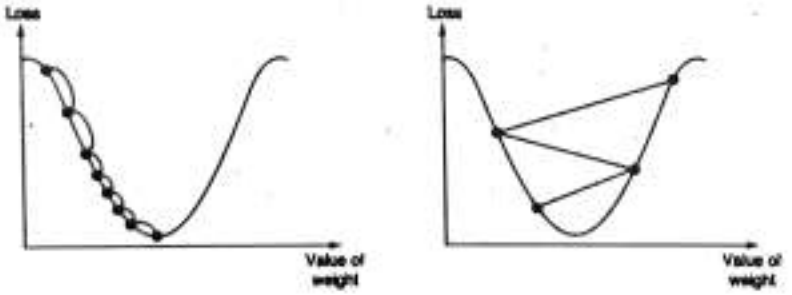


Fig. 2.9.1 : Gradient descent algorithm

- Using these factors, we can calculate partial derivatives of future iterations, and thereby we get local or global minimum i.e. point of convergence.
- **Learning rate :** This is also called as step size or alpha. This is the size of the steps to reach to the minimum. This is typically a small value and it is evaluate and updated, depending upon the behaviour of the cost function. High learning rates result in larger steps but the risk involved is minimum.
- Conversely, a low learning rate has small step sizes. The advantage of this method is gives more precision. Since the number of iterations are more, it takes more time and computations to reach the minimum.



(a) Small learning rate (b) Large learning rate

Fig. 2.9.2

2.9.2 The Cost (or loss) Function

- This function measures the difference, or error between actual y and predicted y at its current position.
- This provides feedback to the model so that it can adjust parameters to minimize the error and can find local or global minimum. This way model's efficacy is improved.
- It iterates continuously i.e. it moves along the direction of steepest descent, until the cost function is minimum.

- At this point, the model stops learning. Here, we note that there is a slight difference between cost function and cost function and loss function. A loss function refers to the error of one training example, while a cost function calculates the average error across an entire training set.

2.9.3 Types of Gradient Descent

There are three types of gradient descent learning algorithms :

(i) batch-gradient descent

(ii) stochastic-gradient descent and

(iii) mini-batch gradient descent

2.9.3(A) Batch Gradient Descent

- Batch gradient descent sums the error at each point in a training set, updating the model only after all training examples have been evaluated.
- This process is referred to as a training epoch.
- While this batching provides computation efficiency, it can have long processing time for large training datasets. It needs actually to store all of the data into memory. Batch descent method produces a stable error gradient and convergence, but the convergence point is not most ideal.

2.9.3(B) Stochastic Gradient Descent

- Stochastic gradient descent (SGD) runs a training epoch for each example in the dataset.
- It updates each training examples parameter one at a time.
- Since it is only one training example, they are easier to keep in memory.
- These frequent updates can offer more detail and also speed, it results in losses in computational efficiency when compared to batch gradient descent.
- Its frequent updates create noisy gradients, but this helps to find global minimum.

2.9.3(C) Mini-batch Gradient Descent

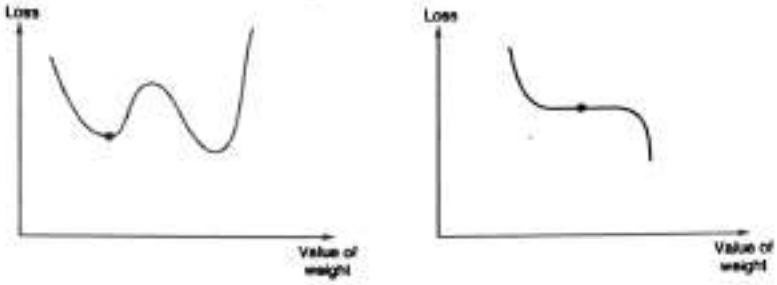
- Mini-batch gradient descent combines concepts from both batch gradient and stochastic gradient descent.
- It splits the training dataset into small batch sizes and performs updates on each of those batches. This approach makes a baloma between the computational efficiency of batch gradient descent and the speed of stochastic gradient descent.

2.9.4 Challenges with Gradient Descent

Gradient descent is the most common approach for optimization problems, but it has its own set of challenges.

For example : Local minima and saddle points

- For, convex problem, local minimum can be found easily. But for non-convex problems, gradient descent has to struggle to find global minimum.
- Note that when the slope of the cost function is at zero (or close to zero), the model stops learning.



(a) (b)

Fig. 2.9.3 : Value of weight

- At local minima and saddle points the slope of the cost function is zero. But the slope of cost function, increases on either side of the current point.
- Noisy gradients can help the gradient escape local minimum and saddle points.

2.10 MOMENTUM BASED GRADIENT DESCENT

- Momentum based gradient descent is an optimization algorithm used to train different models.
- The training process consists of an objective function (or error function), which finds out the error a machine learning model has on a given dataset.
- Initially, the parameters of the algorithm are given random values. When the algorithm iterates, the parameters are updated so that we can reach close to the optimal value of the function.
- In the momentum based gradient optimizer, to the current update a fraction of the previous update is added and this creates a momentum effect that helps the algorithm to move faster towards the minimum.
- The momentum term is set to a value between 0 and 1, with a higher value resulting in a more stable optimization process.

Advantages of momentum-based gradient descent :

(i) Momentum is faster than stochastic gradient descent and the training will be faster than SGD.

(ii) Local minima may not be attained but global minima can reach due to the momentum involved.

- SGD with momentum is like a ball rolling down a hill. It will take large step if the gradient direction points to the same direction from previous. But it will slow down if direction changes.

☞ **Benefit of momentum optimization in deep learning**

(i) Momentum is a strategy used to accelerate the convergence of gradient based optimization techniques.

(ii) Momentum was designed to speed up learning in directions of low curvature, without becoming unstable in the directions of high curvature.

☞ **Disadvantages of gradient descent momentum**

(i) The gradient of the cost function at saddle points is negligible or zero, which in turn leads to small or no weight updates. Hence, the network becomes stagnant, and learning stops.

(ii) The path followed by Gradient descent is not clear even when operating with mini-batch mode.

☞ **Advantages of momentum**

(i) Momentum shows the rate of change in price movement over a period of time to help investors determine the strength of a trend.

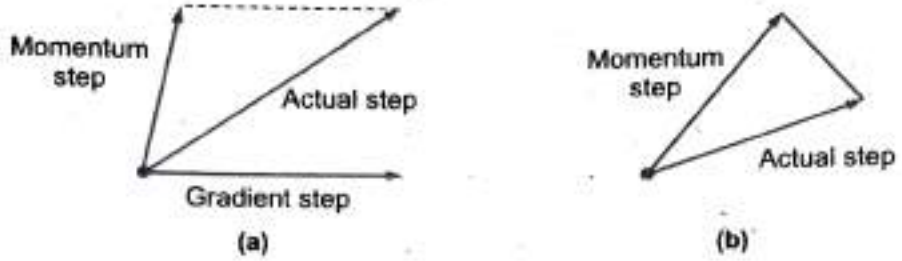
(ii) Investors use momentum to trade stocks whereby a stock can exhibit bullish momentum – the price is rising – or bearish momentum – the price is falling.

(iii) Momentum can accelerate training and learning rate schedules and can help to converge the optimization process.

⏭ 2.11 NESTEROV ACCELERATED - GD

- Nesterov Accederated Gradient (NAG) is a modified version of momentum with stronger theoretical convergence and guarantees for convex functions.
- The gradient descent optimization algorithm follows the negative gradient of an objective function to locate its minimum. But it has limitations. It gets stuck in flat areas and struggles with noisy gradients.
- Nesterov momentum involves calculating the decay moving average of the gradient for projected positions in the search space, but not the actual positions themselves.

Momentum vs. Nesterov Momentum



(a) Momentum update (b) Nesterov momentum update

Fig. 2.11.1

- In the standard momentum method, the gradient is computed using current parameters (θt).
- Nesterov momentum achieves stronger convergence by applying the velocity (vt) to the parameters in order to compute interim parameters ($\theta = \theta t + \mu^* vt$), where μ is the decay rate.
- These interim parameters are then used to compute the gradient, called a "lookahead" gradient step or a Nesterov accelerated gradient.
- Thus, Nesterov momentum can be described in terms of four steps :
  (1) Project the position of the solution.
  (2) Calculate the gradient of the projection.
  (3) Calculate the gradient of the variable.
  (4) Finally update the variable.
- Nesterov momentum improves the rate of convergence of the optimization algorithm, (e.g. reduce the number of iterations required to find the solution), in particular in the field of convex optimization.
- Advantage of nesterov accelerated gradient is that it allows larger decay rate α than momentum method, while preventing oscillations.

Difference between Nesterov and normal momentum

The main difference is in classical momentum we first correct velocity and then make a big step according to that velocity (and then repeat the process), but in Nesterov momentum, we first make a step into velocity direction and then make a correction to a velocity vector based on new location (and then repeat the process).

2.12 ADA GRAD

'Adagrad' stands for 'Adaptive Gradient Optimizer'. The optimisers like gradient Descent, Stochastic Grdient Descent, Mini-batch SGD, are used to reduce the loss function with respect to the weights.

The weight update formula is given by,

$$(w)_{new} = (w)_{old} - \eta \frac{\partial L}{\partial w\ (old)}$$

For iterations, the formula takes the form :

$$w_t = w_{t-1} - \eta \frac{\partial L}{\partial w\ (t-1)}$$

where w (t) = value of w at current iteration, w (t – 1) = value of w at previous iteration and η = learning rate.

In SGD and mini-batch SGD, the value of η used to be the same for each weight, or for each parameter.

But in Adagrad optimizer each weight has a different learning rate (η).

In the given dataset, some features are sparse (i.e. the features are zero) and some are dense (i.e. the features are zero), and some are dense (i.e. the features are non-zero). Thus keeping the same value of learning rate for all weights is not good for optimization.

Hence, the weight updating formula for adagrad is given by,

$$w_t = w_{t-1} - \eta_t' \frac{\partial L}{\partial w\ (t-1)};$$

and

$$\eta_t' = \frac{\eta}{sqrt\ (\alpha_t + \epsilon)}; \qquad ...(1)$$

Here, η is a constant number, ϵ is a small positive number

Now,

$$\alpha_t = \sum_{i=1}^{t} g_i^2, \text{ where}$$

$$g_i = \frac{\partial L}{\partial w\ (old)}$$

g_i is derivative of loss with respect to weight and g_i^2 is always positive, hence α_t is always positive.

$$\therefore \quad \alpha_t \geq \alpha_{t-1}$$

From Equation (1), we note that α_t and η_t' are inversely proportional to one another. Hence, if α_t increases, η_t' decreases. This implies that if number of iterations increases, the learning rate will decrease adaptively. Hence there is no need to select the learning rate.

Advantages of Adagrad

(i) No manual tuning of the learning rate is required,

(ii) Faster convergence,

(iii) More reliable.

Disadvantage of Adagrad

If $\alpha(t)$ becomes large as the number of iterations increase, then η_t' will decrease at the larger rate. This makes the old weight equal to new weight and this leads to slow convergence.

2.13 ADAM

- The name 'Adam' is derived from 'adaptive moment estimation'. This optimization algorithm is an extension of stochastic gradient descent to update network weights during training.
- An Adam optimizer algorithm is used in deep learning that 'helps improve the accuracy of neural networks by adjusting the model's learnable parameters. It was first introduced in 2014 and is an extension of the stochastic gradient descent (SGD) algorithm.
- The benefits of using Adam on non-convex optimization problems are:
  - (i) Straight forward to implement
  - (ii) Computationally efficient
  - (iii) Little memory requirements
  - (iv) Invariant to diagonal rescale of the gradients
  - (v) Well suited for problems that are large in terms of data and / or parameters.
  - (vi) Appropriate for non-stationary objectives.
  - (vii) Appropriate for problems with very noisy / or sparse gradients.
  - (viii) Hyper-parameters have intuitive interpretation and typically require little tuning.
- Adam is a popular algorithm in the field of deep learning because it achieves good results fast.
- Empirical results demonstrate that Adam works well in practice and compares favourably to other stochastic optimization methods.
- Adam is known for its fast convergence and ability to work well on noisy and sparse datasets. It can also handle problems where the optimal solution lies in a wide range of parameter values.

- Overall, the Adam optimizer is a powerful tool for improving the accuracy and speed of deep learning models. By analyzing the gradients and adjusting the learning rate for each parameter in real-time, Adam can help the neural network converge faster and more accurately during training.

☞ Adam configuration parameters

The Adam optimiser has several configuration parameters that can be adjusted to improve the performance of a deep learning model. We mention some of the main Adam configuration parameters :

(i) **Learning rate :** This parameter controls how much the model's parameters are updated during each training step.

A high learning rate can result in large updates to the model's parameters, which can cause the optimization process unstable.

On the other hand, a low learning rate can result in slow convergence and may require some training steps to reach the optimal set of parameters.

(ii) **Weight decay :** This is a regularization term that can be added to the cost function during training to prevent overfitting. It penalizes large values of the model's parameters, which can help to improve generalization to new data.

(iii) **Batch size :** This parameter controls how many training examples are used in each training step. A large batch size can result in faster convergence, but can also increase memory requirements and slow down the training process.

(iv) **Max epochs :** This parameter determines the maximum number of training iterations that will be performed during training. It helps to prevent overfitting and can improve the generalization of the model to new data.

By fine-tuning these parameters, developers can achieve faster convergence and better generalization to new data.

⏭ 2.14 RMS PROP

- RMS prop is Root Mean Square propagation. It is an adaptive learning rate optimization algorithm. It is an extension of the popular Adaptive Gradient Algorithm and is designed to dramatically reduce the amount of computational effort used in training neural networks.
- RMS prop also tries to dampen the oscillations, but in a different way than momentum. RMS prop also takes away the need to adjust learning rate, and does it automatically. Also, RMS prop choses a different learning rate for each parameter.
- In RMS prop, each update is done according to the equations described below. This update is done separately for each parameter.

For each parameter w^j (we drop superscript j for clarity)

$$V_t = \rho V_{t-1} + (1-\rho) * g_t^2 \qquad ...(i)$$

$$\Delta w_t = -\frac{\eta}{\sqrt{V_t + \epsilon}} * g_t \qquad ...(ii)$$

$$w_{t+1} = w_t + \Delta w_t, \qquad ...(iii)$$

where

η : Initial learning rate,

V_t : Exponential average of squares of gradients,

g_t : Gradient at time t along w^j.

In the first equation, we compute an exponential average of the square of the gradient.

Since we do it separately for each parameter, gradient g_t (or G_t) here corresponds to the projection, or component of the gradient along the direction represented by the parameter we are updating.

In the second equation, we choose the step size. We move in the direction of the gradient, but our step size is affected by the exponential average. We choose an initial learning rate 'eta', and then divide it by the average. This will help us avoid bouncing between the ridges and move towards the minima.

The third equation is just the update step. The hyper parameter ρ is generally chosen to be 0.9, but we may have to tune it. The epsilon in Equation (ii) is to ensure that we do not end up dividing by zero and is generally chosen to be $1 \cdot e^{-10}$.

RMS prop also implicitly performs simulated annealing. Suppose if we are heading towards the minima, and we want to slow down so as not to overshoot the minima, RMS prop automatically decreases the size of the gradient steps towards minima when the steps are too large (Large steps are prone to overshooting).

Regularization

2.15 OVERVIEW OF OVERFITTING

- Let us consider that we are designing a machine learning model. A model is said to be a good machine learning model if it generalizes any new input data from the problem domain in a proper way. This helps us to make predictions in the future data, that data model has never seen.
- Now, suppose we want to check how well our machine learning model learns and generalizes to the new data. For that we have overfitting and underfitting, which are majorly responsible for the poor performances of the machine learning algorithms.

Before diving further let's understands two important terms

- Bias – Assumptions made by a model to make a function easier to learn. (The algorithms error rate on the training set is algorithms bias.)
- Variance - If you train your data on training data and obtain a very low error, upon changing the data and then training the same previous model you experience high error, this is variance,
- (How much worse the algorithm does on the test set than the training set is known as the algorithms variance.)

2.15.1 Underfitting

- A statistical model or a machine learning algorithm is said to have underfitting when it cannot capture the underlying trend of the data.
- Underfitting destroys the accuracy of our machine learning model. Its occurrence simply means that our model or the algorithm does not fit the data well enough.
- It usually happens when we have less data to build an accurate model and also when we try to build a linear model with a non-linear data.
- In such cases the rules of the machine learning model are too easy and flexible to be applied on such minimal data and therefore the model will probably make a lot of wrong predictions.
- Underfitting can be avoided by using more data and also reducing the features by feature selection.
- In a nutshell, Underfitting – High bias and low variance

2.15.2 Techniques to Reduce Underfitting

1. Increase model complexity
2. Increase number of features, performing feature engineering
3. Remove noise from the data.
4. Increase the number of epochs or increase the duration of training to get better results.

2.15.3 Overfitting

- A statistical model is said to be overfitted, when we train it with a lot of data . When a model gets trained with so much of data, it starts learning from the noise and inaccurate data entries in our data set.

- Then the model does not categorize the data correctly, because of too many details and noise.
- The causes of overfitting are the non-parametric and non-linear methods because these types of machine learning algorithms have more freedom in building the model based on the dataset and therefore they can really build unrealistic models.
- A solution to avoid overfitting is using a linear algorithm if we have linear data or using the parameters like the maximal depth if we are using decision trees.
- In a nutshell, Overfitting – High variance and low bias

2.15.4 Techniques to Reduce Overfitting

GQ. What are Techniques to Reduce Overfitting.

1. Increase training data.
2. Reduce model complexity.
3. Early stopping during the training phase (have an eye over the loss over the training period as soon as loss begins to increase stop training).
4. Ridge Regularization and Lasso Regularization
5. Use dropout for neural networks to tackle overfitting.

- Ideally, the case when the model makes the predictions with 0 error, is said to have a *good fit* on the data. This situation is achievable at a spot between overfitting and underfitting. In order to understand it we will have to look at the performance of our model with the passage of time, while it is learning from training dataset.
- With the passage of time, our model will keep on learning and thus the error for the model on the training and testing data will keep on decreasing. If it will learn for too long, the model will become more prone to overfitting due to the presence of noise and less useful details.
- Hence the performance of our model will decrease. In order to get a good fit, we will stop at a point just before where the error starts increasing. At this point the model is said to have good skills on training datasets as well as our unseen testing dataset.

Bias-variance trade-off

So what is the right measure? Depending on the model at hand, a performance that lies between overfitting and underfitting is more desirable. This trade-off is the most integral aspect of Machine Learning model training.

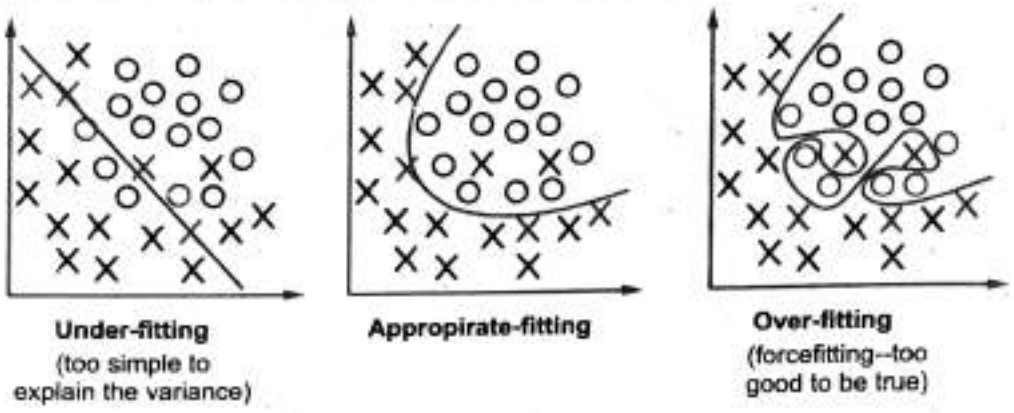


Fig. 2.15.1 : Underfitting and Overfitting

As we discussed, Machine Learning models fulfil their purpose when they generalize well. Generalization is bound by the two undesirable outcomes - high bias and high variance. Detecting whether the model suffers from either one is the sole responsibility of the model developer.

Bias-variance trade-off

So what is the right measure? Depending on the model at hand, a performance that lies between overfitting and underfitting is more desirable. This trade-off is the most integral aspect of Machine Learning model training. As we discussed, Machine Learning models fulfil their purpose when they generalize well. Generalization is bound by the two undesirable outcomes - high bias and high variance. Detecting whether the model suffers from either one is the sole responsibility of the model developer.

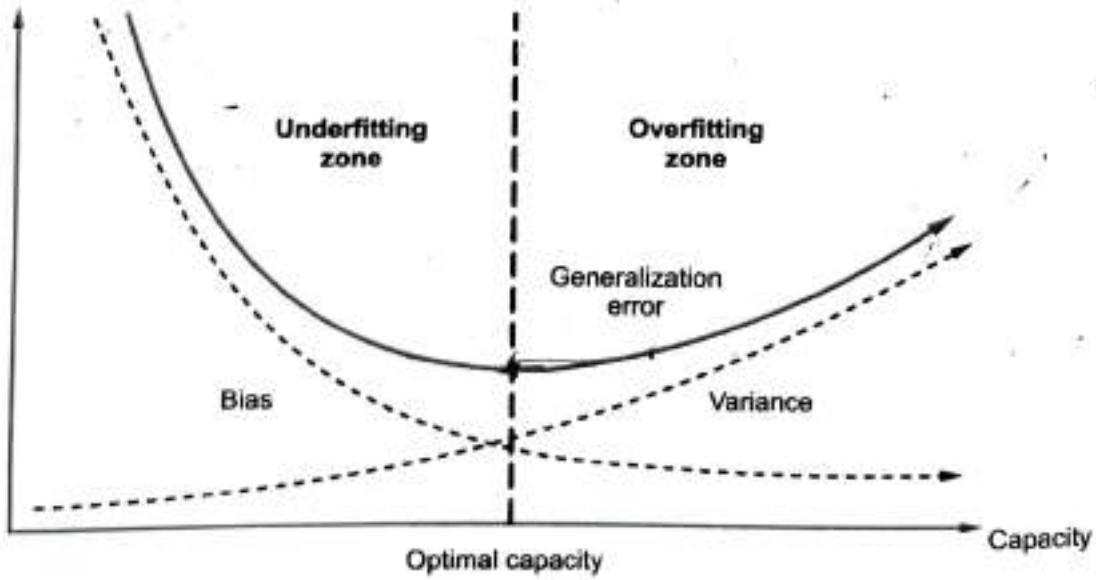


Fig. 2.15.2 : Bias variance Tradeoff as a function of model capacity

2.16 L_1, L_2 REGULARIZATION

To prevent overfitting in the training phase, there are different ways of controlling training of CNNs. In particular they are L_2 / L_1 regularisation and max-norm constraints :

(1) **L_2 regularisation :** This regularisation is implemented by penalising directly the squared magnitude of all parameters in the objectives. Using the gradient descent parameter, every weight is decayed linearly towards zero by L_2 regularisation method.

(2) **L_1 regularisation :** Here in this method, we add the term $\lambda|\omega|$ for each weight ω to the objective. It is also possible to combine the L_1 regularisation with L_2 regularisation $\lambda_1 |\omega_1| + \lambda_2 |\omega_2|$ and is known as Elastic-net regularisation.

(3) **Max-Norm constraint :** Here we enforce an absolute upper bound on the magnitude of the weight vector for every neuron and use projected gradient descent to enforce the constraint. This is altogether a different form of regularisation.

2.17 PARAMETER SHARING (WEIGHT SHARING) IN CNN

- Parameter sharing makes sets of parameters to be similar as we interpret various models or model components as sharing a unique set of parameters. We only store a subset of memory.
- Let us consider two models A and B. They perform a classification task on similar input and output distributions.
- Here we can assume that the parameters for both models to be identical to each other.
- If there is distance between the weights, we can impose norm-penalty. But instead we force the parameters to be similar. This way we have only to store a subset of the parameters (e.g. storing only the parameters for model A instead of storing for both A and B). This leads to significant memory saving.
- **Example :** The most extensive use of parameter sharing is in convolutional neural networks.
- Natural images have specific statistical properties that are robust for translation.
- For example, a photo of a cat remain a photo of a cat if it is translated one pixel to the right.
- Convolutional Neural Networks considers this property by sharing parameters across multiple image locations.

- It implies that we can find a cat with same cat detector in column (i) or (i + 1) in the image.

(1) Benefits of parameter-sharing
(2) Importance of parameter sharing
(3) Weight sharing in CNN
(4) Parameters of CNN
(5) Parameters in neural network
(6) Calculation of CNN Parameters
(7) Parameters in deep learning
(8) Parameters in a model
(9) Difference between a variable and a parameter
(10) Parameter Sharing in Deep Learning
(11) Hard parameter sharing
(12) Soft parameter sharing

(1) Benefits of parameter-sharing

- Convolution Neural Networks have a couple of techniques known as parameter sharing and parameter tying.
- Parameter sharing is the method of sharing weights by all neurons in a particular feature map.
- It helps to reduce the number of parameters in the whole system, and it makes it computationally cheap.
- Parameter sharing is used in all convolution layers in the network. Parameter sharing reduces the training time and that directly reduces the number of weight updates during back propagation.
- Recurrence Neural Networks also use shared parameters. The shared weights perspective comes from thinking about RNNS as feed. Forward networks. If the weights were different at each moment, this would be a feed forward network.

(2) Importance of parameter sharing

- The main purpose of parameter sharing is a reduction of the parameters that the model has to learn.
- This is the purpose of using RNN. If one learns a different network for each time-step and feed the output of the first model to the second etc; one would end up with a regular feed forward network.

(3) Weight sharing in CNN

- A typical application of weight sharing is to share the same weights across all four filters.
- In this content weight sharing has the following effects. It reduces the number of weights that must be learned and that reduces model learning time and cost.

(4) Parameters of CNN

- In a CNN, Each layer has two kinds of parameters : weights and biases.
- The total number of parameters is just the sum of all weights and biases, = Number of weights of convolutional

  Layer = number of biases of the convolutional layer.

(5) Parameters in neural network

- Parameters are the coefficients of the model, and they are chosen by the model itself.
- It means that the algorithm, while learning, optimises these coefficients and returns an array of parameters which minimises the error.

(6) Calculation of CNN Parameters

Number of parameters = 0. CONN layer :

This is where CNN learns, hence we have weight matrices. To calculate the learnable parameters, all we have to do is, just multiply the shape of width m, height n, previous layer's filters d and account for all such filters k in the current layer.

(7) Parameters in deep learning

- Model parameters are properties of training data that will learn during the learning process.
- In case of deep learning, weight and bias are parameters. Parameter is used ϕ as a measure of how well a model is performing.

(8) Parameters in a model

- A model parameter is a configuration variable and it is internal to the model and its value can be estimated from data.
- They are required by the model when making predictions. Their values define the skill of the model on the problem. They are estimated from data.

(9) Difference between a variable and a parameter

- There is a clear difference between variables and parameters. A variable represents a model state, and may change during simulation.
- A parameter is commonly used to describe object statistically.

- A parameter is normally a constant in a single simulation, and is changed only when you need to adjust your model behaviour.

(10) Parameter Sharing in Deep Learning

- The two main approaches for performing MTL (Multi-task Learning) with neural networks are hard and soft parameters sharing, here we wish to learn **shared** or **"similar"** hidden representations for the different tasks.
- In order to impose these similarities between tasks, the model learns simultaneously for all tasks and with some constraints or regularisation on the relationship between related parameters.

(11) Hard parameter sharing

- In **hard parameter sharing** we learn a common space representation for all tasks (i.e. completely share weights/ parameters between tasks). This shared feature space is used to model the different tasks, usually task specific layers.
- Hard parameters sharing acts as regularisation and reduces the risk of over fitting, as the model learns a representation that will generalise well for all tasks.

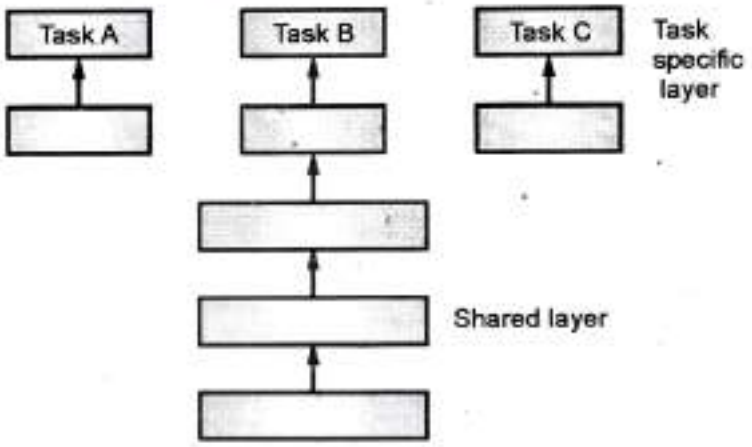


Fig. 2.17.1 : Hard parameter sharing architecture

Example of **hard parameter sharing architecture**

(12) Soft parameter sharing

- Instead of sharing exactly the **same** value of the parameters, in **soft parameter sharing**, we add a constraint to encourage similarities among related parameters.
- We learn a model for each task and penalise the distance between the different models parameters. This approach gives more flexibility for the tasks by only loosely coupling the shared space representations.
- We exhibit an example of soft parameter sharing architecture :

Let us suppose that we are interested in learning 2 tasks, A and B, and denote the i^{th} layer parameters for task j by $W_i^{(j)}$, one possible approach to impose similarities between corresponding parameters is to augment our loss function with an additional loss term,

$$L = l + \sum_{S(L)} \lambda_i \, || \, W_i^{(A)} - W_i^{(B)} \, ||_F^2$$

Where l is the original loss function for both tasks (e.g. sum of losses) and $|| \cdot ||_F^2$ is the squared Frobenius norm.

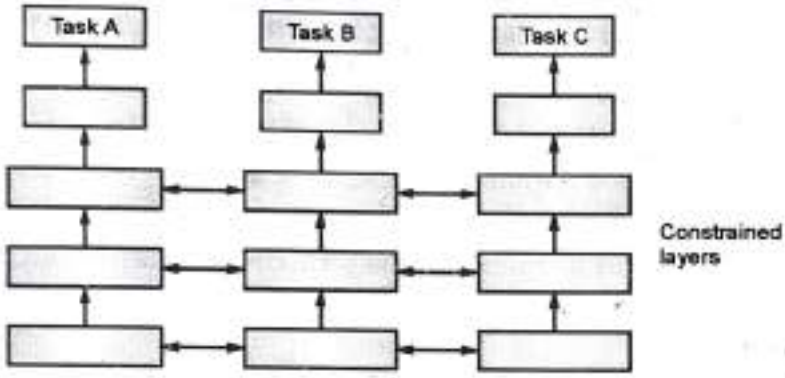


Fig. 2.17.2 : Architecture of soft parameter

2.18 DROPOUT

- A typical characteristic of CNN is a dropout layer. The dropout layer is a 'mask that nullifies the contribution of some neurons towards the next layer and leaves unmodified all others.
- The dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps to prevent overfitting. Inputs not set to 0 are scaled up by $\frac{1}{(1 - \text{rate})}$ such that the sum over all inputs is unchanged.
- The dropout layer acts as a mask, eliminating some neurons contributions to the subsequent layer while maintaining the functionality of all other neurons.
- If we apply a dropout layer to the input vector, some of its features are eliminated; but, if we apply it to a hidden layer, some hidden neurons are eliminated.
- It is because they avoid overfitting the training data, dropout layers are crucial in the training of CNNs. If they are absent, the first set of training samples has a large impact on learning. As a result, traits that only show in later batches would not be learned.
- Let us imagine that during training, we display a CNN ten images of a circle one after the other. If we later show CNN an image of a square, it will not understand that straight lines exist and will be very much puzzled. By incorporating dropout layers into the network's architecture to minimize overfitting, we may avoid such situations.

2.19 WEIGHT DECAY

- Weight decay is a regularization technique that is used in machine learning to reduce the complexity of a model and prevent overfitting.
- It improves the generalization performance of many types of machine learning models, including deep neural networks.
- Weight decay is used to regularize the size of the weights of certain parameters in machine learning models.
- Weight decay is also known as L_2 regularization, because it penalizes weights according to their L_2 norm.
- In weight decay technique, the objective function of minimizing the prediction loss on the training data is replaced with the new objective function, and it minimizes the sum of the prediction loss and the penalty term. It involves adding a term to the objective function that is proportional to the sum of the squares of the weights.
- Then the new loss function using weight decay technique is :

$$L(w, b) + \frac{\lambda}{2} || w ||^2$$

- Here, L (w, b) represents the original loss function before adding the regularization L_2 norm
(weight decay) term.
- For the value of λ as 0, we get the original loss function. For the values of $\lambda > 0$, the size of weight vector is selected appropriately to minimize the overall loss.
- In general, weight decay improves the generalization performance of a machine learning model by preventing overfitting.

2.20 BATCH NORMALIZATION

- Batch norm is a normalization technique done between the layers of a Neural Network instead of in the raw data. It is done along mini-batches instead of the full data set. It serves to speed up training and use higher learning rates, making learning easier.
- We define the normalization formula of batch norm as :

$$Z^N = \left(\frac{Z - m_z}{S_z}\right),$$

where m_z is the mean of the neurons' output and S_z the standard deviation of the neurons' output.

How it is applied

In the following image, we see a regular feed-forward neural network : x_i are inputs, Z the output of the neurons, the output of the activation functions, and y the output of the network.

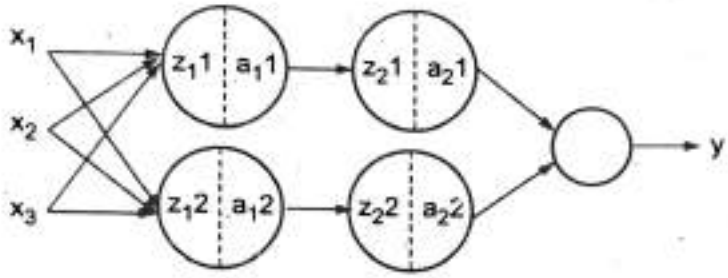


Fig. 2.20.1

Batch norm

In the image represented with a dotted line is applied to the neurons' output just before applying the activation function.

Adding batch norm, a neuron looks as :

$$Z = g(w, x) ; Z^N = \left(\frac{Z - m_z}{S_z}\right) \cdot \gamma + \beta ;$$

being Z^N the output of batch norm, m_z is the mean of the neurons' output, S_z the standard deviation of the output of the neurons, and γ and β learning parameters of batch norm.

The parameters β and γ shift the mean and standard deviation, respectively.

Thus, the outputs of batch norm over a layer results in a distribution with a mean β and a standard deviation of γ.

These values are learned over iterations and the other learning parameters, such as the weights of the neurons, aiming to decrease the loss of the model.

2.21 EARLY STOPPING

In machine learning, 'early stopping' is a form of regularisation used to avoid overfitting when training a learner with an iterative method, such as gradient descent

☞ Overfitting

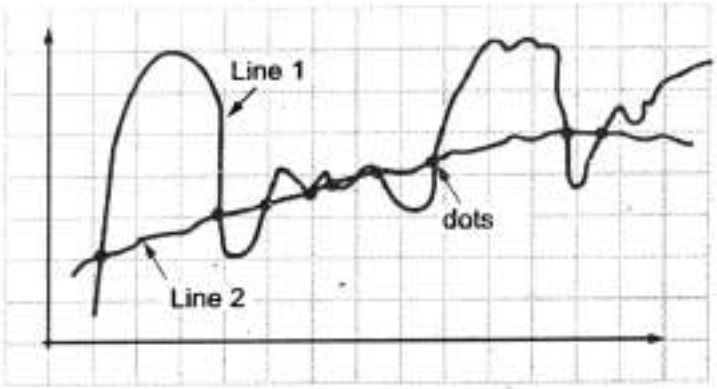


Fig. 2.21.1

- The image represents the problem of over-fitting in machine learning. The 'dots' represent training set data. The line 2 represents the true functional relationship, while the line 1 shows the learned function, which has fallen victim to overfitting.
- Machine learning algorithms train a model based on a finite set of training data.
- The goal of a machine learning scheme is to produce a model that predicts previously unseen observations. Overfitting occurs when a model fits the data in the training set, while incurring larger generalization error.

☞ Regularisation

- Regularisation refers to the process of modifying a learned algorithm so as to prevent overfitting. This involves some sort of smoothness constraint on the learned model.
- This smoothness may be enforced explicitly, by fixing the number of parameters in the model, or by augmenting the cost function.

☞ Gradient descent methods

- Gradient descent methods are first-order, iterative, optimization methods.
- Each iteration updates an approximate solution to the optimization problem by taking a step in the direction of the negative of the gradient of the objective function.
- By appropriately choosing the step-size, such a method can be made to converge to a local minimum of the objective function.

⏭ 2.22 DATA AUGMENTATION

- Data augmentation is a technique of artificially increasing the training set by creating modified copies of a dataset using existing data.
- It includes making minor changes to the dataset or using deep learning to generate new data points. Thus data augmentation is the addition of new data artificially derived from existing training data.

- Techniques include resizing, flipping, rotating, cropping, padding etc. It helps to address issues like overfitting and data scarcity, and it makes the model robust with better performance.
- Data augmentation can help to overcome some common problems in image classification, such as overfitting, class imbalance, and limited data availability.
- Overfitting occurs when the model learns to fit the training data too well, but fails to generate to new and unseen images.

☞ Data augmentation techniques

(a) Audio data augmentation

(i) **Noise injection :** Odd Gaussian or random noise to the audio dataset to improve the model performance.

(ii) **Shifting :** Shift audio left (fast forward) or right with random seconds.

(iii) **Changing the speed :** Stretches times series by a fixed rate.

(iv) **Changing the pitch :** Randomly change the pitch of the audio.

(b) Text data augmentation

(i) **Word or sentence shuffling :** Randomly changing the position of a word or sentence.

(ii) **Word replacement :** Replace words with synonyms.

(iii) **Syntax-tree manipulation :** Inserts words at random.

(iv) **Random word insertion :** Inserts word at random.

(v) **Random word deletion :** Deletes words at random.

(c) Image augmentation

(i) **Geometric transformations :** Randomly flip, crop, rotate, stretch and zoom images. One should be careful about applying multiple transformations on the same images, as this can reduce model performance.

(ii) **Color space transformations :** Randomly change RGB color channels, contrast and brightness.

(iii) **Kernel filters :** Randomly change the sharpness or blurring of the image.

(iv) **Random erasing :** Delete some part of the initial image.

(v) **Mixing images :** Blending and mixing multiple images.

Benefits of data augmentation

(i) It increases the model's ability to generalize.

(ii) It adds variability to the data and minimizes data overfitting.

(iii) It saves on the cost of collecting and labeling additional data.

(iv) It improves the accuracy of the deep learning model's predictions.

2.23 ADDING NOISE TO INPUT AND OUTPUT

- Training a neural network with a small dataset can cause the network to memories all training examples, and it leads to overfitting and poor performance on a dataset.
- One approach to making the input space smoother and easier to learn is to add noise to inputs during training.
- The addition of noise during the training of a neural network model has a regularization effect and, in turn, improves the roubustness of the model.
- The addition of noise to the input data of a neural network during training can, in some circumstances, leads to significant improvements in generalization performance.
- Such training with noise is equivalent to a form of regularization in which an extra term is added to the error function.
- In effect, adding noise expands the size of the training dataset. Each time a training sample is exposed to the model, random noise is added to the input variables making them different every time it is exposed to the model. In this way, adding noise to input samples is a simple form of 'data augmentation'.
- Injecting noise in the input to a neural network can also be regarded as a form of data augmentation.
- Adding noise means that the network is less able to memorise training samples, this results in smaller network weights and a more robust network that has lower generalization error.
- The noise means that it is as though new samples are being drawn from the domain in the vicinity of known samples, and smoothing the structure of the input space. This smoothing means that the mapping function is easier for the network to learn, resulting in better and faster learning.
- Thus, we observe that input noise and weight noise encourage the neural network output to be a smooth function of the input or its weights, respectively.

Adding noise to different network types

- Adding noise during training is a generic method. It can be used irrespective of the type of neural network that is being used.

- It was used with multilayer perceptions but can be used with convolutional and recurrent neural networks.

☞ To test the amount of noise

- One cannot know how much noise will benefit specific model on the training dataset.
- Experiment must be carried out with different amounts and different types of noise to discover what works best.
- It is advisable to carry out experiments on smaller datasets across a range of values.
- Noise is only added during the training of the model.
- One has to be sure that any source of noise is not added during the evaluation of the model, especially when the model is used to make predictions on new data.
- Although additional noise to the inputs is the most common and widely studied approach, random noise can be added to other parts of the network during training.

  Here, we mention some examples :

  (i) Add noise to activations, i.e., the outputs of each layer.

  (ii) Add noise to weights, i.e., an alternative to the inputs.

  (iii) Add noise to the gradients, i.e., the direction to update weights.

  (iv) Add noise to the outputs, i.e. labels or target variables.

Chapter Ends ...

❑❑❑

MODULE 3

CHAPTER 3

Autoencoders : Unsupervised Learning

Syllabus

Introduction, Linear Autoencoder, Undercomplete Autoencoder, Overcomplete Autoencoders, Regularization in Autoencoders

Denoising Autoencoders, Sparse Autoencoders, Contractive Autoencoders

Application of Autoencoders: Image Compression.

3.1 DIFFERENT TYPES OF AUTOENCODERS

3.1.1 Introduction

GQ. What is an autoencoder.
GQ. What is the need of an autoencoder. (2 Marks)
GQ. What are the properties of an autoencoder. (2 Marks)
GQ. Describe the architecture of an autoencoder. (3 Marks)
GQ. What are the different types of autoencoders. (4 Marks)
(4 Marks)

- A typical workflow in a Machine Learning project is designed in a supervised manner. We tell the algorithm what to do and what not to do. This generally gives a structure for solving a problem, but it limits the potential of that algorithm in two ways : **It is bound by the biases in which it is being supervised.**
- Of course it learns how to perform that task on its own, but it is prohibited to think of other corner cases that could occur when solving the problem. As the learning is supervised, **there is a huge manual effort involved in creating the labels for our algorithm.** So fewer the manual labels you create, less is the training that you can perform for your algorithm.
- To solve this issue in an intelligent way, we can use unsupervised learning algorithms. These algorithms derive insights directly from the data itself, and work as summarizing the data or grouping it, so that we can use these insights to make data driven decisions. Let's take an example to better understand this concept.
- Let's say a bank wants to divide its customers so that they can recommend the right products to them. They can do this in a data-driven way, by segmenting the customers on the basis of their ages and then deriving insights from these segments. This would help the bank give better product recommendations to their customers, thus increasing customer satisfaction.
- Let us understand another usecase: I have 2000+ photos in my smartphone right now. If I had been a selfie freak, the photo count would easily be 10 times more. Sifting through these photos is a nightmare, because every third photo turns out to be unnecessary and useless for me. I'm sure most of you will be able to relate to my plight!
- Ideally, what I would want is an app which organizes the photos in such a manner that I can go through most of the photos and have a peek at it if I want. This would actually give me context as such of the different kinds of photos I have right now. To get a clearer perspective of the problem, I went through my mobile and tried to identify the categories of the images by myself. Here are the insights I gathered:
- First and foremost, I found that one-third of my photo gallery is filled with memes. I personally collect interesting quotes / shares I come across on Reddit.

- There are a few photos of whiteboard discussions that happen frequently during meetings. Then there are a few images/screenshots of code tracebacks/bugs that require internal team discussions. They are a necessary evil that has to be purged after use. I also found dispersed "private & personal" images, such as selfies, group photos and a few objects/sceneries. They are few, but they are my prized possessions. Last but not the least – there were numerous "good morning", "happy birthday" and "happy diwali" posts that I desperately want to delete from my gallery. No matter how much I exterminate them, they just keep coming back!
- We will discuss a few approaches I have come up with to solve this problem. Arrange on the basis of time, arrange on the basis of location, extract semantics from the image and use it to define my collection. So our algorithm should ideally capture this information without explicitly tagging what is present and what is not, and use it to organize and segment our photos. This approach is what we call an "unsupervised way" to solve problems. We did not directly define the outcome that we want. Instead, we trained an algorithm to find those outcomes for us! Our algorithm summarizes the data in an intelligent manner, and then tries to solve the problem on the basis of these inferences. Pretty cool, right?
- Now you may be wondering – how can we leverage Deep Learning for unsupervised learning problems? As we saw in the case study above, by extracting semantic information from the image, we can get a better view of the similarity of images. Thus, our problem can be formulated as – how can we reduce the dimensions of an image so that we can reconstruct the image back from these encoded representations?
- Supervised learning uses explicit labels in order to train a network while unsupervised learning relies only on data. e.g. of supervised learning is image classification while of unsupervised learning is image compression. Autoencoders are deterministic and unsupervised training models that learn a mapping from a dataset to itself. The encoding weights map the data into a latent space and decoding weights map the latent representation (generally a compressed representation) back to the original space – the weights are learned such that the reconstruction error is minimized.

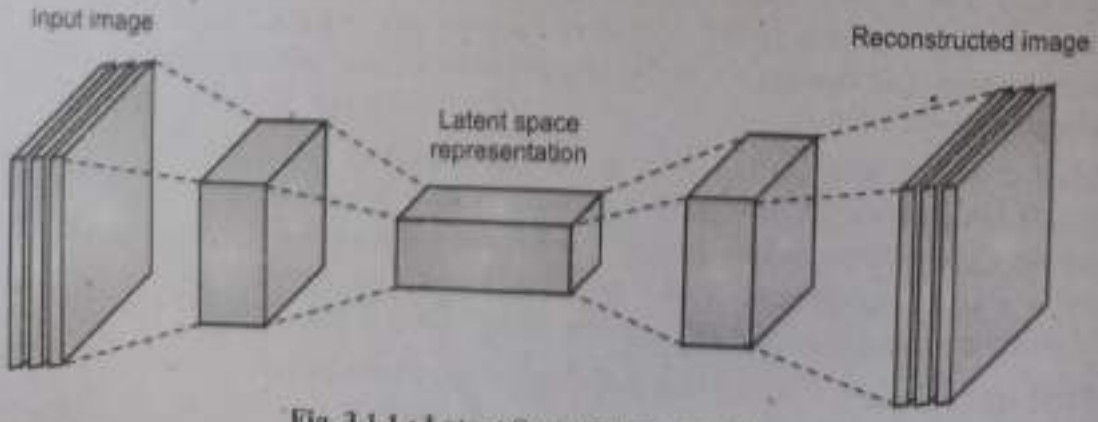


Fig. 3.1.1 : Latent Space Representation

- An autoencoder is a type of feedforward neural network used to learn efficient codings of unlabeled data (unsupervised learning). Autoencoders are designed to reproduce their input esp. for images from a learned encoding. Autoencoders are trained the same way as ANNs via backpropagation.
- Here the input is same as output. An autoencoder learns two functions: an encoding function that transforms the input data, and a decoding function that recreates the input data from the encoded representation. They compress the input into a lower dimensional code and then construct the output from this representation. The code is a compact "summary" or "compression" of the input, also called the *latent-space representation*.
- An autoencoder consists of 3 components: encoder, code and decoder. The encoder compresses the input and produces the code, the decoder then reconstructs the input only using this code. Encoder compresses input into a latent-space (h) of usually smaller dimension. $h = f(x)$ while Decoder reconstructs input from the latent space. $r = g(f(x))$ with r as close to x as possible.
- Autoencoders map high dimensional data to 2 d for better visualization. They are used for compression. Since unlabelled data availability is much more than labelled data, autoencoders learn abstract features in an unsupervised way so you can apply them to a supervised task.
- Basically, autoencoders are approximators for the identity operation; therefore, learning these weights might seem trivial; but by constraining the parameters (such as number of nodes or number of connections), interesting representations can be uncovered in the data.
- Most real datasets are structured i.e. they have a high degree of local correlations; usually, the autoencoder is able to exploit these correlations and yield compressed representations. However autoencoders are not usually used for compression, rather they are used for learning the representations which are later used for classification i.e. for feature learning.
- In the past, these structures have paved way for compression of natural signals like image, speech, and regularized Autoencoders. Modeling these structures has also benefitted Machine Learning, e.g. the success of PCA in classification is a well-known example.
- Feature representation is one of the key factors in any Machine Learning problem; the goal is to identify features so that they best represent the data – Autoencoders learn features so that the Euclidean norm is minimized.

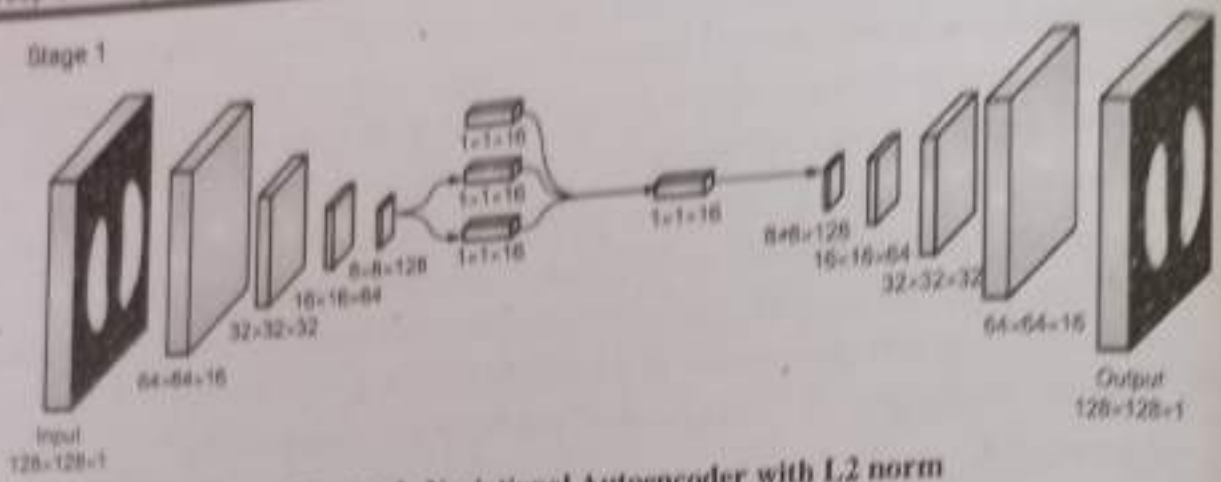


Fig. 3.1.2 :Variational Autoencoder with L2 norm

- To build an autoencoder we need 3 things: an encoding method, a decoding method, and a loss function to compare the output with the target. Autoencoders are mainly a dimensionality reduction (or compression) algorithm with a couple of important properties:
  - **Data-specific :** Autoencoders are only able to meaningfully compress data similar to what they have been trained on. Since they learn features specific for the given training data, they are different than a standard data compression algorithm like gzip. So we can't expect an autoencoder trained on handwritten digits to compress landscape photos.
  - **Lossy :** The output of the autoencoder will not be exactly the same as the input, it will be a close but degraded representation. If you want lossless compression they are not the way to go.
  - **Unsupervised :** To train an autoencoder we don't need to do anything fancy, just throw the raw input data at it. Autoencoders are considered an unsupervised learning technique since they don't need explicit labels to train on. But to be more precise they are self-supervised, because they generate their own labels from the training data.
- Autoencoders can have a larger number of nodes in the representation layer than those at the input / output. It is believed that such networks can learn non-linear relationships in the data; however learning so many parameters requires a large amount of training data, which is not always available. This leads to overfitting. In recent times, instead of architecture with a large number of nodes in a single hidden layer, researchers are going deeper.
- Deep architectures too have a large number of parameters to learn, but the advantage is that the parameters can be learned layer-wise; hence can be learned with limited training samples. It is well known how successful deep architectures have been for a variety of problems arising in image classification and speech processing.

- Proponents of Deep Learning believe that going deeper into the architecture captures more abstract and difficult representations and hence yield lower classification errors.
- A basic Autoencoder learns the encoding and decoding weights by minimizing the l2–norm between the input (training samples) and the output (training samples /corrupted training samples). Over the time several variations have been proposed.
- Let us understand the architecture of an Autoencoder. First the input passes through the encoder, which is a fully-connected ANN, to produce the code. The decoder, which has the similar ANN structure, then produces the output only using the code.
- The goal is to get an output identical with the input. Note that the decoder architecture is the mirror image of the encoder. The only requirement is the dimensionality of the input and output needs to be the same. Anything in the middle can be played with.

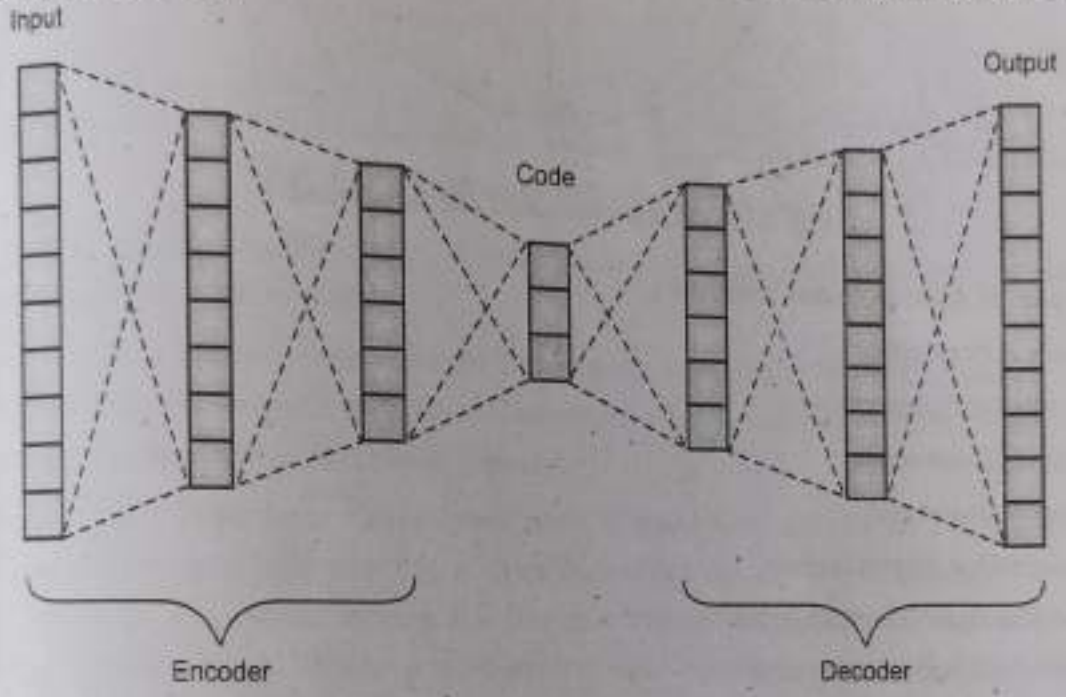


Fig. 3.1.3 : Autoencoder Architecture diagram

There are 4 hyperparameters that we need to set before training an autoencoder:

- **Code size :** The number of nodes in middle layer. Smaller size results in more compression.
- **Number of layers :** The autoencoder can be as deep as we like. In the figure above we have 2 layers in both the encoder and decoder, without considering the input and output.
- **Number of nodes per layer :** The autoencoder architecture shown above is called a *stacked autoencoder* since the layers are stacked one after another. Usually stacked autoencoders look like a "sandwitch". The number of nodes per layer decreases with each subsequent layer of the encoder, and increases back in the decoder. Also the decoder is symmetric to the encoder in terms of layer structure.

- **Loss function :** If the input values are in the range [0, 1] then we typically use crossentropy, otherwise we use the mean squared error (MSE).

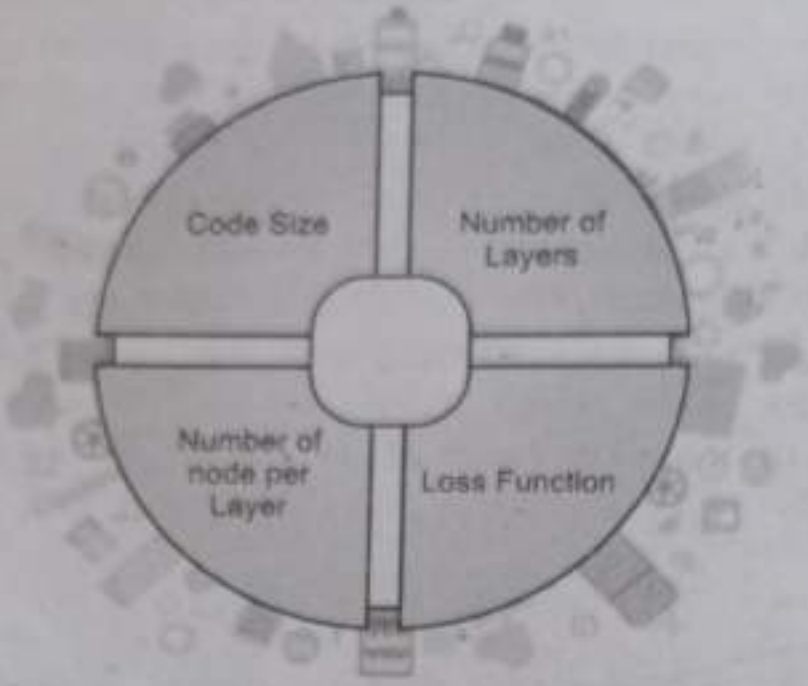


Fig. 3.1.4 : Hyperparameters of Autoencoder

Types of Autoencoder include :

1. Stack autoencoder
2. Denoising Autoencoder
3. Sparse Autoencoder
4. Deep Autoencoder
5. Contractive Autoencoder
6. Undercomplete Autoencoder
7. Convolutional Autoencoder
8. Variational Autoencoder

3.1.2 Linear Autoencoder

GQ. Explain Linear autoencoder with diagram. **(3 Marks)**
GQ. How autoencoders differ from PCA. **(3 Marks)**
GQ. Demonstrate how autoencoders work with Movie Recommendation data **(3 Marks)**

- For a linear autoencoder, the encoder and decoder are composed of only linear layers. The number of nodes in the input layer and the output layer of the encoder and decoder, respectively, should be equal to the dimensionality of the input data and the desired dimensionality of the low-dimensional representation.

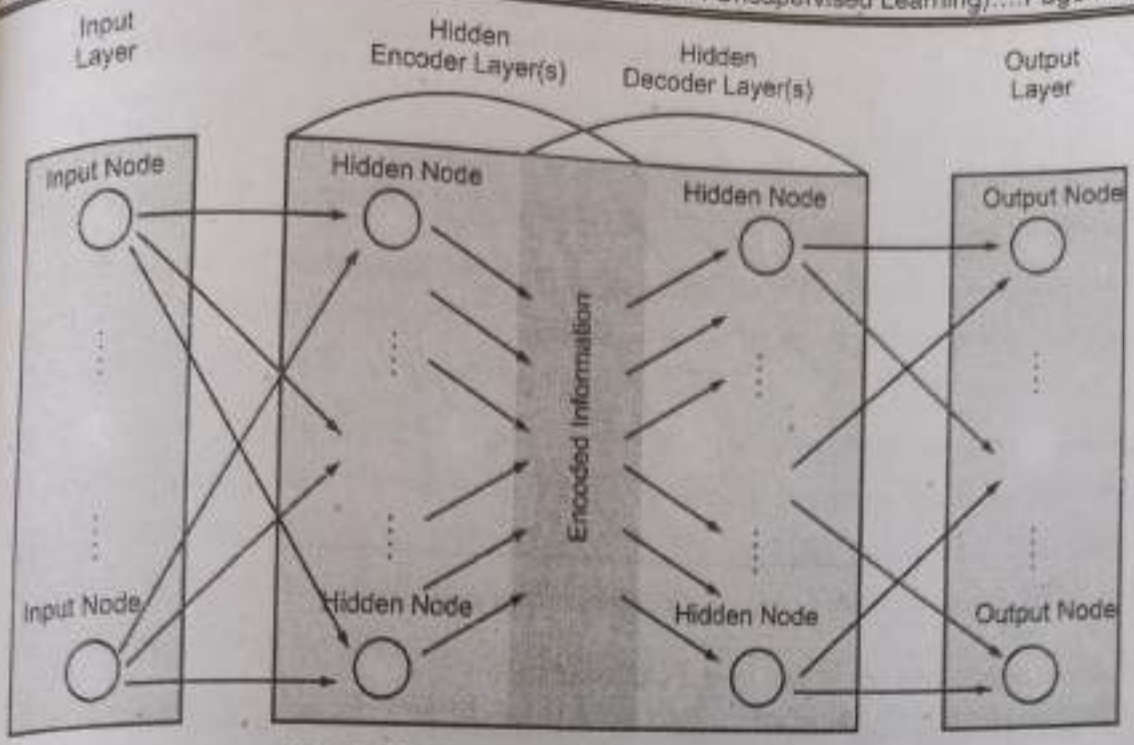


Fig. 3.1.5 : Linear Autoencoder Architecture

- As with other Neural Networks, overfitting is a problem with autoencoders when the capacity is too large for data.
- Autoencoders address this through bottle neck layer with fewer degrees of freedom than in possible outputs, denoise autoencoder, regularized autoencoder with sparsity and contractive autoencoder with penalty.
- Autoencoders differ from dimensionality reduction techniques PCA / SVD as collection of vectors (images) and produce a usually smaller set of vectors that can be used to approximate the input vectors via linear combination. They are very efficient for certain applications. While autoencoders can learn nonlinear dependencies, use convolution layer and transfer learning.
- The encoder part of an autoencoder is equivalent to PCA. PCA is an autoencoder if you are using a Linear autoencoder, if you are using a squared error loss function and if you normalize the inputs to this.
- PCA features are totally linearly uncorrelated with each other since features are projections onto orthogonal vector space. But autoencoded features might have correlation since they are just trained for accurate reconstruction.
- PCA is faster and computationally cheaper than autoencoders. Autoencoder is prone to overfitting due to high number of parameters though regularization and careful design can avoid this.

Linear Vs nonlinear dimensionality reduction

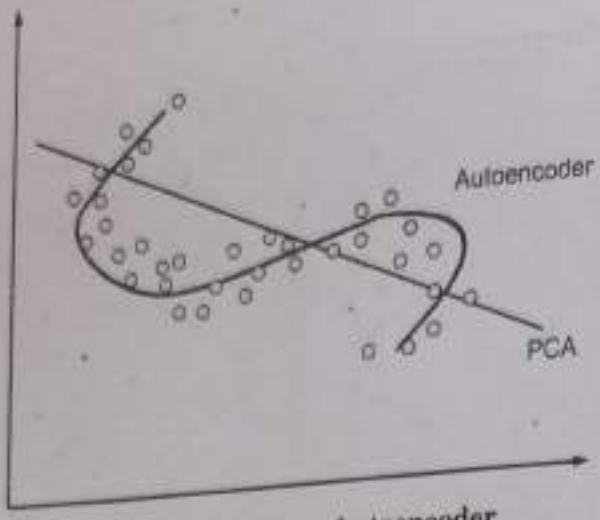


Fig. 3.1.6 : PCA vs Autoencoder

Autoencoders are preferred over PCA because :

- An autoencoder can learn non-linear transformations with a non-linear activation function and multiple layers.
- It doesn't have to learn dense layers. It can use convolutional layers to learn which is better for video, image and series data.
- It is more efficient to learn several layers with an autoencoder rather than learn one huge transformation with PCA.
- An autoencoder provides a representation of each layer as the output.
- It can make use of pre-trained layers from another model to apply transfer learning to enhance the encoder/decoder.

Hidden layer uses tanh activation function [– 1, 1] while sigmoid and softmax activation function [0,1] can be used in Output layer. Eg. Movie **recommendation**. Softmax function converts max value to 1 other to 0. Encoder for an image input compresses data. As shown in figure below, each row is inputted to the autoencoder with 5 input neurons. Important features are propogated to hidden nodes, loss function is computed at output layer and backpropogated for weight adjustment. Training continues with further data.

Table 3.1.1 : Movie rating data

	Movie 1	Movie 2	Movie 3	Movie 4	Movie 5	Movie 6
User 1	1	0		1	1	1
User 2	0	1	0	0	1	0
User 3		1	1	0	0	
User 4	1	0	1	1	0	1
User 5	0		1	1		1

3.1.3 Undercomplete Autoencoder

GQ. What is the need of an undercomplete autoencoder. (2 Marks)

GQ. Draw the architecture of undercomplete autoencoder. (2 Marks)

- Undercomplete Autoencoder has fewer nodes (dimensions) in the middle compared to Input and Output layers which helps to obtain important features from the data. In such setups, we tend to call the middle layer a "bottleneck".
- In the case of Undercomplete Autoencoders, we are squeezing the information into fewer dimensions (hence the bottleneck) while trying to ensure that we can still get back to the original values. Therefore, we are creating a custom function that compresses the data, which is a way to reduce the dimensionality and extract meaningful information.
- After training the Undercomplete Autoencoder, we typically discard the Decoder and only use the Encoder part.
- The objective of undercomplete autoencoder is to capture the most important features present in data. It minimizes the loss function by penalising the g(f(x)) for being different from the input x.

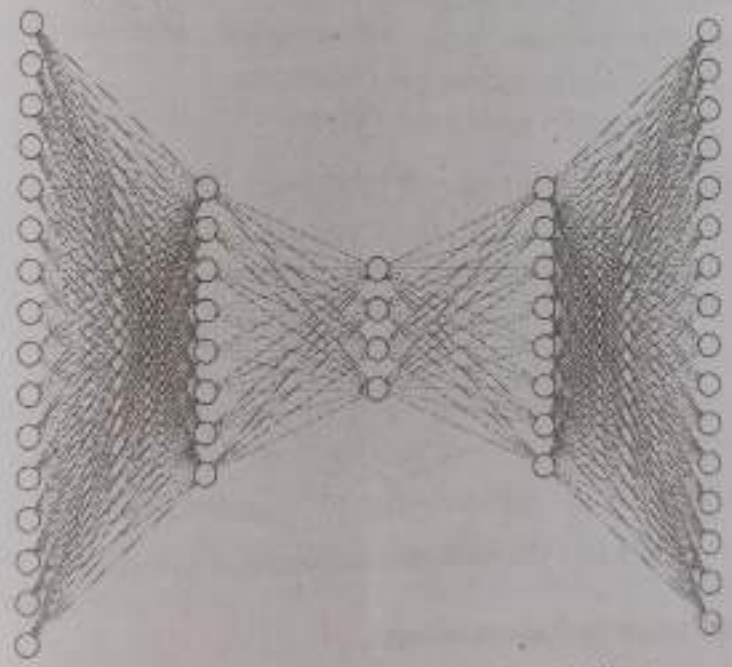


Fig. 3.1.7 : Undercomplete autoencoder Architecture

- **Creating a simple bottleneck in Keras** maps 28 × 28 images into a 32 dimensional vector. We can also use more layers and/or convolutions.

```
input_img = Input(shape=(784,))
encoding_dim = 32
encoded = Dense(encoding_dim, activation='relu')(input_img)
decoded = Dense(784, activation='sigmoid')(encoded)
autoencoder = Model(input_img, decoded)
```

- The figure below shows the I/P and O/P of undercomplete autoencoder.

Fig. 3.1.8 : Undercomplete autoencoder I/P and O/P

3.1.4 Overcomplete Autoencoder

GQ. What is the need of an overcomplete autoencoder. **(2 Marks)**

GQ. Draw the architecture of overcomplete autoencoder. **(2 Marks)**

- Overcomplete Autoencoder has more nodes (dimensions) in the middle compared to Input and Output layers.
- While poor generalization can happen even in undercomplete autoencoders, it is an even more serious problem with overcomplete autoencoders. To avoid poor generalization, we need to introduce generalization.

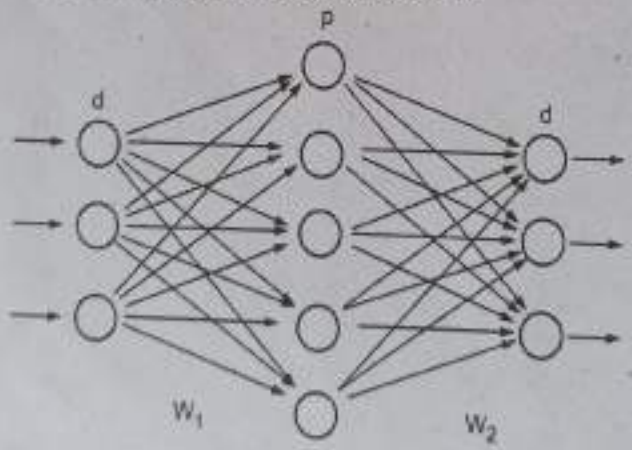


Fig. 3.1.9 : Overcomplete autoencoder Architecture

3.1.5 Regularization in Autoencoders

GQ. Explain regularization autoencoder with diagram. **(3 Marks)**

GQ. Describe regularization techniques that can be applied to autoencoders. **(5 Marks)**

- If training data is not sufficient, all Machine Learning tools tend to overfit.

- One of the ways to combat overfitting in neural networks is based on randomly switching off and on the nodes i.e. Regularized Autoencoders. Egs. Include DropOut, switching on and off the connections i.e. DropConnect or both DropAll. These are stochastic regularization techniques and work well with statistical generative models.
- They have been successfully used in neural networks, but to the best of our knowledge have not been used in autoencoders which have a deterministic formulation. For deterministic formulations the most straightforward way to combat overfitting is to regularize the cost function.
- Regularization helps with the effects of out-of-control parameters by using different methods to minimize parameter size over time. In mathematical notation, we see regularization represented by the coefficient lambda, controlling the trade-off between finding a good fit and keeping the value of certain feature weights low as the exponents on features increase.
- Regularization coefficients L1 and L2 help fight overfitting by making certain weights smaller. Smaller-valued weights lead to simpler hypotheses, which are the most generalizable. Unregularized weights with several higher-order polynomials in the feature sets tend to overfit the training set.

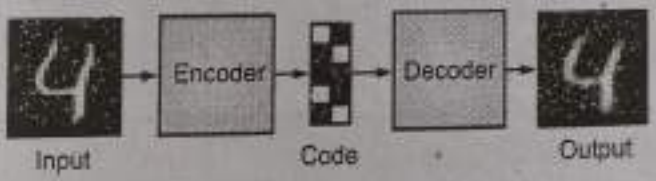


Fig. 3.1.10 : Regularization Autoencoder

- Regularization techniques are often used to prevent overfitting, which occurs when the model learns the noise in the data instead of the underlying pattern. Regularization aims to impose constraints on the model parameters to reduce the model's flexibility and improve generalization performance. In the context of autoencoders, regularization techniques can be used to prevent overfitting and improve the quality of the learned representation. There are various types of regularization techniques that can be applied to autoencoders, including :
  - **L1 and L2 regularization :** These techniques add a penalty term to the loss function that encourages the model to have smaller weights. This can help to prevent overfitting and improve the sparsity of the learned representation.

    L1 norm: $|a| = \text{sqrt}(a1^2 + a2^2 \ldots + an^2)$
  - **Dropout :** Dropout is a technique that randomly drops out neurons during training to prevent overfitting. This can be applied to the
  - **Batch normalization :** Batch normalization is a technique that normalizes the input to each layer to have zero mean and unit variance. This can help to improve the stability of the training process and prevent overfitting. Batch normalization

layer calculates the mean and standard deviation for the previous layer, for the current batch of training instances. It then subtracts the mean and divides by the standard deviation, thus normalizing the layer's output (for the batch). In feed-forwards after the training, the layer will start normalizing using parameters estimated from the whole training dataset. This makes the model converge a lot faster, since it becomes less sensitive to changes in the distribution of the inputs, or the hidden layers. It will also become more robust to natural distribution changes in the inputs, being more sustainable in the long term on a productive environment.

- **Data augmentation :** Data augmentation involves applying transformations to the input data, such as cropping or flipping, to increase the size of the training set and improve generalization performance. Eg. for images, we can use:
- **Geometric transformations :** you can randomly flip, crop, rotate or translate images, and that is just the tip of the iceberg.
- **Color space transformations :** change RGB color channels, intensify any color.
- **Kernel filters :** sharpen or blur an image.

- Overall, regularization techniques can be an effective way to improve the performance of autoencoders and prevent overfitting. The choice of regularization technique will depend on the specific problem and data set being considered.

3.2 DEEP INSIDE : AUTOENCODERS

GQ. How can we use autoencoder in image denoising application. **(2 Marks)**

GQ. What is the advantage of denoising autoencoder. **(2 Marks)**

GQ. Explain the concept of sparse autoencoder in detail. How can we use it to avoid overfitting? **(5 Marks)**

GQ. What is the advantage of sparse autoencoder. **(2 Marks)**

GQ. What is the need of contractive autoencoders. **(2 Marks)**

GQ. What is the advantage of contractive autoencoder. **(2 Marks)**

3.2.1 Denoising Autoencoder

- One of the simpler variations of autoencoder is the denoising autoencoder, where the inputs are corrupted and the outputs are clean; the autoencoder basically learns to clean corrupted samples. Such denoising autoencoders can generate more robust representation which improves classification.
- Autoencoder learns useful features by adding random noise to its inputs and making it recover the original noise-free data. This way the autoencoder can't simply copy the input to its output with learning the features in data because the input also contains random noise.

- We are asking it to subtract the noise and produce the underlying meaningful data. This is called a *denoising autoencoder*.
- The top row contains the original images. We add random Gaussian noise to them and the noisy data becomes the input to the autoencoder. The autoencoder doesn't see the original image at all. But then we expect the autoencoder to regenerate the noise-free original image.
- The bottom row is the autoencoder output. We can do better by using more complex autoencoder architecture, such as convolutional autoencoders.

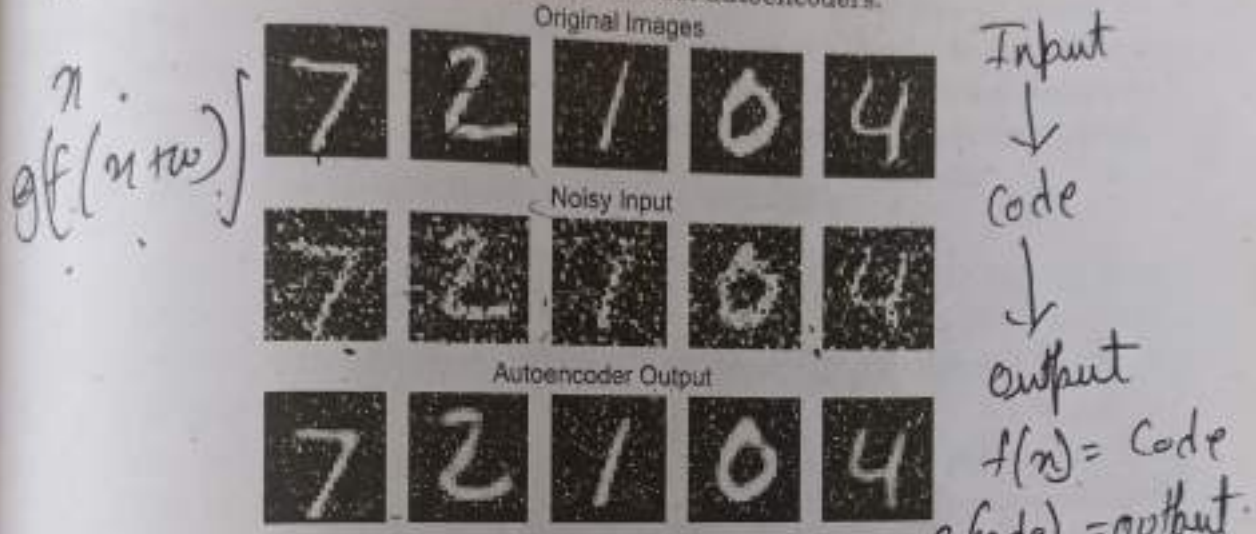


Fig. 3.2.1 : Original image, noisy I/P and O/P

- Basic autoencoder trains to minimize the loss between x and the reconstruction g(f(x)). Denoising autoencoders train to minimize the loss between x and g(f(x + w)), where w is random noise. Denoising autoencoders can't simply memorize the input output relationship.
- Intuitively, a denoising autoencoder learns a projection from a neighborhood of our training data back onto the training data. In figure below, noise is added to original image, it is encoded into the code which is decoded to return the original input image.

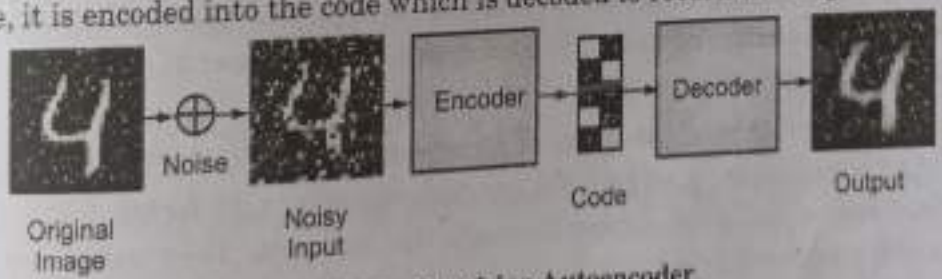


Fig. 3.2.2 : Denoising Autoencoder

- **Advantage of denoising autoencoder :** It is simpler to implement. It requires adding one or two lines of code to regular autoencoder. There is no need to compute Jacobian of hidden layer.

3.2.2 Sparse Autoencoder

- Regularisation is also used to learn useful features apart from keeping the code size small and denoising autoencoders.
- We can regularize the autoencoder by using a *sparsity constraint* such that only a fraction of the nodes would have nonzero values, called **active nodes**.
- In particular, we add a penalty term to the loss function such that only a fraction of the nodes become active. This forces the autoencoder to represent each input as a combination of small number of nodes, and demands it to discover interesting structure in the data. This method works even if the code size is large, since only a small subset of the nodes will be active at any time.

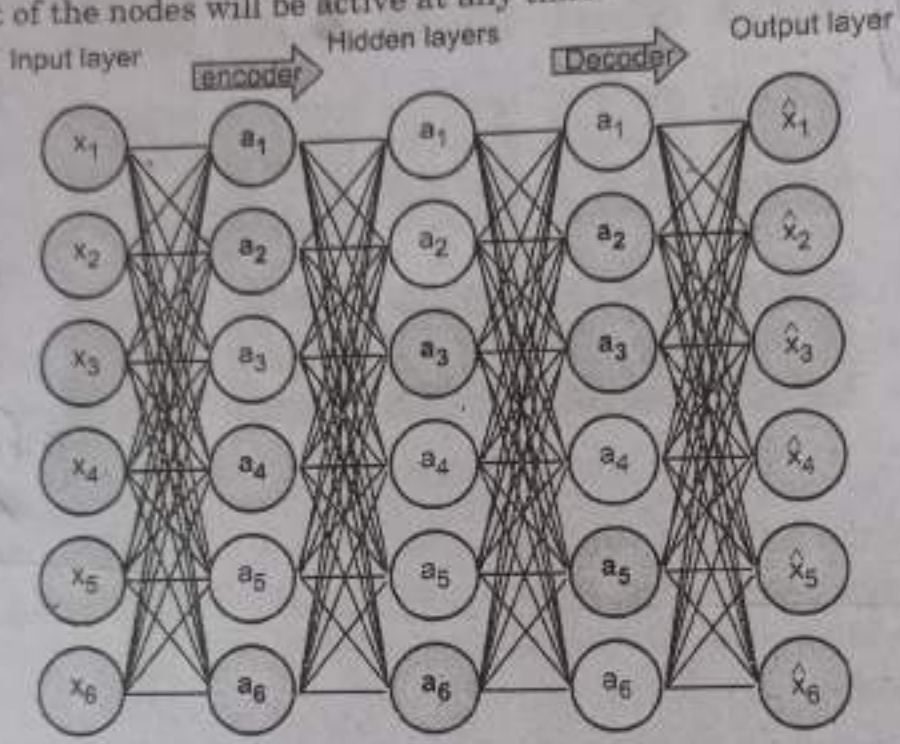


Fig. 3.2.3 : Sparse Autoencoder Architecture

- Sparsity was introduced in terms of firing neurons, if the neurons are of high value (near about 1), it is allowed to be fired, the rest are not. Sparse autoencoders construct a loss function to penalize activations within a layer.
- They usually regularize the weights of a network and not the activations. Individual nodes of a trained model that activate are data-dependent. Different inputs will result in activations of different nodes through the network. They selectively activate regions of the network depending on the input data. Egs.
  - **L1 Regularization :** Penalize the absolute value of the vector of activations a in layer h for observation I.
  - **KL divergence :** Use cross-entropy between average activation and desired activation.

- A hidden neuron with sigmoid activation function will have values 0 and 1. We say that neuron is activated when its output is close to 1 and not activated when its output is close to 0. A sparse autoencoder tries to ensure that neuron is inactive most of the times. Sparse autoencoders have hidden nodes greater than input nodes.
- Sparsity may be introduced by additional terms in the loss function during the training process, either by comparing the probability distribution of the hidden unit activations with some low desired value or by manually zeroing al but the strongest hidden unit activations.
- **Advantage of sparse autoencoder :** We can achieve an information bottleneck (same information with fewer neurons) without reducing the number of neurons in the hidden layers.

3.2.3 Contractive Autoencoder

- A contractive autoencoder is less sensitive to slight variations in the training dataset as it provides a robust learned representation. We can achieve this by adding a penalty term or regularizer to whatever cost or objective function or loss function the algorithm is trying to minimize.
- The result reduces the learned representation's sensitivity towards the training input. This regularizer needs to conform to the Frobenius norm of the Jacobian matrix for the encoder activation sequence concerning the input.
- Frobenius norm of the Jacobian matrix for the hidden layer is calculated w.r.t. the input and is basically the sum of square of all elements as in Figure below. If this value is zero, we don't observe any change in the learned hidden representations as we change the input values. But if the value is huge, then the learned model is unstable as the input values change.
- We generally employ Contractive autoencoders as one of several other autoencoder nodes. It is in active mode only when other encoding schemes fail to label a data point.

Frobenius Norm – Vector norm, L2 norm

$$||A||_F = \sqrt{\sum_{j=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}$$

Jacobian matrix – matrix of all first-order partial derivatives of a vector-valued function.

Regularizing term : $|| J_h(x) ||_F^2 = \Sigma_{ij} \left(\frac{\delta h_j(x)}{\delta z_i} \right)$

- Contractive autoencoders arrange for similar inputs to have similar activations. i.e., the derivative of the hidden layer activations are small with respect to the input.

- Denoising autoencoders make the reconstruction function (encoder + decoder) resist small perturbations of the input while Contractive autoencoders make the feature extraction function (ie. encoder) resist infinitesimal perturbations of the input.

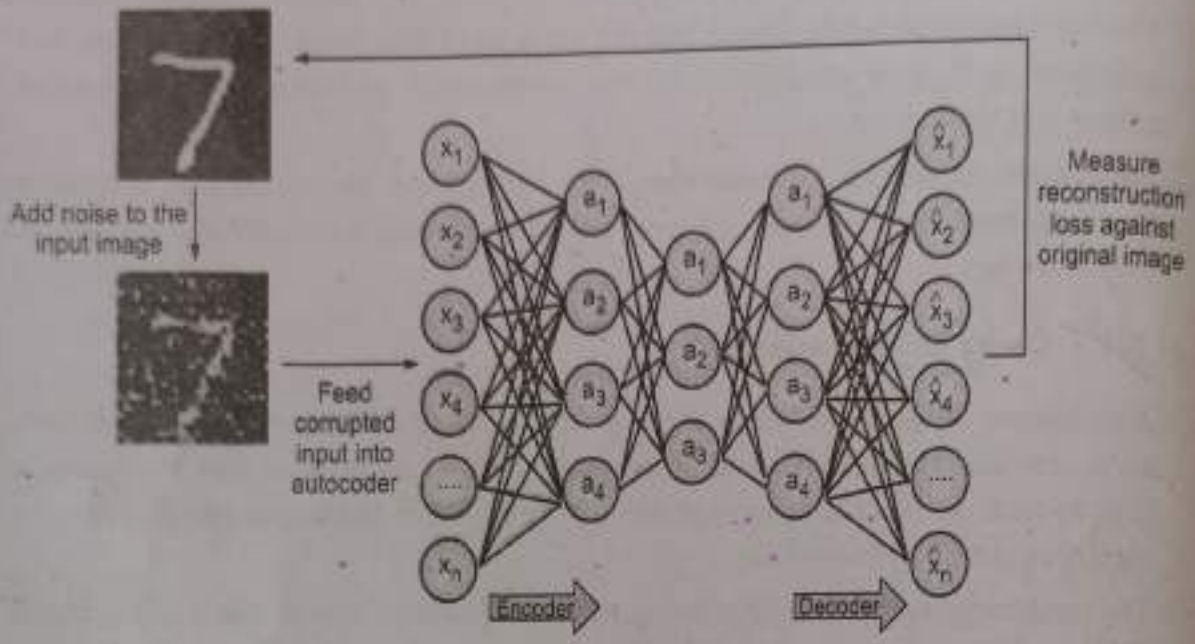


Fig. 3.2.4 : Contractive Autoencoder

Advantage of contractive autoencoder

Since gradient is deterministic, we can use second order optimizers e.g. conjugate gradient, LBFGS, etc. which might be more stable than denoising autoencoder, and it uses a sampled gradient.

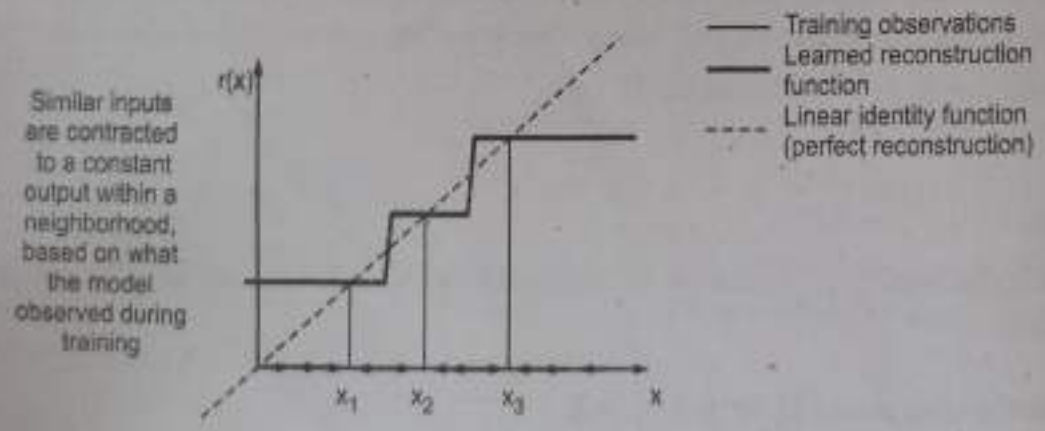


Fig. 3.2.5 : Contractive Autoencoder

3.3 APPLICATION

GQ. Write applications of autoencoders (10 Marks)

For classification or regression tasks, auto-encoders can be used to extract features from the raw data to improve the robustness of the model. They can also be used for :

- **Dimensionality reduction :** Can also learn complex non-linear relationships in the data.
- **Feature extraction :** Take un-labeled data and learn efficient codings about the structure of the data that can be used for supervised learning tasks.
- **Image denoising :** The noise present in the images may be caused by various intrinsic or extrinsic conditions which are practically hard to deal with. The problem of Image Denoising is a very fundamental challenge in the domain of Image processing and Computer vision.
- **Image compression :** Learns how to compress the data efficiently by encoding the actual data and then reconstruct the data back from the compressed data
- **Image search :** Can be used for finding similar images in an unlabeled image dataset. Google reverse image serach.
- **Anomaly detection :** The reconstruction errors are used as the anomaly scores.
- **Missing value imputation :** Learn a representation of the data with missing values and generate plausible new ones to replace them.
- Image generation
- Sequence to Sequence prediction :
- Recommendation System :

Among these we see in detail the image compression application of autoencoders.

3.3.1 Image Compression

GQ. Which autoencoder is used for image compression tasks. Justify. (5 Marks)

GQ. State autoencoder architecture for image compression task. (5 Marks)

- Artificial Intelligence encircles a wide range of technologies and techniques that enable computer systems to solve problems like Data Compression which is used in Computer Vision, Computer Networks, Computer Architecture, and many other fields.
- **Autoencoders** are *unsupervised neural networks* that use Machine Learning to do this compression for us. The raw input image can be passed to the encoder network and obtain a compressed dimension of encoded data.

- The autoencoder network weights can be learned by reconstructing the image from the compressed encoding using a decoder network as in Fig. 3.3.1.

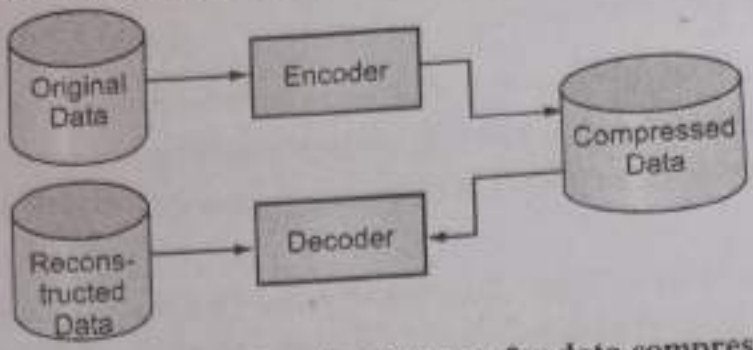


Fig. 3.3.1 : Encoder Decoder diagram for data compression

- Autoencoders are a Deep Learning model for transforming data from a high-dimensional space to a lower-dimensional space.
- They work by encoding the data, whatever its size, to a 1-D vector. This vector can then be decoded to reconstruct the original data (in this case, an image).
- The more accurate the autoencoder is, the closer the generated data is to the original. We will explore the autoencoder architecture and see how we can apply this model to compress images from the MNIST dataset using TensorFlow and Keras.
- The input layer is then propagated through a number of layers :
  - Dense layer with 300 neurons
  - LeakyReLU layer
  - Dense layer with 2 neurons
  - LeakyReLU layer
- The last Dense layer in the network has just two neurons. When fed to the LeakyReLU layer, the final output of the encoder will be a 1-D vector with just two elements.
- Similar to building the encoder, the decoder will be build using the following code. Because the input layer of the decoder accepts the output returned from the last layer in the encoder, we have to make sure these 2 layers match in the size. The last layer in the encoder returns a vector of 2 elements and thus the input of the decoder must have 2 neurons. You can easily note that the layers of the decoder are just reflection to those in the encoder.

```
decoder_input = tensorflow.keras.layers.Input(shape=(2), name="decoder_input")

decoder_dense_layer1 = tensorflow.keras.layers.Dense(units=300,
name="decoder_dense_1")(decoder_input)
decoder_activ_layer1 =
tensorflow.keras.layers.LeakyReLU(name="decoder_leakyrelu_1")(decoder_dense_layer1)
```

```
decoder_dense_layer2 = tensorflow.keras.layers.Dense(units=784,
name="decoder_dense_2")(decoder_activ_layer1)
decoder_output =
tensorflow.keras.layers.LeakyReLU(name="decoder_output")(decoder_dense_layer2)
decoder = tensorflow.keras.models.Model(decoder_input, decoder_output,
name="decoder_model")
```

- The most common type of Machine Learning models is discriminative. If you're a Machine Learning enthusiast, it's likely that the type of models that you've built or used have been mainly discriminative.
- These models recognize the input data and then take appropriate action. For a classification task, a discriminative model learns how to differentiate between various different classes.
- Based on the model's learning about the properties of each class, it classifies a new input sample to the appropriate label. Let's apply this understanding to the next image representing a warning sign.
- If a Machine/Deep Learning model is to recognize the following image, it may understand that it consists of three main elements: a rectangle, a line, and a dot. When another input image has features which resemble these elements, then it should also be recognized as a warning sign.

Fig. 3.3.2 : Sample Image

- If the algorithm is able to identify the properties of an image, could it generate a new image similar to it? In other words, could it draw a new image that has a triangle, a line, and a dot? Unfortunately, discriminative models are not clever enough to draw new images even if they know the structure of these images. Let's take another example to make things clearer.
- Assume there is someone that can recognize things well. For a given image, he/she can easily identify the salient properties and then classify the image. Is it a must that such a person will be able to draw such an image again? No.

- Some people cannot draw things. Discriminative models are like those people who can just recognize images, but could not draw them on their own.
- In contrast with **discriminative models**, there is another group called **generative models** which can create new images. For a given input image, the output of a discriminative model is a class label; the output of a generative model is an image of the same size and similar appearance as the input image.
- One of the simplest generative models is the autoencoder (AE for short). Autoencoders are a Deep Neural Network model that can take in data, propagate it through a number of layers to condense and understand its structure, and finally generate that data again.
- To accomplish this task an autoencoder uses two different types of networks. The first is called an **encoder**, and the other is the **decoder**. The decoder is just a reflection of the layers inside the encoder. Let's clarify how this works.
- The job of the encoder is to accept the original data (e.g. an image) that could have two or more dimensions and generate a single 1-D vector that represents the entire image.
- The number of elements in the 1-D vector varies based on the task being solved. It could have 1 or more elements. The fewer elements in the vector, the more complexity in reproducing the original image accurately.
- By representing the input image in a vector of relatively few elements, we actually compress the image. For example, the size of each image in the MNIST dataset is 28x28. That is, each image has 784 elements. If each image is compressed so that it is represented using just two elements, then we spared 782 elements and thus (782/784)*100=99.745% of the data.
- The next figure shows how an encoder generates the 1-D vector from an input image. The layers included are of your choice, so you can use dense, convolutional, dropout, etc.

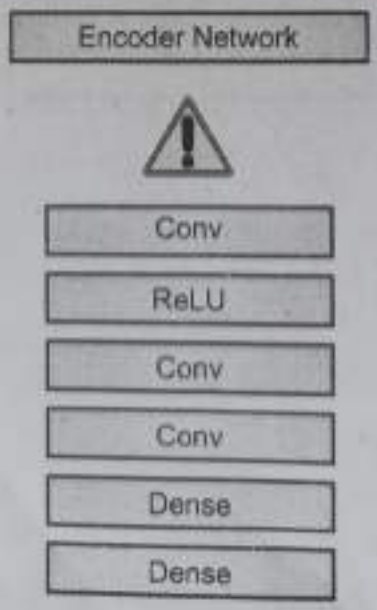


Fig. 3.3.3 : Encoder Network

- The 1-D vector generated by the encoder from its last layer is then fed to the **decoder**. The job of the decoder is to reconstruct the original image with the highest possible quality.
- The decoder is just a reflection of the encoder. According to the encoder architecture in the previous figure, the architecture of the decoder is given in the next figure.

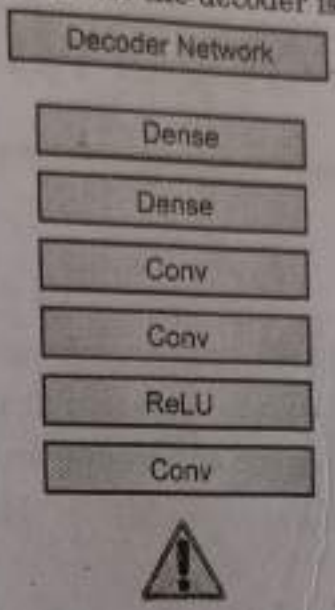


Fig. 3.3.4 : Decoder Network

- The loss is calculated by comparing the original and reconstructed images, i.e. by calculating the difference between the pixels in the 2 images. Note that the output of the decoder must be of the same size as the original image. Why? Because if the size of the images is different, there is no way to calculate the loss.

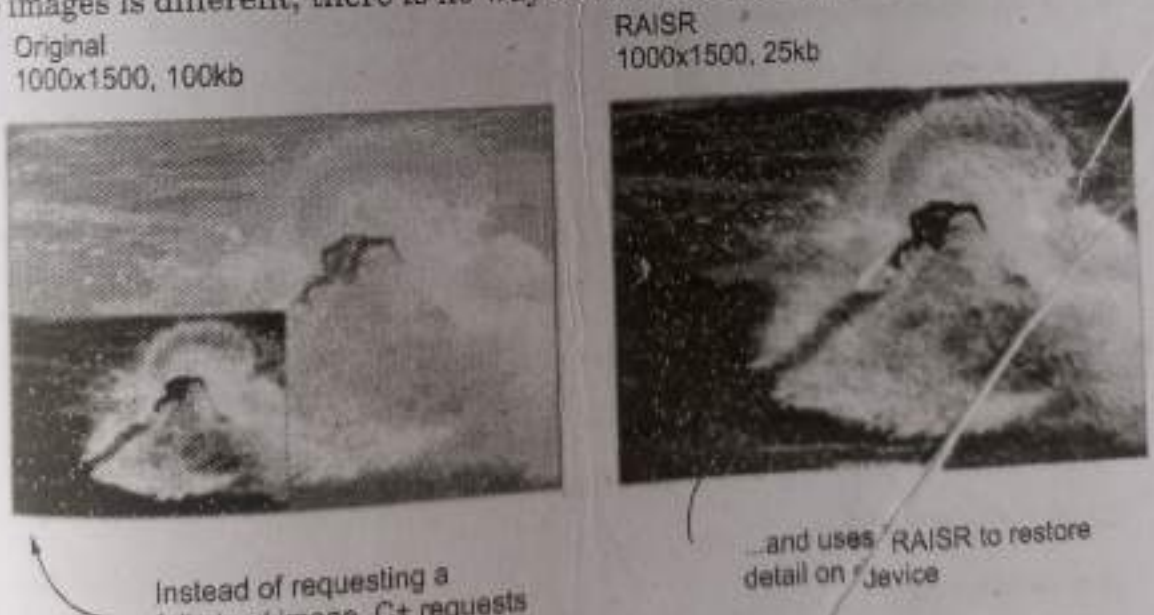


Fig. 3.3.5 : Compressed Image

Chapter Ends ...

❑❑❑

Module 4

CHAPTER 4

Convolutional Neural Networks (CNN) : Supervised Learning

Syllabus

Convolution operation, Padding, Stride, Relation between input, output and filter size. CNN architecture : Convolution layer, Pooling Layer, Weight Sharing in CNN, Fully Connected NN vs CNN. Variants of basic Convolution function, Multichannel convolution operation, 2D convolution. Modern Deep Learning Architectures : LeNET : Architecture, AlexNET : Architecture, ResNet : Architecture.

4.1 CONVOLUTION OPERATION AND MULTICHANNEL C.O.

Convolution is a **mathematical operation**. In mathematics, especially in Fourier, Laplace, Z-transforms, convolution operation paves the way to find the inverse transforms. The concept of convolution operates on two functions f and g that produces a third function (f * g), in the transformed domain.

The term convolution refers to both the **result function and the method** of **evaluating** it. It is **defined** as the **integral of product of two functions**, in which one function is shifted. The integral is evaluated for all values of shift, and gives the convolution function.

4.1.1 Cross-Correlation

Cross-correlation operation is similar to convolution operation. Consider two functions f(x) and g(x). In cross-correlation, either f(x) or g(x) is reflected about Y-axis, i.e. it is a cross-correlation of f(x) and g(– x) or f(– x) and g(x).

In case of complex valued functions, the cross-correlation operator is the adjoint of the convolution operator.

4.1.2 Applications of Convolution

GQ. Mention applications of convolution.

(1) The applications of convolution include probability, statistics, acoustics, spectroscopy, signal processing and image processing, physics, engineering, computer vision, differential equations and various transforms. Also it has applications in the field of numerical analysis and numerical linear algebra, and in the design and implementation of finite impulse response filters in signal processing.

(2) The convolution is also defined for functions on Euclidean space and other groups. For example, periodic functions, such as the 'discrete-time Fourier transform' can be defined for a given interval and convolved by periodic convolution. Note that a 'discrete convolution' can be defined for functions on the set of integers. Finding the inverse of convolution operation is called as 'deconvolution'.

4.1.3 Convolution Formulae

(1) Let f(t) and g(t) be two continuous functions defined in $(-\infty, \infty)$, then

$$(f * g)(t) = f(t) * g(t) = \int_{-\infty}^{\infty} f(u)\, g(t-u) \cdot du$$

$$= \int_{-\infty}^{\infty} f(t-u)\, g(u) \cdot du$$

Remark

(i) The symbol 't' may not represent time domain

(ii) Convolution formula represents the area f(u) weighted by the function g(– u) shifted by the amount 't'. As 't' changes the weighting function g(t – u) gives different parts of the input function f(u).

(2) Let f(t) and g(t) be defined in (0, ∞) and zero elsewhere, then convolution of f(t) and g(t) is defined as

$$(f * g)(t) = f(t) * g(t) = \int_0^t f(u)\, g(t-u) \cdot du$$

$$= \int_0^t f(t-u)\, g(u)\, du \qquad \left[\because \int_0^a f(x)\, dx = \int_0^a f(a-x)\, dx\right]$$

4.1.4 Notation

$$(f * g)(t) = \int_{-\infty}^{\infty} f(u)\, g(t-u)\, du = f(t) * g(t)$$

But $f(t) * g(t-t_0) = (f * g)(t-t_0)$

And $f(t-t_0) * g(t-t_0) = (f * g)(t-2t_0)$

(This result to be carefully noted)

4.1.5 Formation of Convolution

(i) Let f(t) and g(t) be defined in (– ∞, ∞).

(ii) Replace each function in terms of a dummy variable u,

(iii) Replace f(u) or g(u) : g(u) → g(– u)

(iv) Add a time-offset, t, which allows g(t – u) to slide along u-axis in (– ∞, ∞)

(v) The resulting waveform is the convolution of the function f and g.

4.1.6 Circular Convolution

GQ. What is circular and discrete convolution operation ?

When a function g(t) is periodic with period T, then for functions, f, such that (f * g) exists, then the convolution is also periodic and identical to :

$$(f * g)(t) = \int_{t_0}^{t0+T} \left[\sum_{K=-\infty}^{\infty} f(u + KT)\right] g(t+u)\, du,$$

where t_0 is an arbitrary choice. The summation is called a periodic summation of the function f. And (f * g) is called as circular or cyclic convolution of f and g.

4.1.7 Discrete Convolution Operation

Discrete time convolution is an operation on two discrete time signals defined by :

$$(f * g)(n) = \sum_{K=-\infty}^{\infty} f(K)\, g(n-K);$$

for all signal input functions f(n) and g(n) defined over set of integers Z.

(1) The operation is commutative,

(2) The operation of convolution is linear in each of the two function variables.

4.1.8 Properties

(1) $(K_1 f + K_2 g) * h = K_1 (f * h) + K_2 (g * h)$

(2) $f * (K_1 g + K_2 h) = K_1 (f * g) + K_2 (f * h)$

4.2 PADDING, STRIDE

Padding

GQ. Discuss padding or Write short note on Padding.

- In Convolutional Neural Networks (CNN), padding is a term that refers to the a number of pixels added to an image when it is being processed by the kernel of a CNN.
- For example, if padding in CNN is kept zero, then every pixel value that is added will be of value zero.
- Padding is simply a process of adding layers of zeros to our input images so as to avoid the problems mentioned above.
- CNNs are used extensively for tackling problems occurring in image processing and predictive modelling or classification tasks. The main application of CNN is to analyse image data.
- Now, every image in any dataset is a matrix of it's pixel values. When working with simple CNN for an image, we get output reduced in size and that is loss of data. And that hinders to obtain a proper result according to our requirements.
- When we do not want the shape or our outputs to reduce in size, the addition of more layers in the data can help to obtain a proper result and that addition can be done by padding.
- Now we study padding with it's importance and how to use it with CNN models. We also see different methods of padding and how they can be implemented.

4.2.1 Problem with Simple Convolution Layer

- A simple CNN of size (n × n) with (f × f) filter/kernel size gives result for output image as (n – f + 1) × (n – f + 1)
- For example in any convolution operation with a (8 × 8) image and (3 × 3) filter the output image size will be (6 × 6). Thus the output of the layers is shrunk in comparison to the input. Again, the filters we are using may not focus on the corners every time when it moves on the pixels e.g.

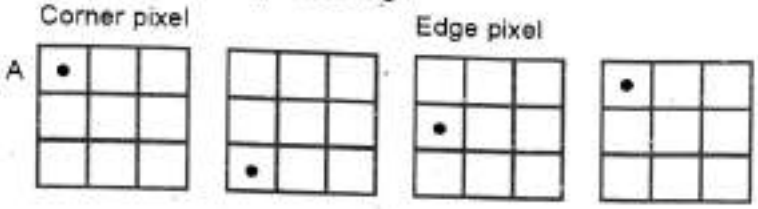


Fig. 4.2.1 : CNN structure

- The above image is an example of the movement of a filter of size (3 × 3) on an image of size (6 × 6), corner pixel A is coming under the filter in only one movement. This shown that pixel A is misinterpreted.
- This causes loss of information available in the corners and also the output from the layers is reduced and this reduced information may create confusion for next layer. This problem of model can be solved by padding layer.
- The convolution layers reduce the size of the output. So when we want to save the information presented in the corners, we can use padding layers where padding helps by adding extra rows and columns on the outer dimension of the images. So the size of the **input data** will remain similar to the output data.
- Padding basically extends the area of an image in which convolutional neural network processes. The kernel/filter which moves across the image scans each pixels and converts the image into a smaller image.
- Padding is added to the outer frame of the image to allow for more space for the filter to cover in the image. This creates a more accurate analysis of images.

4.2.2 Types of Padding

GQ. What are types of Padding.

We come across three types of padding :

(1) Same padding
(2) Valid Padding
(3) Causal padding

(1) **Same padding :** In this type of padding, the padding layers have zero values in the outer frame of the images or data so the filter we are using can cover the edge of the matrix and it carries the required inference.

(2) **Valid padding :** This type of padding is called as no padding. Here we don't apply any padding, but the input gets, fully covered by the filter and so the every pixel of the image is valid.

Valid padding is to be used with Max-pooling layers. Here we use every pixel or point value while learning the model as valid pixel. It works on the validation of pixel and not on the size of the input.

(3) **Causal padding :** This type of padding works with one-dimensional convolution layers, we can use them majorly in time series analysis. Since a time series is sequential data, it helps in adding zeros at the start of data, and it helps in predicting the values of previous time steps.

We consider an example of a simplified image to understand the idea of edge detection.

Here, a 6×6 matrix convolved with a 3×3 matrix, output is 4×4 matrix. recall that if $n \times n$ image convolved with $f \times f$ kernel or filter then output image is of size.

$(n - f + 1) \times (n - f + 1)$

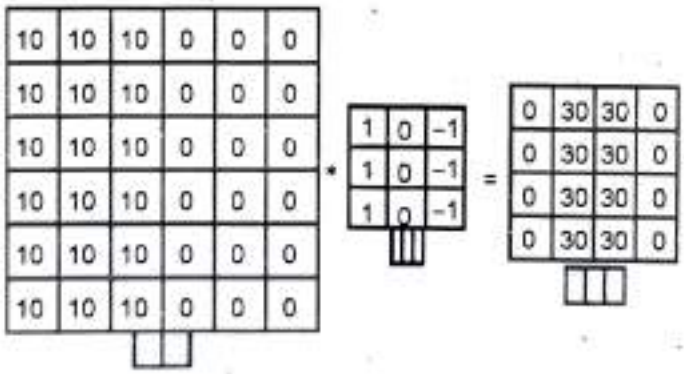


Fig. 4.2.2 : A 6×6 image convolved with 3×3 filter

As we have already seen, there are two problems with convolution :

1. After multiple convolution operation, our original image becomes small which we don't want to happen.
2. Corner features of any image are not used much in the output

Here padding preserves the size of the original image

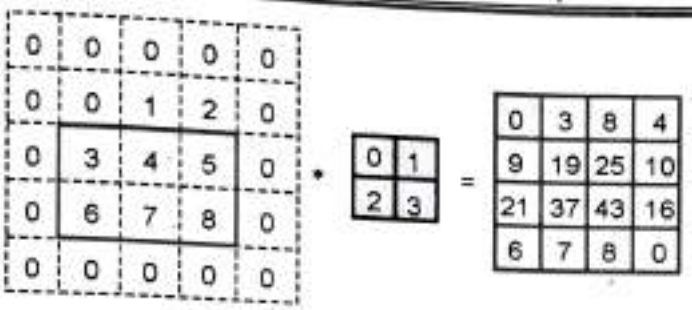


Fig. 4.2.3 : Padded image convolved with 2 * 2 kernel

Hence, if a $(n \times n)$ matrix convolved with an $(f \times f)$ matrix with padding P then the size of the output image becomes.

$(n + 2P - f + 1) \times (n + 2P - f + 1)$; here $P = 1$

Remark

Zero padding is introduced to make the shapes match as required, equally on every side of the input map.

Valid means no padding.

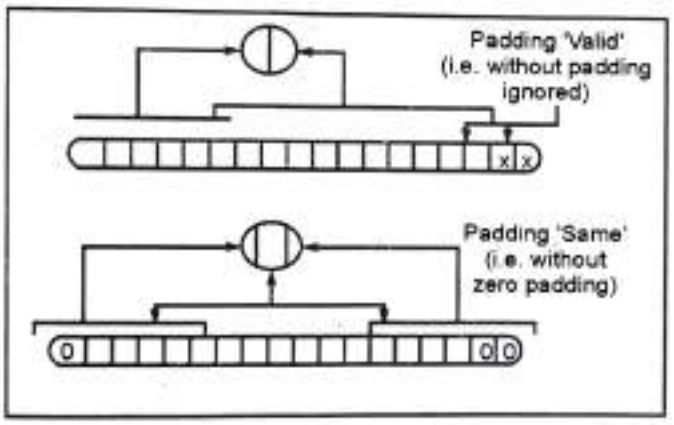


Fig. 4.2.4 : Same versus valid padding with CNN

4.3 STRIDED CONVOLUTION

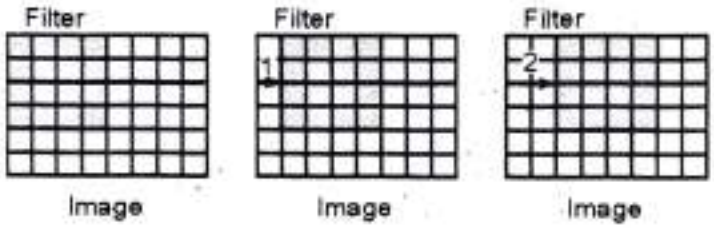
GQ. What is strided convolution ?

Definition : Basically stride means number of pixel over shifts the input matrix.

- A strided convolution is another type of building block of C, that is used in convolutional neural network. Suppose we want to convolve the 7 times 7 image with 3 time 3 filters using a stride of 2. Thus stride is the distance between random location where the convolution kernel is applied.
- The whole concept of stride convolution is that we can stride the window over the input vector, matrix or tensor. The stride parameter indicates the length of step in the stride. In any framework it is always 1.

- Of course we can increase the stride-step length in order to save space or cut calculation time. We may for go some information while doing so. We cannot have a stride of 0, this would mean not sliding at all.
- Applying convolution means sliding a kernel over an input signal outputting a weighted sum where the weights are the values inside the kernel.
- If we use a convolution with a (2×2) stride, the step is 2 in both x and y direction, followed by non-strided convolution, stride 1, step 1.
- We observe that the output of a convolution with stride 2 halves the width and height of the input, whereas the output of a convolution with stride 1 has width = input . width – 2 and height = height – 2, since the kernel is 3×3.

☞ **Stride**



Left image : Stride = 0
Middle image : Stride = 1
Right image : Stride = 2

Fig. 4.3.1 : Convolution

4.3.1 Output Dimension

For padding P, filter size $(f \times f)$ and input image size $(n \times n)$, and stride 's' our output image dimension will be $\left[\frac{(n + 2P - f + 1)}{S} + 1\right] * \left[\left(\frac{n + 2P - f + 1}{S}\right) + 1\right]$

4.4 RELATION BETWEEN INPUT, OUTPUT AND FILTER SIZE

☞ **Rectified linear unit**

GQ. Discuss rectified linear unit and its advantages.

❒ **Definition** : In artificial neural networks, the rectifier activation function is an activation function defined as the positive part of it's argument :

$$f(x) = x^+ = \max(0, x)$$

where x is the input to a neuron. This is also known as a ramp function and is analogous to half wave rectification in electrical engineering.

- A node or unit that implements this activation function is referred to as a rectifier linear activation unit or ReLu.

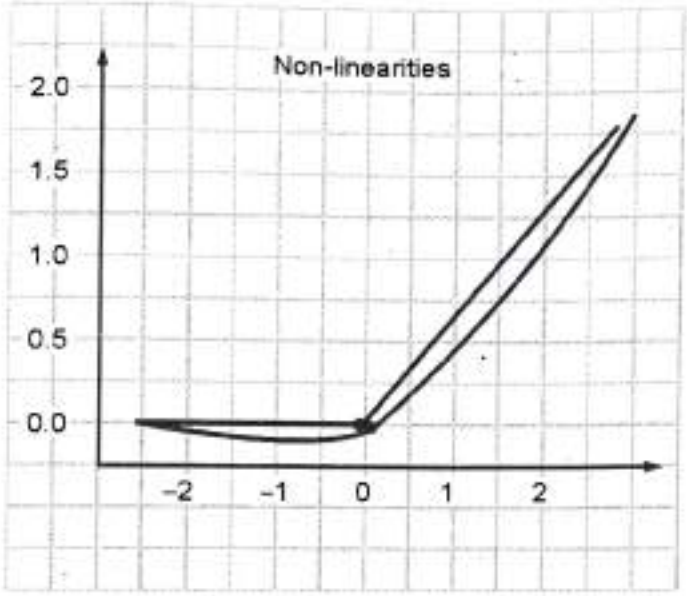


Fig. 4.4.1 : Plot of ReLu rectifier near x = 0 (and GELu) function (Gaussian Error Linear unit)

- In a neural network, the activation function is responsible for transforming the summed weighted input from the node into activation of the node or output for that input.
- The rectified linear artificial function or ReLu is a piecewise linear function that will output the input directly if it is positive, otherwise the output is zero.
- It has become default activation function for many types of neural networks because a model that uses it is easier to train and often achieves better performance.

Here we discuss the rectified linear activation function or ReLu for deep learning neural networks :

(1) The sigmoid and hyperbolic tangent activation functions cannot be used in networks with many layers because of gradient problems and gradient vanishes. But the rectified linear activation function overcomes the vanishing gradient problems allowing models to learn faster and to perform better.

(2) The rectified linear activation function is default activation when developing multilayer perception and convolutional neural networks.

(3) For a given node, the inputs are multiplied by the weights in a node and summed together. This value is referred to as the summed activation of the node. The summed activation is then transformed via an activation function and defines the specific output or "activation" of the node.

(4) The linear activation functions are the simplex activation function. A network consisting of linear activation functions cannot learn complex mapping functions but it is very easy to train. But linear activation functions predict regression problems, hence they are used in output layer for networks.

(5) On the other hand nonlinear activation functions learn more complex structures in data. The sigmoid and hyperbolic tangent activation functions are used as nonlinear activation functions.

(6) The sigmoid activation function is also called as logistic function in neural networks. The input to the function is transformed into a value between 0.0 and 1.0. Inputs which are larger than 1.0 are regarded as 1.0 similarly values smaller than 0.0 are taken as 0.0. The shape of the function is an S-shape from zero up through 0.5 to 1.0.

(7) Hyperbolic tangent function is similar shaped nonlinear activation function, and it's output values lie between – 1.0 and 1.0. This function has better predictive performance and was easier to train. Thus hyperbolic tangent function's performance is better than logistic sigmoid function.

(8) The sigmoid and tanh functions saturate. Tanh function saturates to 1.0 for larger values and sigmoid function saturate to 0 to – 1.0 for small values.

4.4.1 Advantages of Rectified Linear Unit

(1) Here gradient propagation is better : Very few vanishing gradient problems, compared to sigmoidal activation function. Moreover sigmoidal activation function saturate in both direction. In a randomly initialised network, only bout 50% of hidden units have a non-zero output.

(2) Computation is fast and easy. It involves only addition, multiplication and comparison.

(3) It is scale invariant, i.e.

$$\max(0, ax) = a \max(0, x) \text{ for } a \geq 0$$

(4) Rectifying activation functions were trained to separate specific excitation and unspecific inhibition in the neural networks.

(5) The use of rectifier as a non-linearity enables deep supervised training without requiring unsupervised pre-training. Compared to sigmoid function or other similar activation functions, rectified linear units allow effective training of deep neural architectures on complex database.

4.4.2 Disadvantages of Rectified Linear Unit

(1) The function is differentiable except at zero, and the value of the derivative at 0 is arbitrarily chosen as 0 or 1.

(2) It is unbounded

(3) It is not zero-centered.

(4) Rectified linear unit neurons become sometimes inactive for essentially all inputs. In this state, there is no gradients flow backward, through the neuron, and so the neuron remains in an inactive state and 'dies'. This is called as **'Vanishing gradient'** problem.

(5) In some cases, model capacity gests decreased because large number of neurons in a network get struck in dead states.

This problems happens especially when the learning rate is too high. By using leaky ReLus, the performance is reduced.

4.4.3 Variants : Linear Variants

GQ. Explain different types of Variants.

(I) Leaky ReLu

When the unit is not active, leaky ReLus allow a small, positive gradient :

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ 0.01x, & \text{otherwise} \end{cases}$$

(II) Parametric ReLu

This is an advanced variant that Leaky ReLu. Here the coefficient of leakage is made into a parameter and that is learned along with the other neural network parameters.

Here :

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ ax, & \text{otherwise} \end{cases}$$

Note : Note that for $a \leq 1$, this is equivalent to $f(x) = \max(x, ax)$ and has a relation of "maxout" networks.

4.4.4 Non-Linear Variants

GQ. Explain different types of Variants.

(I) Gaussian Error Linear Unit (GELu)

It is a smooth approximation to the rectifier. It has a bump when $x < 0$, and is non-monotonic. It serves as a default, activation for some standard models.

It is defined as : $f(x) = x \cdot \phi(x)$

Where $\phi(x)$ is the cumulative distribution function of the standard normal distribution.

(II) Softplus

A smooth approximation to the rectifiers is the analytic function :

$f(x) = \log_e (1 + e^x)$, and it is called as **'Softplus'** or **'SmoothReLu' function**

If we include sharpness parameters 'K'

$$\text{then} \quad f(x) = \frac{\log(1 + e^{Kx})}{K}$$

$$\text{and } f'(x) = \frac{1}{K}\left[\frac{1}{(1 + e^{Kx})} \cdot Ke^{Kx}\right] = \frac{e^{Kx}}{1 + e^{Kx}}$$

Dividing N and D by e^{Kx}

$$f'(x) = \frac{1}{1 + e^{-Kx}}$$

Thus the logistic sigmoid function is a smooth approximation of the derivative of the rectifier.

The multivariable generalisation of single-variable softplus is the 'Log Sum Exp' function with the first argument kept to zero.

$$LSE_0^+(x_1, x_2, \ldots, x_n) = LSE(0, x_1, \ldots, x_n)$$

$$= \log(1 + e^{x1} + e^{x2} + \ldots + e^{xn})$$

The 'Log Sum Exp' function is

$$LSE(x_1, x_2, \ldots, x_n) = \log(e^{x1} + e^{x2} + \ldots + e^{xn})$$

It's gradient is the softmax

(III) ELU

ELU is exponential linear unit. Using this mean activations are made closer to zero. And it speeds up learning. And we can have higher classification accuracy than ReLus.

It is defined as : $$f(x) = \begin{cases} x, & \text{if } x > 0 \\ a(e^x - 1), & \text{otherwise} \end{cases}$$

Where $a \geq 0$ and is a hyper-parameter.

The ELU is a smoothed version of ReLu, which has the form

$f(x) = \max(-a, x)$, where $a \geq 0$.

4.4.5 One Layer of a Convolutional Network

- Convolution with first filter gives one 4/times 4 images output and convolving with the second filter gives a different 4×4 output.
- To convert this into a convolutional neural network layer, we add a scalar quantity, 'bias'. Bias is added to every element in 4×4 output, or to all these 16 elements. Then we apply activation function ReLu.
- The same we will do with the output we got by applying the second $3 \times 3 \times 3$ filter (kernel). Again we add a **different bias** and apply ReLu activation function. After

applying bias or after applying ReLu artificial function dimensions of outputs remain the same, so we have two 4×4 matrices. And then we repeat the steps and we get one layer of a convolutional neural network.

4.4.6 Fully Connected Layers

- Fully connected layers multiply the input by a weight matrix and then adds a 'bias' vector.
- The convolutional layers are followed by one or more fully connected layers. Hence all neurons in a fully connected layers connect to all neurons in the previous layer.

4.4.7 Regularisation

To prevent overfitting in the training phase, there are different ways of controlling training of CNNs. In particular they are L_2/L_1 regularisation and max-norm constraints :

(1) **L_2 regularisation :** This regularisation is implemented by penalising directly the squared magnitude of all parameters in the objectives. Using the gradient descent parameter, every weight is decayed linearly towards zero by L_2 regularisation method.

(2) **L_1 regularisation :** Here in this method, we add the term $\lambda|\omega|$ for each weight ω to the objective.

It is also possible to combine the L_1 regularisation with L_2 regularisation $\lambda_1 |\omega_1| + \lambda_2 |\omega_2|$ and is known as Elastic-net regularisation.

(3) **Max-Norm constraint :** Here we enforce an absolute upper bound on the magnitude of the weight vector for every neuron and use projected gradient descent to enforce the constraint. This is altogether a different form of regularisation.

4.5 CNN ARCHITECTURE : CONVOLUTION LAYER, POOLING LAYER

Convolution neural network (CNN) architecture

GQ. Explain CNN architecture.

- A **Convolution Neural Network** (CNN) is a feed-forward neural network (FNN). So far CNNs have established extraordinary performance in image search services, voice recognition and natural language processing (NLP).
- We have already seen that a regular multilayer perceptron gives good result for small images. But it breaks down for larger images.
- For example, if the first layer has 1000 neurons, then there will be 10 million connections.

- CNNs solve this problem using partially connected layers. CNN has fewer parameters than a deep neural network (DNN), hence it requires less training data.
- In addition, CNN can detect any particular feature anywhere on the image, but a DNN can detect it only in that particular location.
- Since images have generally repetitive features, CNNs can generalise much better than DNNs for image processing tasks such as classification.
- A CNNs architecture has prior knowledge of how pixels are organised.
- Lower layers identify features in small areas of the images, while higher layers combine features of lower-layer into larger features. In case of DNNs, this doesn't work. Thus, in short, CNN is a class of neural networks that specialises in processing data that has a grid-like topology, such as an image.
- Each neuron works in it's own receptive field and is connected to other neurons in a way that they cover entire visual field.
- A CNN design begins with feature extraction and finishes with classification.

☞ **DNN Neural Network**

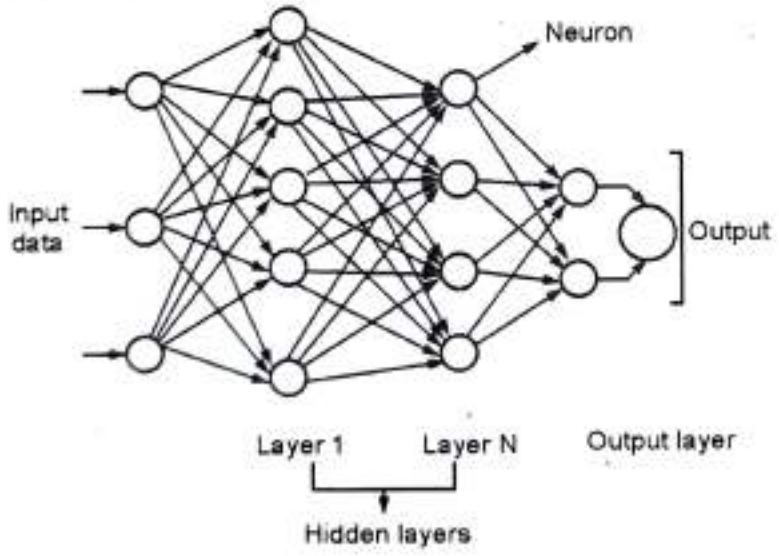


Fig. 4.5.1 : DNN Neural Network

In Fig. 4.5.1, there is DNN neural network. On the right, a convolution net arranges it's neurons in 3-dimensions, as seen in one of the layers.

❒ **Definition :** Convolution is a mathematical operation. In 'convolution neural network' convolution is employed in the network.

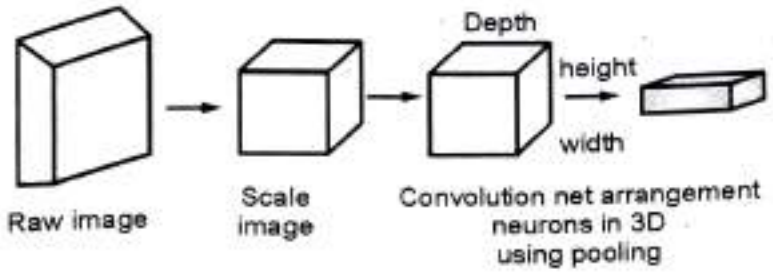


Fig. 4.5.2 : 3D-pooling

4.5.1 Architecture

- A conventional neural network consists of an input layer, hidden layers and output layer. (as shown in the figure). The middle layers are called as hidden layers because their inputs and outputs are governed by activation function and convolution.
- In CNN, the input is a tensor with a shape :

  (Number of inputs) × (input height) × (input width) × (input channels).
- After passing through a convolutional layer, the image becomes a features map, also called an activation map, with shape :

(number of inputs) × (feature map height) × (feature map width) × (feature map channels).

- The input is convolved in convolutional layers and the result is forwarded to the next layer. Each convolutional neuron processes data only for it's receptive field.
- Fully connected feed forward neural networks architecture is impractical for larger inputs. It requires a very high number of neurons. For example, a fully connected layer of size 100 × 100 has 10,000 weights for each neuron in the second layer.
- Instead using convolution a 5 × 5 filing regions, only 25 learnable parameters are required. Also convolutional neural networks are ideal for data with a grid-like topology since separate features are taken into account during convolution.

4.5.2 Functions of Hidden Layers

GQ. Explain function of Hidden layers.

1. The first hidden layer performs convolution. Each feature map in the hidden layer consists of neurons and each neuron is assigned a receptive field.
2. The second hidden layer performs averaging and subsampling. Each layer consists of feature maps and the neurons of each feature map has a receptive field. It has a trainable bias, trainable coefficient and sigmoid function. They control the operating point of the neuron.
3. The next hidden layer performs a second convolution. Again each feature map in this hidden layer consists of neurons. And each neuron has connections with the previous hidden layer.

4. The next hidden layer performs averaging and subsampling.
5. The output layer performs final stage of convolution. Each neuron is assigned a receptive field and is assigned the possible characters.

The layers in the network alternate between convolution and subsampling, and we get a "bipyramidal" effect. Thus at each layer i.e. either subsampling or convolutional layer, the number of feature maps is increased while the space-resolution is reduced. The weight sharing reduces the number of free parameters in the network compared to the synaptic connections in multilayer perceptron. Also by the use of weight sharing, implementation of convolutional network in parallel form is possible.

4.5.3 Design of Multilayer Perceptron

GQ. Explain in detail Multilayer Perceptron.

- Using convolutional network, a multilayer perceptron of manageable size can learn a complex, high-dimensional, nonlinear mapping.
- Through the training set synaptic weights and bias levels can be learned by simple back propagation algorithm.
- Back propagation algorithm is the key-stone algorithm for the multilayer perceptrons. The partial derivatives of the cost function with respect to the free parameters of the network are determined by back-propagating of the error signals of the output neurons, hence the algorithm is called as Back-Propagation Algorithm.
- Updating the synaptic weights and biases of multilayer perceptron and deriving all partial derivatives of the cost function with respect to free parameters are the base factors for evaluating the power of the algorithm.

Details involved in the **design of a multilayer perceptron** are as follows :

(1) All the neurons in the network are **nonlinear**. And nonlinearity is achieved by using a sigmoid function, and they are

(i) The nonsymmetric logistic function, and

(ii) The antisymmetric hyperbolic tangent function.

(2) Each neuron has it's own hyperplane in decision space, and the combination of hyperplanes formed by all the neurons in the network is interatively adjusted by a supervised learning process. And this patterns from different classes are separated with the few classification errors.

(3) For the pattern classification, the random back-propagation algorithm is used to perform the training.

(4) In nonlinear regression, the output range of the multilayer perceptron should be sufficiently larger.

4.5.4 Performance Measure

(i) Algorithm is based on **minimising the cost function**,

(ii) But minimising the cost function may lead to optimising the intermediate quantity, and this may not be the aim of the system. Hence 'reward-to-volatility' ratio as a performance measure of risk-adjusted return is more appreciable than Eav.

4.5.5 Input Layer

The input layer passes the data directly to the first hidden layer. Here the data is multiplied by the first hidden layers weights. Then the input layer passes the data through the activation function then it passes on.

4.6 POOLING LAYERS

GQ. Explain Pooling layers and its different types.

(1) Pooling layers are used to reduce the dimensions of the feature maps. It reduces the number of parameters to learn and the amount of computation performed in the network.

(2) The pooling layer summarises the features present in a region of the feature map generated by a convolutional layer.

(3) A pooling layers is another building block of a CNN, when processing multichannel input-data, the pooling layer pools each input channel separately.

(4) Pooling layer reduce the dimensions of the data by combining the output of neuron-clusters. The pooling layer is used to reduce spatial dimensions, but not depth, on a convolutional neural network.

(5) Pooling layer operates on each feature map independently.

(6) Pooling is basically 'downscaling' the image obtained from the previous layers. It can be compared to shrinking an image to reduce the pixel density.

4.6.1 Average Pooling

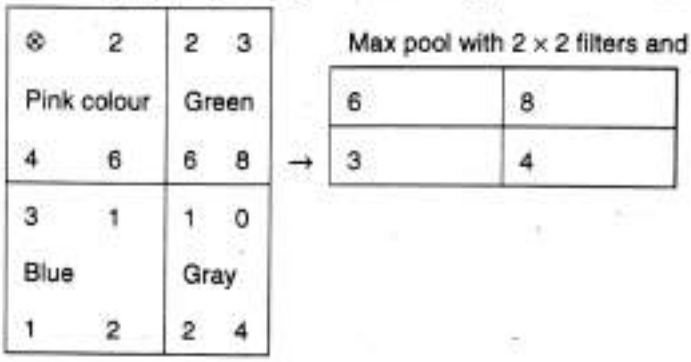
There are two types of widely used pooling in CNN layer.

1. Max pooling 2. Average Pooling

1. Max-pooling

- The popular kind of pooling is **Max-pooling**. Suppose we want to pool by a ratio of 2. It implies that the height and width of your image will be half of it's original value. So we have to compress every 4 pixels (a 2×2 grid) and map it to a new single pixel with **loosing** the **important** data from the **missing** pixels.

- Max pooling is done by taking the largest value of these 4 pixels. Thus one new pixel represents 4 old pixels by using the largest value of these 4 pixels. This is repeated for every group of 4 pixels throughout the whole image.



Here the largest pixels are 6,8,3 and 4.

- Here the quality of an image is reduced to avoid computational load on the system. By reducing the quality of the image, we can increase the 'depth' of the layer, to have more features in the reduced image.
- Reducing the image size helps the convolution layer after the pooling layer to look for 'higher level features'. It means that the convolution layer looks at the picture as a whole.

2. Average pooling

- **Average pooling** is different from **Max Pooling**. Average pooling retains 'less important' information about the elements of a block, or pool.
- But Max pooling chooses the maximum value and discards less important values (as shown in the Fig. 4.6.1).
- But 'less important' is also sometimes useful in a variety of situations.

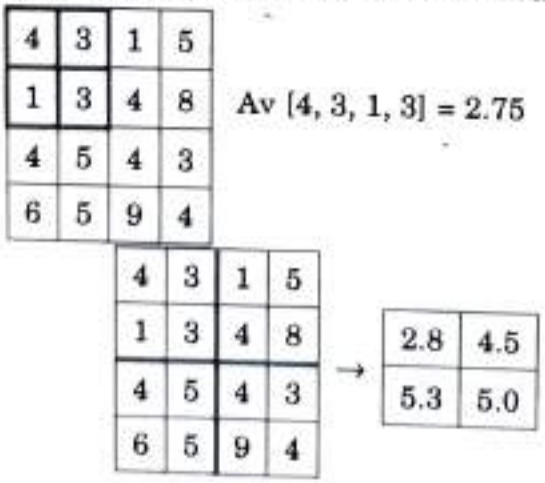


Fig. 4.6.1 : Max pooling

4.6.2 Padding

GQ. Discuss padding or Write short note on Padding.

- In Convolutional Neural Networks (CNN), padding is a term that refers to the a number of pixels added to an image when it is being processed by the kernel of a CNN.
- For example, if padding in CNN is kept zero, then every pixel value that is added will be of value zero.
- Padding is simply a process of adding layers of zeros to our input images so as to avoid the problems mentioned above.
- CNNs are used extensively for tackling problems occurring in image processing and predictive modelling or classification tasks. The main application of CNN is to analyse image data.
- Now, every image in any dataset is a matrix of it's pixel values. When working with simple CNN for an image, we get output reduced in size and that is loss of data. And that hinders to obtain a proper result according to our requirements.
- When we do not want the shape or our outputs to reduce in size, the addition of more layers in the data can help to obtain a proper result and that addition can be done by padding.
- Now we study padding with it's importance and how to use it with CNN models. We also see different methods of padding and how they can be implemented.

4.6.3 Problem with Simple Convolution Layer

- A simple CNN of size $(n \times n)$ with $(f \times f)$ filter/kernel size gives result for output image as $(n - f + 1) \times (n - f + 1)$
- For example in any convolution operation with a (8×8) image and (3×3) filter the output image size will be (6×6). Thus the output of the layers is shrunk in comparison to the input. Again, the filters we are using may not focus on the corners every time when it moves on the pixels e.g.

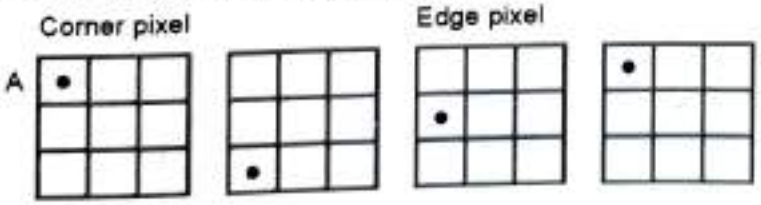


Fig. 4.6.2

- The above image is an example of the movement of a filter of size (3×3) on an image of size (6×6), corner pixel A is coming under the filter in only one movement. This shown that pixel A is misinterpreted.

- This causes loss of information available in the corners and also the output from the layers is reduced and this reduced information may create confusion for next layer. This problem of model can be solved by padding layer.
- The convolution layers reduce the size of the output. So when we want to save the information presented in the corners, we can use padding layers where padding helps by adding extra rows and columns on the outer dimension of the images. So the size of the **input data** will remain similar to the output data.
- Padding basically extends the area of an image in which convolutional neural network processes. The kernel/filter which moves across the image scans each pixels and converts the image into a smaller image.
- Padding is added to the outer frame of the image to allow for more space for the filter to cover in the image. This creates a more accurate analysis of images.

4.6.4 Types of Padding

GQ. What are types of Padding.

We come across three types of padding :

(1) Same padding
(2) Valid Padding
(3) Causal padding

(1) **Same padding :** In this type of padding, the padding layers have zero values in the outer frame of the images or data so the filter we are using can cover the edge of the matrix and it carries the required.inference.

(2) **Valid padding :** This type of padding is called as no padding. Here we don't apply any padding, but the input gets, fully covered by the filter and so the every pixel of the image is valid.

Valid padding is to be used with Max-pooling layers. Here we use every pixel or point value while learning the model as valid pixel. It works on the validation of pixel and not on the size of the input.

(3) **Causal padding :** This type of padding works with one-dimensional convolution layers, we can use them majorly in time series analysis. Since a time series is sequential data, it helps in adding zeros at the start of data, and it helps in predicting the values of previous time steps.

We consider an example of a simplified image to understand the idea of edge detection.

Here, a 6×6 matrix convolved with a 3×3 matrix, output is 4×4 matrix. recall that if $n \times n$ image convolved with $f \times f$ kernel or filter then output image is of size.

$(n - f + 1) \times (n - f + 1)$

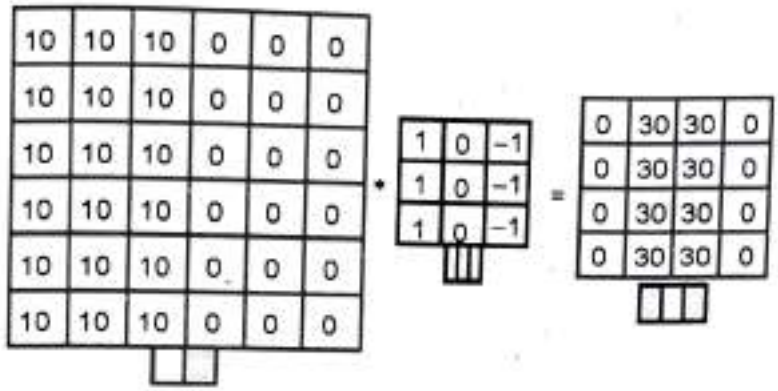


Fig. 4.6.3 : A 6×6 image convolved with 3×3 filter

As we have already seen, there are two problems with convolution :

1. After multiple convolution operation, our original image becomes small which we don't want to happen.
2. Corner features of any image are not used much in the output

Here padding preserves the size of the original image :

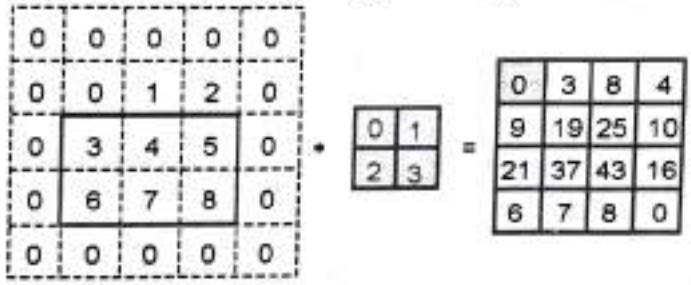


Fig. 4.6.4 : Padded image convolved with 2 * 2 kernel

Hence, if a $(n \times n)$ matrix convolved with an $(f \times f)$ matrix with padding P then the size of the output image becomes.

$(n + 2P - f + 1) \times (n + 2P - f + 1)$; here $P = 1$

Remark

Zero padding is introduced to make the shapes match as required, equally on every side of the input map.

Valid means no padding.

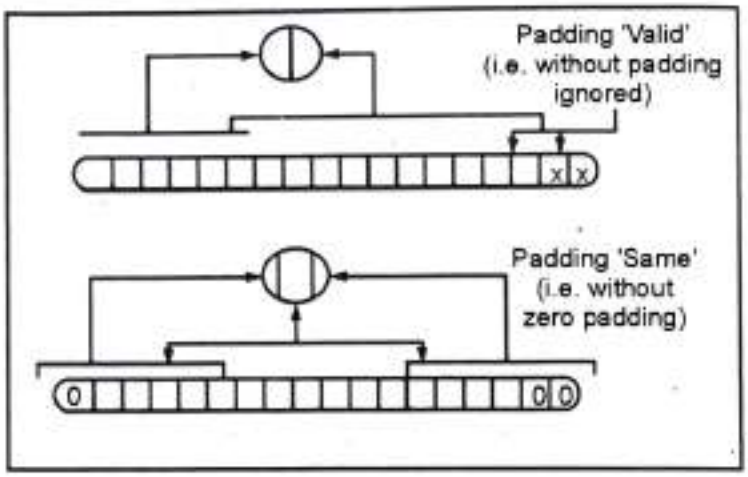
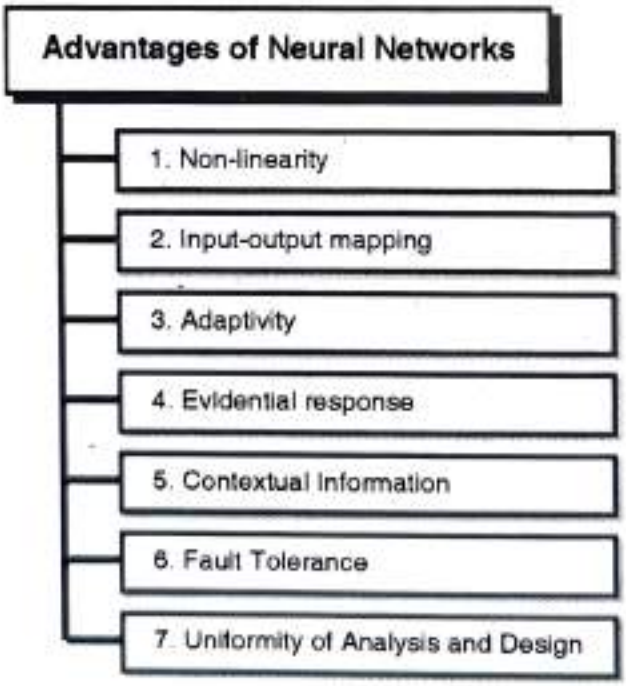


Fig. 4.6.5 : Same versus valid padding with CNN

4.7 FULLY CONNECTED NN V/S CNN

Advantages of Neural Networks

The use of neural networks offers the following useful capabilities :



(1) **Non-linearity :** An artificial neuron can be **linear or non-linear**. A neural network consists of an interconnection of non-linear neurons, and hence it is itself non-linear. The non-linearity is distributed throughout the network.

(2) **Input-output mapping :** Each example consists of a unique **input signal** and corresponding desired response. Thus the network learns from the examples by constructing an **input-output mapping** for the problem at hand.

(3) **Adaptivity :** A neural network is trained to operate in a specific environment can be easily retrained to deal with minor changes in the operating environmental conditions.

(4) **Evidential response :** A neural network can be designed to provide information not only about which particular pattern to **select,** but also about the **confidence** in the decision made.

(5) **Contextual Information :** Knowledge is represented by the very structure of a neural network. Every neuron in the network is potentially affected by the global activity of all other neurons in the network.

(6) **Fault Tolerance :** A neural network has the potential to be **fault tolerant.** In order to have neural network **fault tolerant,** it is necessary to take corrective measures in designing the algorithm used to develop the network.

(7) **Uniformity of Analysis and Design :** Neural networks are basically information processors. Hence the same notation is used in all domains involving the application of neural networks.

4.8 VARIANTS OF THE BASIC CONVOLUTION FUNCTION

First we define a few parameters that define a convolutional layer.

(1) **Kernel Size :** The kernel size defines the field of view of the convolution. A common choice for 2 D is 3 – that is 3 × 3 pixels.

(2) **Stride :** The stride defines the step size of the kernel when traversing the image. While its default is usually 1, we can use a stride of 2 for down sampling an image similar to Man Pooling.

(3) **Padding :** The padding defines how the border of a sample is handled. A (half) padded convolution will keep the spatial output dimensions equal to the input, but the unpadded convolution will crop away of the borders if the kernel is larger than 1.

(4) **Input and output channels :** A convolutional layer takes a certain number of input channels (I) and calculates a specific number of output channels (0). The needed parameters for such a layer can be calculated by I. O. K, where K is equal to number of values in the kernel.

4.8.1 Dilated Convolutions

- Dilated convolutions introduce another parameter to convolutions introduce another parameter to convolutional layers called the **dilation rate**. This gives a spacing between the values in a kernel.
- A 3 × 3 kernel with a dilation rate of 2 will have the same field of view as a 5 × 5 kernel, while using only 9 parameters.

- This delivers a wider field of view at the same computational cost.
- Dilated convolutions are particularly popular in the field of real-time segmentation.

4.8.2 Transposed Convolutions

- We note that transposed convolution is **not a deconvolution**.
- Deconvolutions do exist but they are not common in the field of deep learning. A transposed convolution is somewhat similar because it produces the same spatial resolution a hypothetical deconvolutional layer will. But the actual mathematical operation that is performed on the values is different.
- A transposed convolutional layer carries out a regular convolution but reverts its spatial transformation. We consider an **example** :

  An image of 5×5 is fed into a convolutional layer.

  The stride is set to 2, the padding is deactivated and the kernel is 3×3. This results in a 2×2 image.
- To reverse the process, a transposed convolution produces an output of 5×5 image. It performs a normal convolution operation.
- It reconstructs the spatial resolution from before and performs a convolution. (It is actually not a mathematical inverse, but for Encoder-decoder architecture, it is still very useful). This way we can combine the up scaling of an image with a convolution.

4.8.3 Separable Convolutions

- In a separable convolution, we can split the kernel operation in to multiple steps.
- We express a convolution as y = convolutional (x, k) where y is the output image, x is the input image and k is the kernel.
- Let us assume that k can be calculated as $k = k_1 \cdot \text{dot}(k_2)$.
- This makes it a separable convolution because instead of doing a 2D convolution with k, we could get the same result by doing 2 1 D convolution with k_1 and k_2.

  In image processing we use sobel kernel

SOBEL X and Y filters

−1	0	+1
−2	0	+2
−1	0	+1

X filter

+1	+2	+1
0	0	0
−1	−2	−1

Y filter

We can get the same kernel by multiplying the vector [1, 0, – 1] and [1, 2, 1] T. This will require 6 instead of a parameters while doing the same operation.

4.9 2D-CONVOLUTION

- Convolution involving one-dimensional signals is referred to as 1D convolution or just convolution. But, if the convolution is performed between two signals spanning along two mutually perpendicular dimensions (i.e., if signals are two-dimensional in nature), then it will be referred to as 2D-convolution.
- The same concept can be extended to involve multi-dimensional signals and hence we can have multi-dimensional convolution.
- Similar to the one-dimensional case, an asterisk is used to represent the convolution operation. In case of 2-dimensional convolution, it would DC written with 2 asterisk.
- The following represents as 2-dimensional convolution of discrete signals. :

$y(n_1, n_2) = x(n_1, n_2) * ... * h(n_1, n_2)$.

For discrete-valued signals, the convolution can be directly computed as :

$$\sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} h(k_1, k_2)\, x(n_1 - k_1, n_2 - k_2)$$

- We list several properties of the two-dimensional convolution operator. These can be extended to signals of N-dimensions.

(i) Commutative Property

$$x ** h = h ** x$$

(ii) Associative property

$$(x ** h) ** g = x ** (h ** g)$$

(iii) Distributive property

$$x ** (h + g) = (x ** h) + (x ** g)$$

- Given some input $x(n_1, n_2)$ that goes into a filter with impulse response $h(n_1, n_2)$ and then another filter with impulse response $g(n_1, n_2)$, the output is $y(n_1, n_2)$.

Let the output given by first filter be $w(n_1, n_2)$; this implies that :

$$w = x ** h$$

- Then this function is convolved with the impulse response of the second filter, and the output can be written as

$$y = w ** g = (x ** h) ** g$$

- Using associative property, this becomes

$$y = x ** (h ** g).$$

☞ **Pictorially**

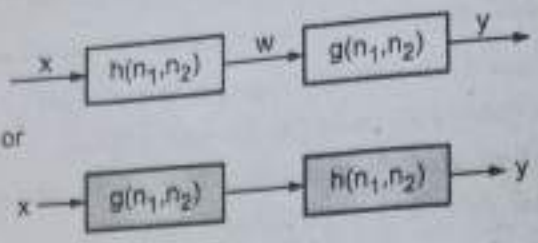


Fig. 4.9.1

☞ **Applications of 2D-convolution**

2D-convolution filtering is a technique that can be used for an immediate array of image processing objective some of which include that as (i) image sharpening, (ii) image smoothing, (iii) edge detection, and (iv) texture analysis.

☞ **Difference between 1D and 2D**

For 1D, measurements data are only collected under a single point at the surface,

For 2D, a profile is measured.

The 1D-array consists of a list of variables that have the very same data type.

A 2D-array consists of a list of arrays that have similar data types.

☞ **Types of 2D**

- The basic types of 2d-shapes are a circle, triangle, square, rectangle, pentagon, quadrilateral, hexagon, octagon etc.
- Apart from the circle, all the shapes are considered as polygon, which have sides. A polygon which has all the sides and angles as equal is called a regular polygon.

⏭ 4.10 LeNeT : ARCHITECTURE

☞ **Introduction**

- LeNet-5 is one of the earliest pre-trained models proposed by Yann LeCun and Others in year 1998, in the research paper : Gradient - Based Learning Applied to Document Recognition.
- In general, LeNet refers to LeNet-5 and is a [simple convolutional neural network] [Convolutional neural networks are a kind of feed-forward neural network] whose artificial neurons can respond to a part of the surrounding cells in the coverage range and perform well in large-scale image processing.
- LeNet is a very efficient convolutional neural network for handwritten character

- The architecture is straight forward.
- It consists of two sets of convolutional and average pooling layers, followed by a flattening convolutional layer, then two fully connected layers and finally a softmax classifier.

First Layer

- The input for LeNet-5 is a 32 × 32 gray scale image which passes through the first convolution layer with 6 features maps or filters having size 5 × 5 and a stride of one.
- The image dimensions change from 32 × 32 × 1 to 28 × 28 × 6,

Second Layer

- Then the LeNet-5 applies average pooling layer or sub-sampling layer with a filter size 2 × 2 and a stride of two.
- The resulting image dimensions will be reduced to 14 × 14 × 6.

Third Layer

- Now, there is a second convolutional layer with 16 feature maps having size 5 × 5 and a stride of 1. In this layer, only 10 out of 16 feature maps are connected to 6 feature maps of the previous layer.
- The main reason is to break the symmetry in the network and keeps the number of connections within reasonable limits.

Fourth Layer

The fourth layer (S4) is again an average pooling layer with filter size 2 × 2 and a stride of 2. This layer is the same as the second layer (S2) except that it has 16 feature maps so the output will be reduced to 5 × 5 × 16.

Fifth layer

- The fifth layer (C5) is a fully connected convolutional layer with 120 feature maps each of size 1 × 1.
- Each of 120 units in C5 is connected to all the 400 nodes (5 × 5 × 16) in the fourth layer S4.

☞ C5 : Fully connected layer

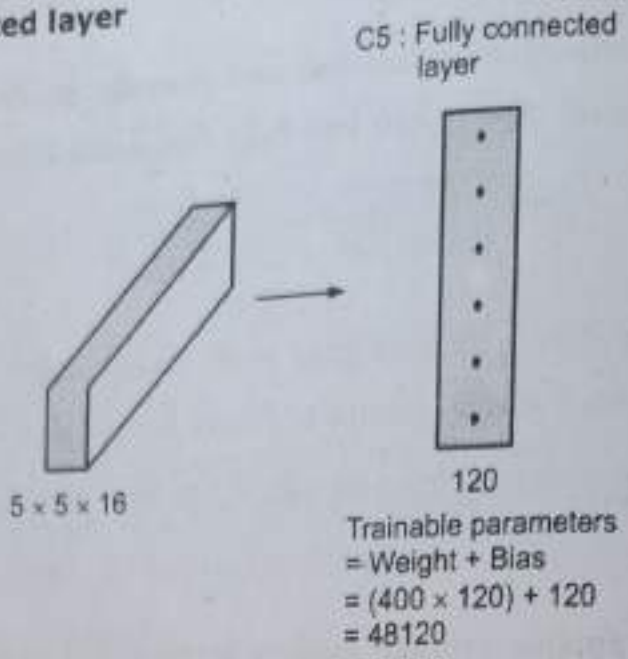


Fig. 4.10.1

☞ Sixth Layer

Sixth layer is a fully connected layer (F6) with 84 units.

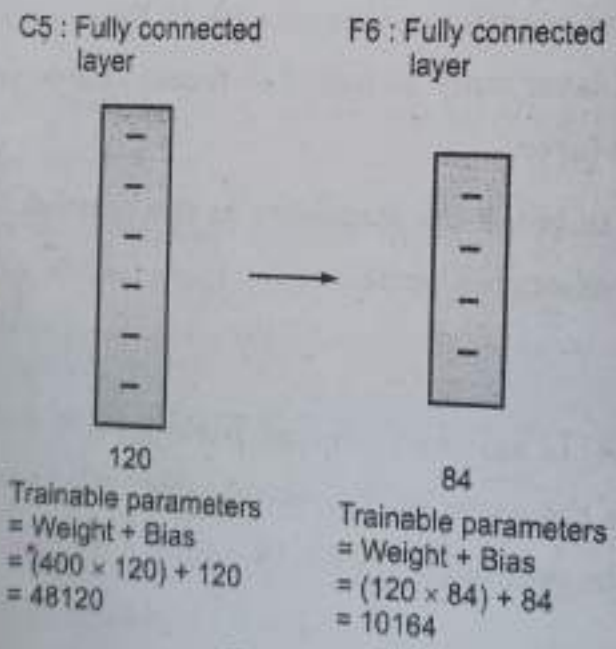


Fig. 4.10.2

☞ Output layer

Now, there is a fully connected softmax output $\hat{y}$ with 10 possible values corresponding to the digits form 0 to 9.

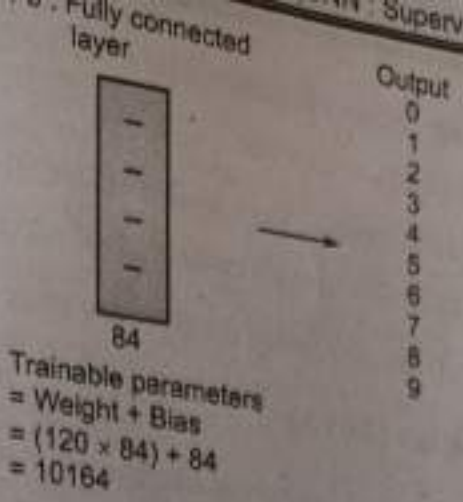


Fig. 4.10.3

Advantage of Le-Net 5

The significant advantage of Le Net-5 is

(i) That it contains of a convolutional layer with stride two, and an-average pooling layer with stride 1.

(ii) It is designed to work on a fixed-size input.

Disadvantages Le-Net 5

(i) LeNet-5 is not applied widely ; instead AlexNet and VGG – 16 Net are used as they consist of deeper layers resulting in accurate classifications.

(ii) LeNet was not designed to work on large images.

Just like AlexNet, LetNet is designed to work on a fixed – size input.

4.11 ALEX NET : ARCHITECTURE

Popular CNN architecture – alex net

Before we proceed with popular CNN Architecture, we define some terminology :

(i) A **wider** network means more feature maps (filters) in the convolutional layers.

(ii) A **deeper** network means more convolutional layers.

(iii) A network with **higher resolution** means that it processes input images with larger width and depth (spatial resolutions). That way the produced feature maps will have higher spatial dimensions.

Alex Net

- Alex net is made up of 5 convolutional layers starting from an (11 × 11) kernel.
- It was the first architecture that employed **max-pooling** layers, ReLu activation functions, and dropout for 3 enormous linear layers.

- The network was used for image classification with 1000 possible classes; Now we can implement it in 35 lines of pyTorch code.
- It was the first convolutional model that was successfully trained on **Imagenet** and at that time, it was more difficult to implement such a model in CUDA.
- To avoid over fitting, dropout is heavily used in the enormous linear transformations.

☞ **Inceptionnet / GoogleNet (2014)**

- Increasing the depth (number of layers) is not the only way to make a model bigger.
- Increasing **both** the depth and width of the network while keeping computations to a constant level, is the main problem.
- To achieve this, information is processed at multiple scales and then aggregated locally.
- Now, aim is to achieve this without a memory explosion.
- The answer is with |X|convolutions. The main aim is dimension reduction, by reducing the output channels of each convolution block. Then we can process the input with different kernel sizes. As long as the output is padded, it is the same as in the input.
- To find the appropriate padding with single stride convs without dilation, padding P and kernel K are defined so that

  Out = in (input and output spatial dims) :

  Out = $in + 2 \cdot P - K + 1$, which means that

$$P = \left(\frac{k-1}{2}\right)$$

- Now we need the |X| convolutional layer to 'project' the features to fewer channels in order to win computational power. And with these extra resources, we can add more layers.
- Actually, the 1×1 convs work similar to a low dimensional embedding. This increases not only depth, but also the width of the famous GoogleNet by using inception modules.

- The core building block, called the inception module is as follows :

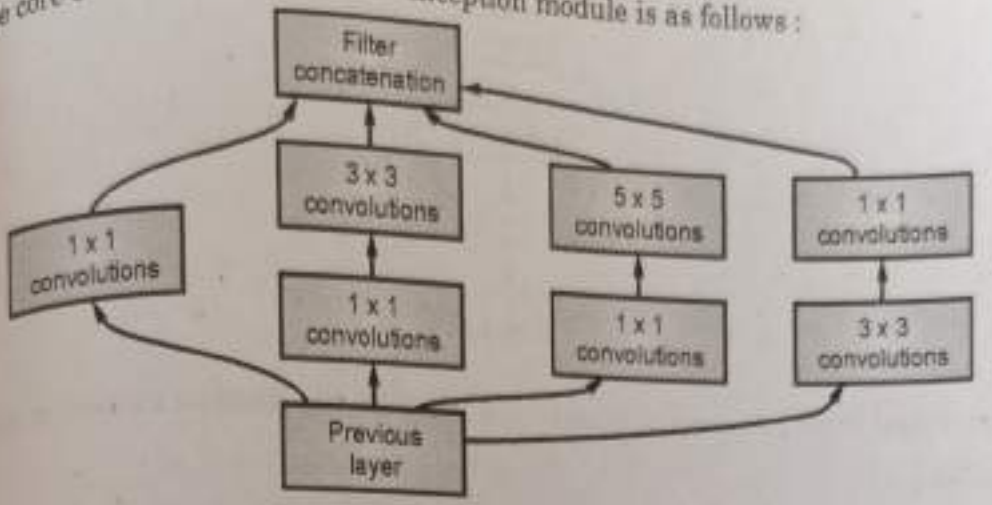


Fig. 4.11.1 : Inception module

- The whole architecture is called GoogleNet or InceptionNet. In essence, they try to approximate a sparse convent with normal dense layers.
- Note that only a small number of neurons are effective. They satisfy the Hebbian Principle
- "Neurons that fire together, wire together".
- In general, a larger kernel is preferred for information that resides globally, and a smaller kernel is preferred for information that is distributed locally.
- Besides, 1×1 convolution are used to compute reductions before the computationally expensive convolutions (3×3 and 5×5).
- The InceptionNet / GoogLeNet architecture consists of 9 inception modules stacked together, with max-pooling layers between (to halve the spatial dimensions).
- It consists of 22 layers (27 with the pooling layers). It uses global average pooling after the last inception module.

4.12 ResNeT ARCHITECTURE

- After the first CNN-based architecture (AlexNet), every subsequent architecture uses more layers in a deep neural network to reduce the error rate. This works for less number of layers, but when the number of layers increases, it causes the gradient to become 0 or too large. Thus when we increase number of layers, the training and test error rate also increases.
- A residual neural network skips connections that perform identity mapping, and merges with the layer outputs by addition. The identity skip connections are referred to as "residual connections", and are also used in the LSTM networks.

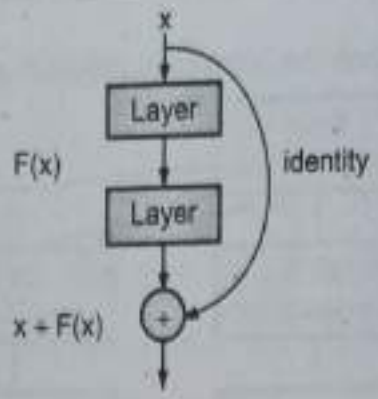


Fig. 4.12.1

- A residual Block in a deep Residual Network. Here the Residual connection skips two layers.

☞ **Signal propagation**

The introduction of identity mappings facilitates signal propagation in both forward and backward paths.

☞ **Forward propagation**

If the output of the l^{th} residual block is the input to the $(l + 1)^{th}$ residual block, with no activation function between blocks, we have.

$$x_{t+1} = F(x_t) + x_t$$

Using recursively,

$$x_{t+2} = F(x_{t+1}) + x_{t+1}$$

$$= F(x_{t+1}) + F(x_t) + x_t$$

Thus, we have,

$$x_L = x_l + \sum_{i=l}^{L-1} F(x_i)$$

Where L is the index of any later residual block (e.g. the last block) and l stands for any earlier block.

Thus the formulation says that there is always a signal that is directly sent from a shallower block l to a deeper block L.

Variants of Residual Blocks

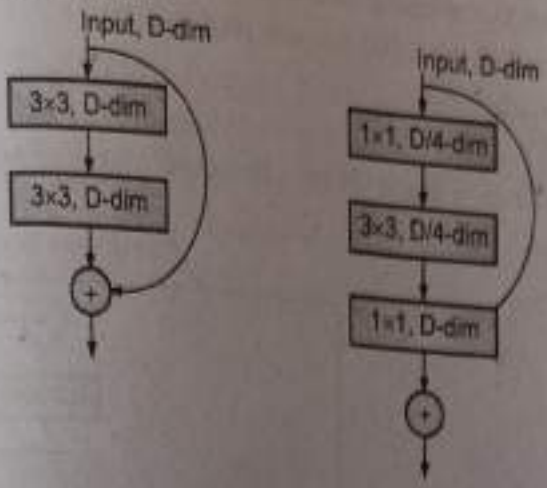


Fig. 4.12.2

Two variants of convolutional Residual Blocks.

Left : A Basic block that has two 3×3 convolutional layers.

Right : A Bottleneck Block that has a 1×1 convolutional layer for dimension reduction to $\frac{1}{4}$, a 3×3 convolutional layer, and another 1×1 convolutional layer for dimension restoration.

Basic Block

- A basic block is the simplest building block in the original ResNet. This block consists of two sequential 3×3 convolutional layers and a residual connection.
- The input and output dimensions of both layers are equal.

Bottleneck Block

- A bottleneck block consists of three sequential convolutional layers and a residual connection.
- The first layer in this block is a 1×1 convolution for dimension reduction, e. g. $\frac{1}{4}$ of the input dimension, the second layer performs a 3×3 convolution, the last layer is another 1×1 convolution for dimension restoration.

Pre-activation Block

- The pre-activation Residual block applies the activation functions, e.g. non-linearity and normalisation, before applying the residual function F.
- The computation of a pre-activation Residual Block can be written as ;

$$x_{t+1} = F(\phi(x_t)) + x_t$$

Where ϕ can be any non-linearity activation, (e. g. ReLu), operation.

This design reduces the number of non-identity mappings between Residual Blocks. This design can train models with 200 to over 1000 layers.

Transformer Block

- A Transformer block is a stack of two Residual blocks. Each Residual Block has a Residual Connection.

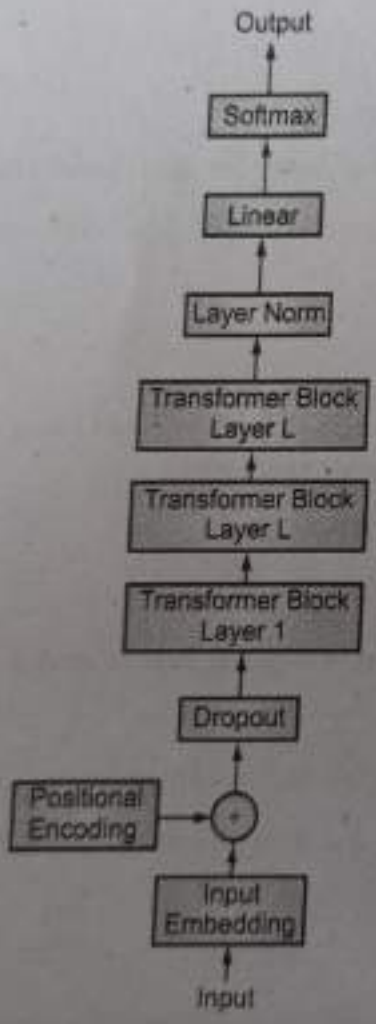


Fig. 4.12.3

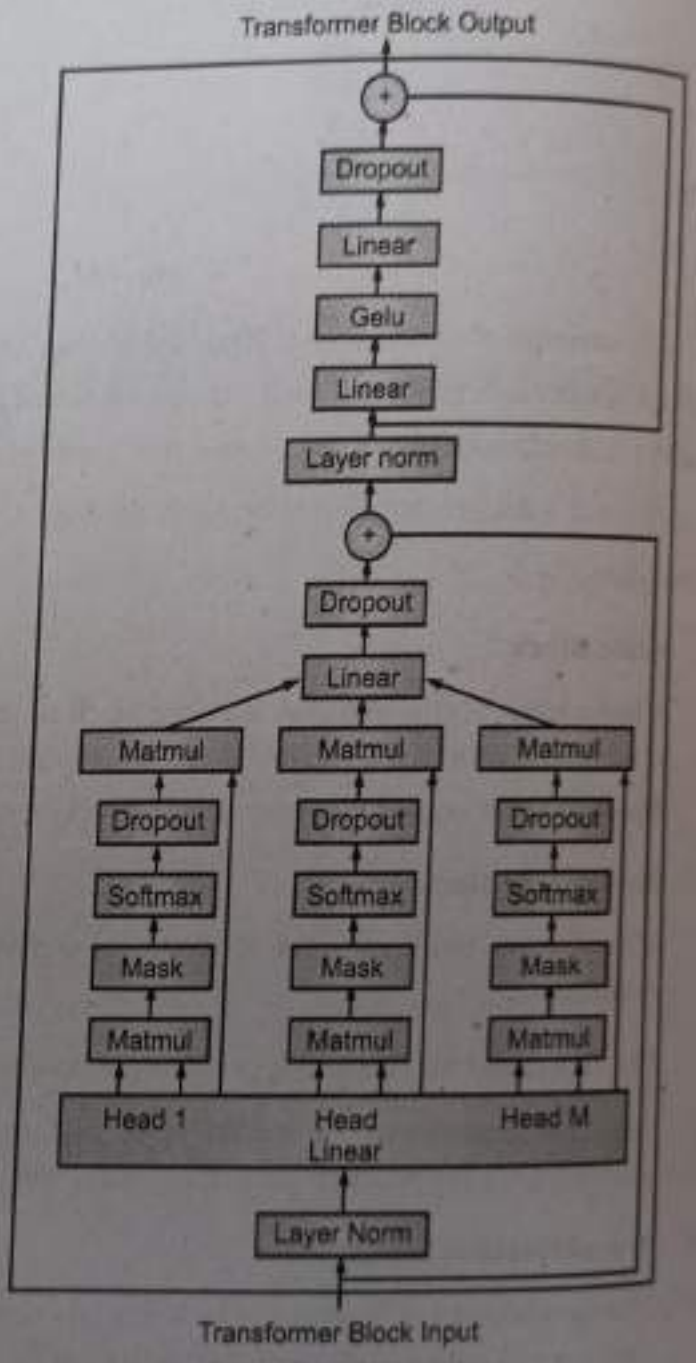


Fig. 4.12.4 : Transformer Architecture

- A transformer block is a stack of two Residual blocks. Each Residual Block has a Residual connection.

- The first Residual block is a multi head Attention Block, and it performs self-attention computation followed by a linear projection
- The second Residual Block is feed-forward Multi-Layer Perception (MLP) Block. It has a linear projection layer and that increases the dimension, and another linear projection that reduces the dimension.
- Thus, we can say that Residual Network (ResNet) architecture is a type of artificial neural network that allows the model to skip layers without affecting performance.

Chapter Ends ...

❑❑❑

MODULE 5

CHAPTER 5

Recurrent Neural Networks

Syllabus

Sequence Learning Problem, Unfolding Computational graphs, Recurrent Neural Network, Bidirectional RNN, Backpropagation Through Time (BTT), Limitation of " Vanilla RNN" Vanishing and Exploding Gradients, Truncated BTT.

Long Short Term Memory (LSTM) : Selective Read, Selective write, Selective Forget, Gated Recurrent Unit (GRU).

5.1 RECURRENT NEURAL NETWORKS (RNN)

5.1.1 Sequence Learning

GQ. What is Sequence Learning Problem. **(4 Marks)**

A sequence learning issue in the context of Recurrent Neural Networks (RNNs) is a challenge where the input data is sequential information and the RNN model is trained to recognize the temporal connections within the data. When processing sequential input, RNNs keep track of hidden state information. This enables them to recognize patterns and long-term dependencies in sequences.

(a) **Sequential Data :** The input data is provided as a sequence of items in a sequence learning problem with RNNs. Anything that appears in a specific order can be considered one of these elements, including words in a phrase, characters in a text, audio samples in speech, time-series data, or frames in a video.

(b) **Time Steps :** There are distinct time steps for each element in the sequence. Each time step can represent a word or a character in tasks involving natural language processing, but in tasks involving time series prediction, each time step is associated with a particular timestamp.

(c) **RNNs :** Recurrent neural networks are built to process sequential data. RNNs can keep concealed state information over time steps because they have connections that create cycles, in contrast to conventional feedforward neural networks. The model can interpret sequential data and take into account the context of earlier time steps while predicting the present one thanks to this hidden state, which serves as memory.

(d) **RNN training :** Backpropagation through time (BPTT), an extension of backpropagation used in feedforward networks, is used to train RNNs. Based on the order of inputs and matching goal outputs, BPTT computes gradients and modifies the model's parameters.

(e) **Sequence-to-Sequence learning :** In some circumstances, the aim of sequence-to-sequence learning is to map input sequences to output sequences. Sequence-to-sequence learning is what this is, and it has plenty of uses, including text summarization and machine translation.

Capturing the temporal connections or patterns in the data poses the fundamental challenge in sequence learning. For instance, in language modelling, anticipating the next word in a sentence depends on the context of the words that came before it. Similar to this, in time series prediction, previous values affect predictions of the future.

GQ. Which are the different domains under Sequence Learning problem in RNN. **(6 Marks)**

The purpose of tasks in the wide category of "sequence learning" is to identify dependencies and patterns in sequential data. Due to its capacity to process sequential data and preserve hidden state information, recurrent neural networks (RNNs) are a

family of deep learning models that are frequently utilized for sequence learning tasks. Here are several RNN-specific subtopics that fall under the general heading of Sequence Learning Problems:

(a) **Language modelling :** Entails foreseeing the likelihood of the subsequent word in a string of words. It is frequently used in activities involving natural language processing, including text generation, audio recognition, and machine translation.

(b) **Text generation :** Builds on language modelling, is the process of creating text that is coherent and contextually appropriate. Examples of text creation include chatbots, automatic text completion, and creative writing.

(c) **Speech Recognition :** Using RNNs, audio signals can be converted into text for use in voice-activated technology, transcription services, and virtual assistants.

(d) **Forecasting :** RNNs are excellent at time series prediction problems, which include predicting future valûes using historical data from observations. Forecasting the stock market, the weather, and sales are a few examples.

(e) **Sequence-to-sequence Learning :** Seq2Seq maps input sequences to output sequences, Seq2Seq models employ RNNs. They are employed in systems for question-answering, summarization, and machine translation.

(f) **Sentiment analysis :** Is utilized in market research, customer feedback analysis, and social media monitoring, uses RNNs to examine sequential data, such as text or audio, to ascertain the sentiment communicated by the author or speaker.

(g) **Named Entity Recognition (NER) :** The process of finding names of persons, places, and other entities in text. RNNs, which are helpful in information extraction and search engines, can be used for this task.

RNNs can be used to create music sequences based on patterns discovered from previously collected musical data, opening up possibilities for applications in music creation and recommendation systems.

RNNs are useful in applications like digitizing handwritten documents and pen-based input systems because they can recognize and translate handwritten text.

RNNs are capable of processing sequential data, such as the frames of a video, making it possible to do tasks like action recognition, video captioning, and gesture recognition. RNNs can be utilized in reinforcement learning configurations where the model learns to conduct successive actions to maximize cumulative rewards.

RNNs can be used to find anomalies in sequential data, such as detecting fraud in financial transactions or spotting flawed patterns in industrial processes.

5.2 UNFOLDING COMPUTATIONAL GRAPHS

5.2.1 Unfolding Computational Graphs

GQ. Explain the principle behind unfolding computational graphs in detail. **(10 Marks)**

Understanding the inner workings of Recurrent Neural Networks (RNNs) requires a basic understanding of how computational graphs unfold. It is simpler to understand the information flow and the temporal dependencies inside the network thanks to the unfolding process, which gives a visual picture of how an RNN analyses sequential data over a number of time steps.

(a) RNN cell operations

- An RNN's fundamental building block is the RNN cell. The RNN cell receives two inputs at each time step, the current input vector x_t and the prior hidden state h_t-1, which represents the data from the previous time step. The hidden state h_t at the current time step is then updated using computations.
- The following equations describe how an RNN cell functions:

  The hidden state at time t:

  $$h\_t = f(W\_h * h\_(t-1) + W\_x * x\_t + b)$$

  where:

  h_t : Hidden state at time step t

  f : Activation function (usually tanh or ReLU)

  W_h : Weight matrix for the hidden state

  W_x : Weight matrix for the input

  b : Bias vector

(b) Unfolding the RNN

- We unfold the RNN across time, resulting in a chain of connected RNN cells, each of which corresponds to a different time step, in order to examine how the RNN processes the sequential input over various time steps.
- We can see the network's temporal dependencies and information flow thanks to this unfolding. The weights and biases of each RNN cell at time step t are shared by that cell and all other RNN cells connected to it at time step t – 1. The unfurled RNN appears as follows up to time step T:

  At time step t = 1 :

  $$h\_1 = (W\_h * h\_0 + W\_x * x\_1 + b)$$

  At time step t = 2 :

$$h\_2 = f(W\_h * h\_1 + W\_x * x\_2 + b)$$

At time step t = 3 :

$$h\_3 = f(W\_h * h\_2 + W\_x * x\_3 + b)$$

At time step t = T :

$$h\_T = f(W\_h * h\_(T-1) + W\_x * x\_T + b)$$

(c) BPTT and Sequence Training

- We employ the Backpropagation Through Time (BPTT) technique during training on the unfolded RNN. A modification of the common backpropagation algorithm used to train RNNs is BPTT. The complete sequence is used to calculate gradients and update the model's parameters.
- At each time step, the loss function is calculated, and the gradients of the loss with regard to the model's weights and biases are back propagated from time step T to time step 1.

(d) Vanishing/Exploding Gradient Problem

The problem of vanishing/exploding gradients is one of the difficulties in training RNNs. Gradients can spread through time during BPTT and either become very small (vanishing) or very huge (exploding). The model may have trouble learning long-term dependencies as a result of this problem.

(e) Addressing the Gradient Problem

- RNN variants such Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) were introduced to overcome the gradient problem. The vanishing/exploding gradient problem is reduced and crucial information is preserved over lengthy sequences because to the employment of gating methods by LSTM and GRU.

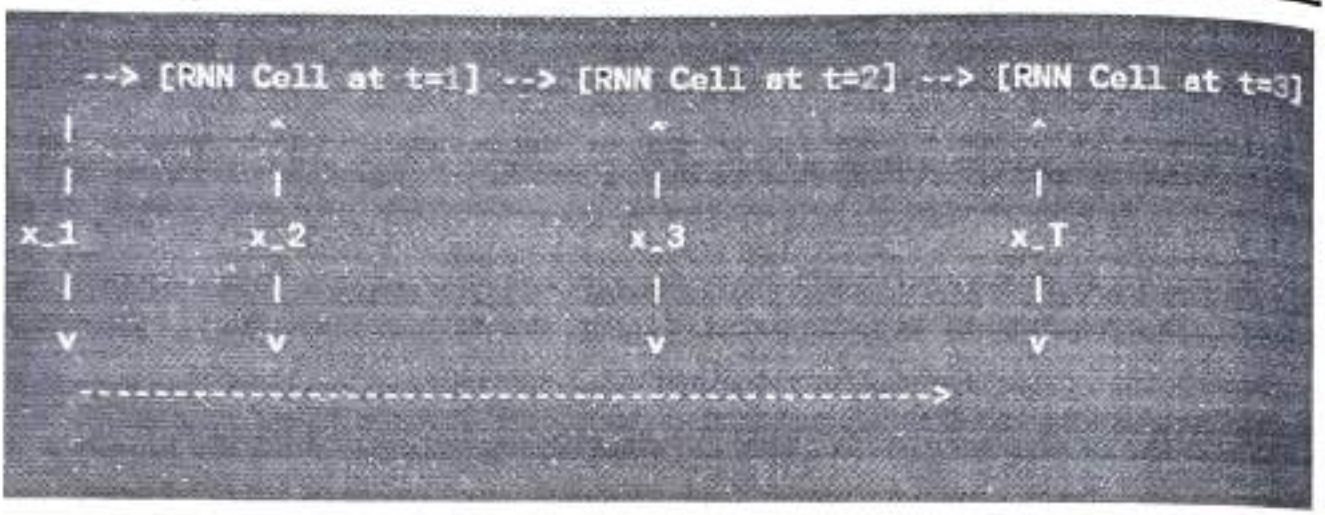
  W_h : Weight matrix for the hidden state

  W_x : Weight matrix for the input

  B : Bias vector

- Consider about a simple RNN with three time steps (T=3). Each time step, the RNN cell receives an input (x_t) and has a single hidden unit (h_t). The nomenclature for weights and biases will be as follows:

  The unfolding computational graph for this RNN up to time step T=3 looks as follows :



- The RNN cells at each time step (t=1, t=2, t=3) are shown by the circles.
- The movement of information and computation through time is depicted by the arrows.
- The associated RNN cell receives the input at each time step (x_t).
- The RNN cell computes the hidden state at each time step (h_t) and transmits it to the following time step.
- All time steps use the same set of weights and biases (W_h, W_x, and b).
- Visualization of the unfolding computational graph demonstrates how the RNN analyzes sequential data across a number of time steps. The RNN can detect temporal relationships in the data since the output at each time step (h_t) depends on both the previous hidden state (h_t-1) and the current input (x_t).

5.3 RECURRENT NEURAL NETWORKS

5.3.1 Overview of Recurrent Neural Networks

GQ. What are Recurrent Neural Networks? Explain. **(10 Marks)**

(a) Overview of Recurrent Neural Networks (RNNs)

- RNNs are a subclass of neural networks created with the purpose of processing sequential data while preserving hidden state information.
- RNNs are thus particularly well adapted for tasks involving sequential data, including natural language processing, time series analysis, speech recognition, and more. This enables RNNs to capture temporal dependencies and patterns in sequences.

(b) RNN Cell Operations

- An RNN cell receives two inputs at each time step, the current input vector x_t and the prior hidden state h_t-1, which represents the data from the previous time step.
- In order to update the hidden state h_t at the current time step, the cell then executes computations. The following equations describe how an RNN cell functions:

  The hidden state at time t :

  $h_t = f(W_h * h_{t-1} + W_x * x_t + b)$

  where:

  h_t : Hidden state at time step t

  f : Activation function (usually tanh or ReLU)

  W_h : Weight matrix for the hidden state

  W_x: Weight matrix for the input

  b : Bias vector

  The activation function f introduces non-linearity to the RNN, allowing it to learn complex patterns in sequential data.

(c) Unrolling the RNN

The RNN cell is unrolled over time to process a whole sequence of length T, resulting in a chain of connected RNN cells, each of which corresponds to a distinct time step. We can see the network's temporal dependencies and information flow thanks to this unfolding. The weights and biases of each RNN cell at time step t are shared by that cell and all other RNN cells connected to it at time step t-1.

Example : RNN for Language Modelling

Let's look at an instance of language modelling utilizing an RNN. Predicting the likelihood of the following word in a sequence based on the context of the preceding words is the challenge of language modelling.

Imagine that the statement is, "I love to eat ice cream."

The next word "in" should be predicted by an RNN given the statement "I love to eat ice cream."

(a) Equations

- Input at time step t: x_t = word embedding of the t-th word in the sentence.
- Hidden state at time step t: $h_t = f(W_h * h_{t-1} + W_x * x_t + b)$

  where:

  x_t : Word embedding vector of the t-th word (e.g., a one-hot vector representation).

h_t : Hidden state at time step t.
W_h : Weight matrix for the hidden state.
W_x : Weight matrix for the word embedding.
B : Bias vector.
Output at time step t: y_t = softmax(W_y * h_t + b_y)
where:
y_t : Output probability distribution over the vocabulary at time step t.
W_y : Weight matrix for the output layer.
b_y : Bias vector.

(b) Training the RNN

At each time step of training, we compare the predicted probability distribution (y_t) with the actual target word (ground truth) using the Cross-Entropy Loss function. The losses at each time step are added together to form the overall loss. In order to reduce the loss, we compute gradients using Backpropagation Through Time (BPTT) and adjust the model's weights and biases.

(c) Generating Text

We compare the predicted probability distribution (y_t) with the actual target word (ground truth) at each time step during training using the Cross-Entropy Loss function. The sum of the losses at each time step represents the overall loss. In order to reduce the loss, we compute gradients and update the model's parameters (weights and biases) using Backpropagation Through Time (BPTT).

5.3.2 Architecture of Recurrent Neural Networks

GQ. Explain in detail architecture of Recurrent Neural Networks. **(10 Marks)**

A recurrent neural network (RNN) architecture is created to handle sequential data by preserving hidden state variables across time. The network can record temporal dependencies and patterns in sequences because to its hidden state. The following are the main elements of an RNN architecture :

(a) Input Layer : An RNN gets an input vector at each time step that represents the data for that time step. This input for tasks involving language can be either a word embedding or a one-hot encoded version of the word. The input at time step t will be referred to as x_t.

(b) Hidden State (Memory) : An RNN's hidden state (memory) is a representation of the knowledge the network has gathered over the course of past time steps. It functions as a memory that stores pertinent information from the past and affects how the current time step is processed. The symbol for the hidden state at time step t is h_t.

(c) **Activation Function :** To add non-linearity, an activation function is added to the RNN cell's output. ReLU (Rectified Linear Unit) and tanh (hyperbolic tangent) are two popular activation functions used in RNNs.

(d) **Output Layer :** At each time step, the RNN's calculations are produced by the output layer. The result can be a single value (for example, in time series prediction) or a probability distribution across a set of classes (for example, in language modelling), depending on the job. y_t stands for the output at time step t.

(e) **RNN Cell Operations :** The fundamental operation of an RNN cell is to update the hidden state based on the current input and the previous hidden state. The hidden state at time step t is computed as follows :

$$h\_t = f(W\_h * h\_(t-1) + W\_x * x\_t + b)$$

where :

h_t: Hidden state at time step t.

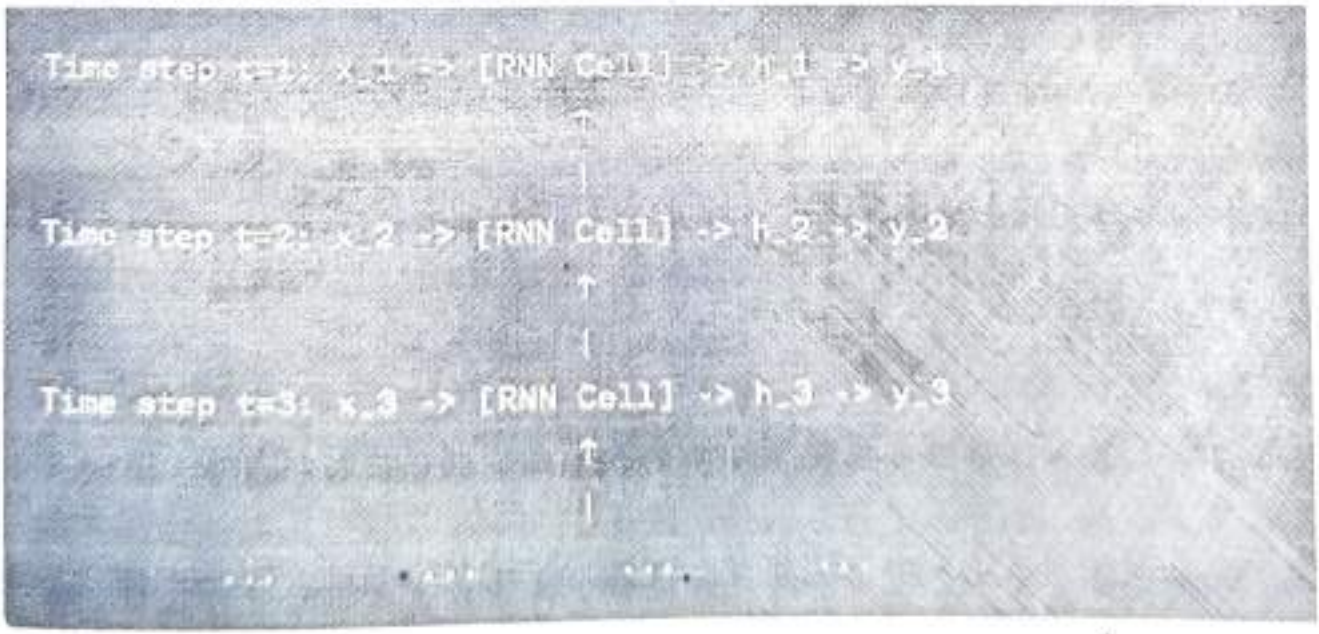
f : Activation function (e.g., tanh or ReLU).

W_h : Weight matrix for the hidden state.

W_x : Weight matrix for the input.

b : Bias vector.

(f) **Unfolding the RNN :** The RNN cell is unrolled over time to process a sequence of length T, resulting in a chain of connected RNN cells, each of which corresponds to a distinct time step. We can see the network's temporal dependencies and information flow thanks to this unfolding. The weights and biases of each RNN cell at time step t are shared by that cell and all other RNN cells connected to it at time step t-1.



(g) Training the RNN : During training, we use the Backpropagation Through Time (BPTT) algorithm to compute gradients and update the model's parameters (weights and biases). The loss function is calculated at each time step, and the gradients are backpropagated through time from the last time step T to the first time step 1. The overall loss is the sum of the losses at each time step.

Example - Time Series Prediction

Architecture and Equations

Let's consider an example of using an RNN for time series prediction. Given a time series of past observations, we want to predict the value at the next time step.

Input at time step t: x_t = value of the time series at time step t.

Hidden state at time step t :

$$h\_t = f(W\_h * h\_(t-1) + W\_x * x\_t + b)$$

Output at time step t: $y\_t = W\_y * h\_t + b\_y$

where:

y_t : Predicted value at time step t.

W_y : Weight matrix for the output layer.

b_y : Bias vector.

Calculation

- When training, we compare the predicted value (y_t) with the actual target value at each time step using the Mean Squared Error (MSE) loss function. The sum of the losses at each time step represents the overall loss. Then, in order to reduce the loss, we compute gradients using BPTT and change the model's parameters.
- Input layers, hidden states, activation functions, and output layers make up the architecture of an RNN. To learn about temporal dependencies in sequential data, we apply BPTT during training and visualize the RNN's actions by unrolling it across time. RNNs are effective models utilized in a wide range of sequence-to-sequence tasks, including time series prediction and language modelling.

5.4 BIDIRECTIONAL RECURRENT NEURAL NETWORKS

5.4.1 Overview and Architecture of Bidirectional Recurrent Neural Networks

GQ. Explain the architecture of Bidirectional Neural Network in detail. List few applications. **(10 Marks)**

A class of neural networks known as bidirectional neural networks (Bi-RNNs) processes sequential data simultaneously in both the forward and backward directions.

Bi-RNNs additionally take into consideration future information to improve the comprehension of the context and dependencies in the data, in contrast to regular RNNs, which solely take into account past information to predict the future. In situations where the present prediction depends on both past and future context, this makes them extremely effective.

1. **Architecture of Bidirectional RNNs :** Bidirectional RNNs have the following architectural design:

   The input sequence is fed into two different RNNs :

(a) **Forward RNN (or Left-to-Right RNN) :** The forward RNN, also known as a left-to-right RNN, processes the input sequence going forward (from the first to the last time step). The forward RNN modifies its hidden state based on the current input and the prior hidden state at each time step.

(b) **Backward RNN (or Right-to-Left RNN) :** The backward RNN (also known as a right-to-left RNN) processes the input sequence backward (from the last to the first-time step). Based on the current input and the prior hidden state, the backward RNN updates its hidden state at each time step.

The final representation of the input at each time step is then created by combining the hidden states of the forward and backward RNNs, enabling the model to take into account both past and future context.

1. **Formulation of Bidirectional RNNs**

- Let's denote the input sequence as X = [x_1, x_2, ..., x_T], where x_t represents the input at time step t. The hidden states of the forward and backward RNNs at time step t are denoted as h_t^f and h_t^b, respectively.
- The computation for the forward RNN at time step t is given by :

  h_t^f = f(W_h^f * h_{t-1}^f + W_x^f * x_t + b^f)
- Similarly, the computation for the backward RNN at time step t (working in reverse) is given by:

  h_t^b = f(W_h^b * h_{t+1}^b + W_x^b * x_t + b^b)

  where :

  f: Activation function (e.g., tanh or ReLU).

  W_h^ f, W_h^b: Weight matrices for the hidden states of the forward and backward RNNs, respectively.

  W_x^f, W_x^b: Weight matrices for the input of the forward and backward RNNs, respectively.

  b^f, b^b: Bias vectors for the hidden states of the forward and backward RNNs, respectively.

- The final representation of the input at time step t in the Bi-RNN is given by the concatenation of the hidden states from the forward and backward RNNs :

  h_t = [h_t^f; h_t^b]

  Then, the combined hidden states of the forward and backward RNNs are used to

  Bidirectional RNNs have the following architectural design :

- The input sequence is fed into two different RNNs:
- The forward RNN, also known as a left-to-right RNN, processes the input sequence going forward (from the first to the last time step). The forward RNN modifies its hidden state based on the current input and the prior hidden state at each time step.

2. Applications of Bidirectional RNNs

Bidirectional RNNs are frequently employed in jobs where accurate prediction depends on future context. Among the uses for bi-RNNs are :

- **Named Entity Recognition (NER) :** When identifying names, places, and organizations in text, it's important to take into account both the words that come before and after them.
- **Part-of-Speech (POS) Tagging :** Speech component (POS) Knowing the surrounding context is helpful when tagging words in a phrase with their appropriate parts of speech.
- **Machine Translation :** Understanding the context on both sides of the statement can help machine translation provide better translation outcomes.
- **Speech Recognition :** Bidirectional RNNs are used to extract future and historical information from audio sequences.
- Bidirectional RNNs, which analyze sequential data in both forward and backward directions, are a powerful expansion of conventional RNNs. Bi-RNNs have shown to be efficient in a variety of sequence processing tasks by adding future context, which enhances the model's comprehension of the context and dependencies within the data.

5.4.2 Recurrent Neural Network and Bidirectional Recurrent Neural Network

GQ. Explain in the difference between Recurrent Neural Networks and Bidirectional Neural Networks. **(8 Marks)**

Recurrent Neural Networks (RNNs) and Bidirectional Recurrent Neural Networks (Bi-RNNs) are two different neural network architectures intended for sequential data processing. They differ, nevertheless, in terms of how they are built and how they use the input sequence's temporal information. Let's examine the main distinctions between RNNs and Bi-RNNs in more detail:

1. **Data Processing Direction**
   - **RNN's :** Traditional RNNs only process the input sequence going forward, from the first-time step to the last time step. RNNs can capture dependencies within the sequence in the forward direction because they have recurrent connections that keep concealed state information.
   - **Bi-RNN's :** Bi-RNNs, on the other hand, simultaneously process the input sequence going both forward and backward. They are made up of two independent RNNs, one of which processes the sequence from the first-time step to the last, and the other of which processes the sequence backward from the last to the first. Due to its bidirectional processing, Bi-RNNs can gather data from both historical and speculative settings.

2. **Hidden State Representation:**
   - **RNNs :** In RNNs, the hidden state is solely dependent on the current input and the prior hidden state at each time step. As a result, the RNN is able to detect temporal relationships that span the past and present.
   - **Bi-RNNs :** In Bi-RNNs, two distinct hidden states-one from the forward RNN (h_tf) and one from the backward RNN (h_tb)-are present at each time step. These two hidden states, h_t_f and h_t_b, respectively, reflect information from the past and the future, capturing different facets of the input sequence.

3. **Information Flow**
   - **RNNs :** As the information flow in RNNs is unidirectional, the model can only forecast future time steps using data from prior ones. When the data contains long-range dependencies, it could run into problems.
   - **Bi-RNNs :** In Bi-RNNs, information flows in both directions, enabling the model to make predictions using data from both past and future time steps. Bi-RNNs can perform jobs that call for understanding both past and future information more effectively because of their improved ability to grasp long-range dependencies and context.

4. **Training and Computation**
   - **RNNs :** Since RNNs only operate in the forward direction, their training and calculation are rather simple. Backpropagation Through Time (BPTT) is used to train the model, and the recurrent connections update the hidden state at each time step.
   - **Bi-RNNs :** Due to their bidirectional character, bi-RNNs call for additional calculation. In comparison to unidirectional RNNs, bidirectional RNNs need updating both the forward and backward RNNs during training, which might increase computational complexity.

5. Applications

- **RNNs :** RNNs are frequently employed in jobs that need sequential data, such as speech recognition, time series analysis, and natural language processing.
- **Bi-RNNs :** Bi-RNNs excel at activities that need a thorough awareness of both the present and the future. They are frequently employed in tasks including speech recognition, machine translation, named entity recognition (NER), and part-of-speech (POS) tagging.

In conclusion, the direction of input processing and the usage of bidirectional information in Bi-RNNs are the main distinctions between Bidirectional Recurrent Neural Networks and Recurrent Neural Networks. Bi-RNNs can capture richer temporal connections and perform better in some sequential data processing tasks by taking into account both past and future context. In contrast to conventional RNNs, they might need higher processing power.

5.5 BACKPROPAGATION THROUGH TIME BTT

5.5.1 Backpropagation in RNN

GQ. What is backpropagation in RNN? Explain in detail. **(4 Marks)**

The extension of the backpropagation method used to train recurrent neural networks (RNNs) is called backpropagation through time (BPTT). While tackling the inherent difficulties of handling variable-length sequences, BPTT takes into account the sequential nature of data and enables the RNN to learn from the full input sequence.

1. Overview of Backpropagation Through Time (BPTT)

- Backpropagation, a technique used in conventional feedforward neural networks, computes gradients and modifies the network's parameters depending on the difference between the true goal and the expected output at a single time step. RNNs, on the other hand, update the hidden state at each time step based on the input and the prior hidden state, which establishes a temporal relationship between the time steps.
- By unrolling the RNN across time, BPTT expands backpropagation to take into account this temporal link. Backpropagation is carried out over an unrolled computational graph while treating the RNN as a deep neural network with shared weights across time steps. By doing this, BPTT allows the model to learn from the entire sequence and takes into consideration the dependencies between the time steps.

2. BPTT Equations

Assume a simple RNN architecture with a single hidden unit and a single output unit. The forward pass equations for a single time step t are as follows:

Hidden state update:

$$h\_t = f(W\_h * h\_\{t-1\} + W\_x * x\_t + b)$$

Output computation :

$$y\_t = W\_y * h\_t + b\_y$$

Where :

h_t : Hidden state at time step t.

f : Activation function (e.g., tanh or ReLU).

W_h : Weight matrix for the hidden state.

W_x : Weight matrix for the input.

b : Bias vector.

x_t : Input at time step t.

y_t : Output at time step t.

W_y : Weight matrix for the output.

b_y : Bias for the output.

5.5.2 BTT Algorithm

GQ. Explain BPTT Algorithm in detail. Discuss various challenges and solutions. **(8 Marks)**

BPTT Algorithm

The BPTT algorithm involves the following steps :

- **Step 1 : Forward Pass**

For the whole input sequence, unroll the RNN over time, and calculate the forward pass for each time step. The hidden state is updated throughout the forward pass, and the RNN's equations are used to compute the output at each time step.

- **Step 2 : Calculating the Loss**

By contrasting the expected output (y_t) with the actual target (the ground truth) at each time step, determine the loss at each time step. The sum of the losses at all time steps is the overall loss. Depending on the particular task, several loss functions are employed, such as Mean Squared Error (MSE) for regression or Cross-Entropy Loss for classification.

▶ **Step 3 : Back propagation**

To calculate gradients with regard to the RNN's parameters (weights and biases), use backpropagation across time. The gradients are computed via the chain rule starting from the last time step (T) and going back to the initial time step (t = 1). As they share weights, the gradients are accumulated over time steps.

▶ **Step 4 : Gradient Update**

Using the obtained gradients, modify the RNN's parameters. For this update, you can use the common gradient descent or one of its variations (such Adam or RMSprop). Based on the particular problem and design, an optimization technique and learning rate can be selected.

Challenges with BPTT

Although BPTT is a useful technique for RNN training, it has significant drawbacks, particularly for extended sequences :

- **Vanishing/ Exploding Gradient Problem :** Gradients, especially in deep RNNs, can either become very small (vanishing) or very huge (exploding) as they are propagated backward through time. Long-term dependency learning may be difficult as a result of this problem.
- **Memory Requirements :** BPTT has greater memory requirements, especially for extended sequences, because it must store all intermediate concealed states during the forward pass.

Solutions to BPTT Challenges

Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are two examples of sophisticated RNN architectures that have been created to overcome the vanishing/exploding gradient problem and increase training efficiency. These topologies employ gating methods to regulate the information flow, enabling RNNs to keep track of crucial data across longer sequences.

Additionally, shortened BPTT, which accumulates gradients over a limited number of time steps rather than the complete series, is frequently employed in practice. For longer sequences, this reduces the need for memory and solves the vanishing/exploding gradient issue.

5.6 LIMITIATION OF "VANILLA RNN" VANISHING AND EXPLODING GRADIENTS

5.6.1 Vanishing and Exploding Gradients

GQ. Explain Vanishing and Exploding Gradients. **(4 Marks)**

The problem of vanishing and exploding gradients during training is one of Vanilla Recurrent Neural Networks' (RNNs') primary drawbacks. The recurrent nature of RNNs, where gradients are propagated through time, results in these issues, which can make it challenging to learn long-term dependencies in sequential data.

1. **Vanishing Gradients :** As a gradient moves backward in time, its magnitude gets less and smaller until it vanishes entirely. This occurs when the recurrent weights are nearly zero or when the activation function, such as the sigmoid function, utilized in the RNN has a small derivative. Because of this, the model has trouble updating the weights in earlier time steps, which prevents it from discovering long-term dependencies.
2. **Exploding Gradients :** On the other hand, explosive gradients happen when a gradient's magnitude increases dramatically during backpropagation. This may occur if the activation function has a high derivative or if the recurrent weights are substantial. The model diverges during training as a result of the steep gradients' heavy weight updates.

Understanding Vanishing Gradients

Consider the RNN update equation for a single time step t:

$$h\_t = f(W\_h * h\_(t-1) + W\_x * x\_t + b)$$

where :

h_t : Hidden state at time step t.

f : Activation function (e.g., tanh or sigmoid).

W_h : Weight matrix for the hidden state.

W_x : Weight matrix for the input.

b : Bias vector.

x_t : Input at time step t.

During backpropagation, the gradient of the loss with respect to the hidden state ($\partial L/\partial h\_t$) is calculated and propagated backward to update the weights (W_h, W_x) and biases (b). The gradient update involves taking the derivative of the activation function (f) and is multiplied with the chain rule as gradients are propagated through time.

If the activation function is the sigmoid function ($f(z) = 1 / (1 + \exp(-z))$), its derivative is given by:

$$f'(z) = f(z) * (1 - f(z))$$

As the absolute value of f'(z) approaches zero, the gradient vanishes, leading to the vanishing gradient problem.

Understanding Exploding Gradients

The recurrent weights (W_h) at each time step, on the other hand, are doubled when gradients are propagated backward in time. The exploding gradient problem occurs when the gradients expand exponentially as they progress backward through the sequence when the recurrent weights are significant (more than 1).

5.6.2 Limitation of "Vanilla RNN"

GQ. Write a code for simple Vanilla RNN. **(6 Marks)**

```
import numpy as np

# Define the activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Initialize the recurrent weight matrix
W_h = np.random.randn(1, 1) * 0.01

# Define a sequence of inputs and targets
sequence_length = 5
X = np.random.randn(sequence_length, 1)
y = np.random.randn(sequence_length, 1)

# Forward pass and backpropagation
h_prev = np.zeros((1, 1))  # Initial hidden state
gradients = []  # To store the gradients for each time step

for t in range(sequence_length):
    # Forward pass
    h_t = sigmoid(np.dot(W_h, h_prev) + X[t])

    # Calculate the loss and the gradient of the loss with respect to the hidden state
    loss_t = 0.5 * (h_t - y[t]) ** 2
```

```
    gradient_h_t = h_t - y[t]

    # Backpropagation
    gradients.append(gradient_h_t)
    gradient_h_prev = np.dot(W_h.T, gradient_h_t) * sigmoid_derivative(h_t)

    # Update the hidden state for the next time step
    h_prev = h_t

# Print the gradients for each time step
for t in range(sequence_length):
    print(f"Gradient at time step {t}: {gradients[t]}")
```

In this code example, we can observe that the gradients for the hidden state (gradient_h_t) decrease exponentially as we move backward through time, indicating the vanishing gradients problem.

GQ. Explain Limitations of "Vanilla RNN" in detail. **(4 Mark)**

The following are the limitations of "Vanilla RNN":

1. **Vanishing and Exploding Gradients :** The vanishing and expanding gradients problem can occur when training vanilla RNNs. The vanishing gradients problem arises when the gradients are transmitted backward in time and become incredibly tiny. As a result, learning long-term dependencies is challenging since the model finds it challenging to change weights in earlier time steps. The exploding gradients issue, on the other hand, happens when gradients grow to exceptionally large sizes, resulting in unstable weight updates and causing the model to diverge during training.
2. **Short-Term Memory :** Vanilla RNNs have a finite amount of memory. They struggle to recall details from far earlier time steps, but they are adept at capturing dependencies within a brief time range. Their inability to preserve context over extended sequences, which is essential for jobs involving long-range dependencies, is hampered by this restriction.
3. **Difficulty in Capturing Long-Term Dependencies :** The vanishing gradients issue makes it challenging for Vanilla RNNs to identify long-term dependencies in sequential data. In tasks requiring long-range context, information from distant past time steps may not have a major impact on the predictions, resulting in less-than-ideal performance.
4. **Fixed-Length Input and Output Sequences :** The input and output sequence lengths for vanilla RNNs must be fixed. It is frequently necessary to truncate or pad sequences for activities with variable-length sequences (like natural language processing), which can lead to information loss or inefficient processing.

5. **Sequential Processing Limitations :** The sequential processing of sequences by vanilla RNNs prevents parallelization during training and inference. Slower training times may result from this sequential structure, particularly for lengthy sequences.
6. **Gradient Explosion in Deep RNNs :** The gradient explosion problem can worsen as Vanilla RNNs are layered to create deep RNN architectures, making training even more difficult.
7. **Lack of Robustness to Noisy Inputs :** Plain RNNs are susceptible to erroneous or incomplete inputs. The robustness of the model can be impacted by a slight perturbation or lack of input, which can cause major differences in the output.
8. **Difficulty in Capturing Long Dependencies in Multimodal Data :** Vanilla RNNs may have trouble capturing intricate connections between several modalities for multimodal data (such as merging text and images over long time scales), which restricts their applicability in jobs requiring the fusion of data from various sources.
9. **Gating Mechanisms Absent :** In advanced RNN architectures like LSTMs (Long Short-Term Memory) and GRUs (Gated Recurrent Units), gating mechanisms are absent from vanilla RNNs. RNNs can manage the flow of information, keep track of crucial information over lengthy sequences, and solve the vanishing/exploding gradient problem thanks to gating mechanisms.

Despite these drawbacks, Vanilla RNNs have their uses, especially when working with short sequences and straightforward dependencies. Advanced RNN architectures like LSTMs and GRUs are chosen for applications needing long-range context and handling more intricate temporal interactions. These architectures have replaced Vanilla RNNs as the go-to option for many sequence-to-sequence workloads since they overcome many of their shortcomings.

5.7 TRUNCATED BTT

5.7.1 Truncated BTT

GQ. Explain Truncated BTT in detail. Write algorithm for truncated BTT. List advantages. **(10 Marks)**

Recurrent Neural Networks (RNNs) must be trained with lengthy sequences, which presents a hurdle. Truncated Backpropagation Through Time (Truncated BPTT) is a method used to overcome this challenge. It is a modification of the popular Backpropagation Through Time (BPTT) technique that accounts for the temporal dependencies in RNNs. Truncated BPTT reduces memory requirements and mitigates the issues with vanishing and expanding gradients by cutting large sequences into smaller chunks.

1. Overview of Truncated Backpropagation Through Time (Truncated BPTT)

Gradients are calculated and accumulated over the whole sequence in normal BPTT, which can be computationally and memory-intensive, particularly for extended sequences. Truncated BPTT resolves this problem by breaking the lengthy sequence into smaller units. Gradients are only propagated over a predetermined amount of time steps (truncated sequence length) as opposed to being propagated throughout the complete sequence. The model can learn dependencies over shorter segments thanks to this truncation, which also reduces the temporal range over which gradients must be computed.

Truncated BPTT Algorithm

The steps of the Truncated BPTT algorithm are as follows :

- **Step 1 : Truncation of Sequence :** The input sequence is broken up into smaller chunks or segments that are a defined length. Each segment is a condensed version of the original lengthy sequence.
- **Step 2 : Forward Pass and Loss Calculation :** The forward pass is used to compute the hidden states and predictions for each truncated segment. Every time step in the reduced segment is used to calculate the loss.
- **Step 3 : Backpropagation :** The gradients are calculated for each shortened segment, working backward from the segment's most recent time step. The backpropagation is, however, terminated at the end of each segment. This indicates that gradients from one segment are not carried over to the earlier segments. Instead, the hidden state from the final segment of the previous segment is used to initialize the hidden state at the border.
- **Step 4 : Gradient Accumulation and Parameter Update :** The gradients determined in each truncated segment are added together across the full sequence in step four, which also updates the parameters. Gradient descent or its derivatives, which are common gradient-based optimization methods, are used to update the model's parameters (weights and biases) once the gradients for all segments have been computed.

Truncated BPTT has the following advantages :

1. **Reduced Memory Requirements :** By simply computing and storing gradients for the truncated segments rather than the complete sequence, truncating the sequence decreases the amount of memory required during training.
2. **Mitigating Vanishing and Exploding Gradients :** Truncated BPTT, particularly in deep RNN architectures, helps reduce the issue of vanishing and inflating gradients by restricting the temporal range during which gradients are computed.

3. **Efficient Training :** Truncated BPTT eliminates the need to keep all intermediate hidden states for the whole sequence, making training more effective, especially for lengthy sequences.

Choosing Truncated Sequence Length

The task-specific requirements and the model's architectural design have a major role in the selection of the truncated sequence length. The model may lose long-term dependencies and perform poorly on tasks requiring long-range context if the truncated sequence length is too short. On the other side, if it is too long, the advantages of truncation might be reduced, and the effectiveness of the training might be jeopardized. The truncated sequence length is typically selected as a hyperparameter in practice, and it is established through experimentation and confirmation.

In conclusion, the technique known as Truncated Backpropagation Through Time (Truncated BPTT) solves the computational and memory issues associated with training RNNs with large sequences. Truncated BPTT reduces the length of the sequence into smaller chunks, allowing RNNs to learn dependencies over shorter timescales while also reducing the issues with vanishing and exploding gradients that come with large sequences.

5.8 LONG SHORT-TERM MEMORY (LSTM)

5.8.1 LSTM

GQ. Explain LSTM in brief. **(6 Marks)**

Long Short-Term Memory, sometimes known as LSTM, is a sort of sophisticated Recurrent Neural Network (RNN) architecture. The disappearing and ballooning gradient difficulties, as well as the challenge of capturing long-term relationships in sequential data, are some of the shortcomings of conventional RNNs that LSTM was created to overcome.

Simply put, the LSTM introduces a unique memory cell and three gating mechanisms (input gate, forget gate, and output gate) that allow it to selectively keep or forget information over time. As a result, LSTM can handle long-range relationships and keep crucial information over lengthy sequences.

- **Memory Cell :** An LSTM's main building block is a memory cell. It is updated using the input, forget, and output gates and stores the context or data from earlier time steps.
- **Input Gate :** The input gate chooses which data from the most recent concealed state and the current time step should be added to the memory cell.
- **Forget Gate :** The forget gate makes the decision as to which data in the memory cell should be deleted or forgotten.

- **Output Gate :** The output gate regulates how much of the data stored in the memory cell should be used to create the output or hidden state for the current time step.

It is highly effective in a variety of applications, including natural language processing, audio recognition, time series prediction, and more, because to the LSTM architecture's ability to handle extended sequences and learn complicated patterns in sequential data. One of the most well-known and effective RNN variations used in deep learning is the LSTM because of its capacity to capture long-term dependencies and alleviate the vanishing gradient problem.

GQ. Write a code for LSTM cell in python. **(6 Marks)**

```
import numpy as np

# Define the LSTM cell
def lstm_cell(x_t, h_prev, C_prev, W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc,
b_c):
    # Calculate candidate cell
    C_tilde = np.tanh(np.dot(W_xc, x_t) + np.dot(W_hc, h_prev) + b_c)

    # Calculate input and forget gates
    i_t = sigmoid(np.dot(W_xi, x_t) + np.dot(W_hi, h_prev)
                  + b_i)
    f_t = sigmoid(np.dot(W_xf, x_t) + np.dot(W_hf, h_prev)
                  + b_f)

    # Update memory cell
    C_t = f_t * C_prev + i_t * C_tilde

    # Calculate output gate
    o_t = sigmoid(np.dot(W_xo, x_t) + np.dot(W_ho, h_prev)
                  + b_o)

    # Calculate hidden state
    h_t = o_t * np.tanh(C_t)

    return h_t, C_t

# Helper function for sigmoid activation
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Example usage
input_dim = 3
```

```
hidden_dim = 2

x_t = np.random.randn(input_dim, 1)
h_prev = np.random.randn(hidden_dim, 1)
C_prev = np.random.randn(hidden_dim, 1)

# Random weight matrices and bias vectors
W_xi, W_hi, b_i = np.random.randn(hidden_dim, input_dim), np.random.randn(hidden_dim, hidden_dim), np.random.randn(hidden_dim, 1)
W_xf, W_hf, b_f = np.random.randn(hidden_dim, input_dim), np.random.randn(hidden_dim, hidden_dim), np.random.randn(hidden_dim, 1)
W_xo, W_ho, b_o = np.random.randn(hidden_dim, input_dim), np.random.randn(hidden_dim, hidden_dim), np.random.randn(hidden_dim, 1)
W_xc, W_hc, b_c = np.random.randn(hidden_dim, input_dim), np.random.randn(hidden_dim, hidden_dim), np.random.randn(hidden_dim, 1)

h_t, C_t = lstm_cell(x_t, h_prev, C_prev, W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc, b_c)
print("Hidden State (h_t):")
print(h_t)
print("\nMemory Cell (C_t):")
print(C_t)
```

5.9 SELECTIVE READ IN LSTM

5.9.1 Selective Read

GQ. Explain selective read in LSTM in brief. **(6 Marks)**

In the context of LSTM architecture, the word or idea "selective read" is not often used. I can, however, describe how LSTMs' attention mechanisms-which can be viewed as a type of selective reading-work.

Attention Mechanism in LSTM

The attention mechanism is frequently employed in the context of LSTM to concentrate on particular segments of the input sequence when making predictions or producing output. It enables the model to selectively focus on pertinent information while ignoring irrelevant input sequence elements.

Equations for Attention Mechanism in LSTM

Let's think about the attention mechanism applied to LSTM. The attention mechanism computes the context vector c_t, which is a weighted sum of the input sequence elements, given an input sequence of length T, each element represented as x_t, and the hidden state of the LSTM at each time step t as h_t.

Typically, a set of learnable parameters—often referred to as attention weights or attention scores—are used to create the attention mechanism. The attention scores will be defined as follows:

$e_t = score(h_t, x_t)$

where score is a function that computes the compatibility or similarity between the hidden state h_t and the input element x_t. The attention scores are then normalized using the softmax function to get the attention weights:

$a_t = softmax(e_t)$

The context vector c_t is computed as the weighted sum of the input elements based on the attention weights:

$c_t = \Sigma(a_t * x_t)$

The context vector c_t is then used as an additional input to the LSTM, allowing it to focus on the relevant information from the input sequence while making predictions.

GQ. Write a code for Attention Mechanism in LSTM. **(6 Marks)**

```
import numpy as np
import tensorflow as tf

# Define the input sequence and hidden state of the LSTM
input_sequence = np.random.randn(5, 3)  # 5 time steps, 3 features per time step
hidden_state = np.random.randn(1, 2)  # Hidden state of the LSTM with 2 units

# Convert the input sequence and hidden state to tensors
input_sequence_tensor = tf.constant(input_sequence, dtype=tf.float32)
hidden_state_tensor = tf.constant(hidden_state, dtype=tf.float32)

# Define the attention mechanism
```

```
def attention_mechanism(hidden_state, input_sequence):
    # Compute attention scores (compatibility between hidden_state and each element in input_sequence)
    scores = tf.matmul(input_sequence, tf.expand_dims(hidden_state, axis=-1))
    scores = tf.squeeze(scores, axis=-1)  # Remove the last dimension

    # Apply softmax to obtain attention weights
    attention_weights = tf.nn.softmax(scores, axis=0)

    # Compute context vector as the weighted sum of input_sequence elements
    context_vector = tf.reduce_sum(input_sequence * tf.expand_dims(attention_weights, axis=-1), axis=0)

    return context_vector

# Get the context vector using the attention mechanism
context_vector = attention_mechanism(hidden_state_tensor, input_sequence_tensor)

print("Input Sequence:")
print(input_sequence)
print("\nHidden State of LSTM:")
print(hidden_state)
print("\nContext Vector (Computed by Attention Mechanism):")
print(context_vector.numpy())
```

In this code example, we build an attention mechanism that computes the context vector using the input sequence and the hidden state of the LSTM as inputs. The context vector is the weighted sum of the input sequence elements, with the weights being set by the attention mechanism according to how well each input piece fits with the hidden state.

5.10 SELECTIVE WRITE IN LSTM

5.10.1 LSTM

GQ. Explain architecture of LSTM in brief. **(6 Marks)**

In the context of LSTM architecture, the word or notion "selective write" is not often used. Long Short-Term Memory (LSTM) is a sort of recurrent neural network created to manage long-range dependencies in sequential data and circumvent the vanishing gradient problem. Although LSTM features gating mechanisms that allow it to regulate the flow of information and update the memory cell selectively, the term "selective write" is not used to designate a particular method in LSTM.

LSTM Architecture

The candidate cell, input gate, forget gate, output gate, and memory cell make up the LSTM. Together, these elements enable the LSTM to selectively remember or forget data over time, making it very efficient at managing long-term dependencies in sequential data.

Equations for LSTM

Let's define the equations governing the information flow in an LSTM cell for a single time step t:

Candidate Cell (C_tilde) Calculation:

The candidate cell represents new information that could be added to the memory cell.

C_tilde = tanh(W_x * x_t + W_h * h_{t – 1} + b_c)

where :

x_t is the input at time step t.

h_{t-1} is the hidden state at the previous time step.

W_x and W_h are weight matrices for the input and hidden state, respectively.

b_c is the bias vector for the candidate cell.

Input Gate (i_t) and Forget Gate (f_t) Calculation:

The input gate determines how much of the candidate cell to add to the memory cell, while the forget gate decides how much information to retain from the previous memory cell.

i_t = sigmoid(W_{xi} * x_t + W_{hi} * h_{t – 1} + b_i)

f_t = sigmoid(W_{xf} * x_t + W_{hf} * h_{t – 1} + b_f)

where:

W_{xi}, W_{hi}, W_{xf}, and W_{hf} are weight matrices for the input and hidden state corresponding to the input and forget gates, respectively.

b_i and b_f are bias vectors for the input and forget gates, respectively.

Memory Cell (C_t) Update

The memory cell is updated by combining the information from the candidate cell and the previous memory cell based on the input and forget gates.

C_t = f_t * C_{t – 1} + i_t * C_tilde

Output Gate (o_t) Calculation

The output gate determines how much of the updated memory cell to expose as the output of the LSTM cell.

o_t = sigmoid(W_{xo} * x_t + W_{ho} * h_{t – 1} + b_o)

Hidden State (h_t) Calculation

The hidden state is the output of the LSTM cell and is obtained by applying the output gate to the updated memory cell.

h_t = o_t * tanh(C_t)

5.10.2 Selective Write

GQ. Explain selective write in LSTM in brief. **(4 Marks)**

- In the context of LSTM (Long Short-word Memory) architecture, the word "Selective Write" relates to a procedure when the LSTM selectively updates a memory cell with fresh data while taking the input's applicability and the situation into account.
- It is crucial to make clear that the input and forget gates together form the fundamental component of the LSTM that is responsible for memory updates in order to give a more thorough explanation.
- The LSTM can store significant information over long sequences because of these gates, which regulate the flow of information into and out of the memory cell.
- **Input Gate :** The input gate (often represented as i_t) decides how much new data, represented by candidate cell C_tilde, should be added to the already-existing memory cell C_t-1. Based on the most recent input and the previous hidden state, it is calculated.
- **Forget Gate :** The forget gate, which is often represented by the symbol f_t, determines how much of the previous memory cell (C_t-1), should be kept and used in the current time step.
- The LSTM may selectively write fresh information into its memory cell while taking into account the relevance of the input at each time step by utilizing the input and forget gates.
- The gates allow the model to maintain or update particular information over extended sequences while also helping to avoid the vanishing gradient problem.

In conclusion, selective write in LSTM refers to the process of carefully updating the memory cell using the input and forget gates, which enables the LSTM to learn long-term dependencies in sequential data and retain pertinent information. This feature is a major factor in the LSTM's success in a variety of tasks involving sequential data, including time series prediction, speech recognition, and natural language processing.

5.11 SELECTIVE FORGET IN LSTM

5.11.1 Selective Forget

GQ. Explain selective forget in LSTM in detail with help of a code. **(8 Marks)**

The "forget gate" is a device used by LSTM to regulate what data is remembered or erased from the preceding memory cell. This enables the LSTM to selectively refresh its memory and preserve crucial data across lengthy runs.

In this code sample below, the forget gate (f_t) computation is used to implement an LSTM cell. The forget gate determines how much of the old candidate cell (C_tilde) should be kept and added to the new memory cell (C_t-1). For each memory cell component, the forget gate value lies between 0 and 1.

In order to effectively capture long-term dependencies in sequential input, LSTMs need to be able to selectively update their memory cells, which is made possible by the forget gate.

Code Example of LSTM Cell with Forget Gate

```
import numpy as np
# Define the LSTM cell with forget gate
def lstm_cell_with_forget(x_t, h_prev, C_prev, W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o,
W_xc, W_hc, b_c):
  # Calculate candidate cell
  C_tilde = np.tanh(np.dot(W_xc, x_t) + np.dot(W_hc, h_prev) + b_c)

  # Calculate input gate
  i_t = sigmoid(np.dot(W_xi, x_t) + np.dot(W_hi, h_prev) + b_i)

  # Calculate forget gate
  f_t = sigmoid(np.dot(W_xf, x_t) + np.dot(W_hf, h_prev) + b_f)

  # Update memory cell
  C_t = f_t * C_prev + i_t * C_tilde

  # Calculate output gate
  o_t = sigmoid(np.dot(W_xo, x_t) + np.dot(W_ho, h_prev) + b_o)

  # Calculate hidden state
  h_t = o_t * np.tanh(C_t)
```

```
    return h_t, C_t

# Helper function for sigmoid activation
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Example usage
input_dim = 3
hidden_dim = 2

x_t = np.random.randn(input_dim, 1)
h_prev = np.random.randn(hidden_dim, 1)
C_prev = np.random.randn(hidden_dim, 1)

# Random weight matrices and bias vectors
W_xi, W_hi, b_i = np.random.randn(hidden_dim, input_dim), np.random.randn(hidden_dim,
hidden_dim), np.random.randn(hidden_dim, 1)
W_xf, W_hf, b_f = np.random.randn(hidden_dim, input_dim), np.random.randn(hidden_dim,
hidden_dim), np.random.randn(hidden_dim, 1)
W_xo, W_ho, b_o = np.random.randn(hidden_dim, input_dim), np.random.randn(hidden_dim,
hidden_dim), np.random.randn(hidden_dim, 1)
W_xc, W_hc, b_c = np.random.randn(hidden_dim, input_dim), np.random.randn(hidden_dim,
hidden_dim), np.random.randn(hidden_dim, 1)

h_t, C_t = lstm_cell_with_forget(x_t, h_prev, C_prev, W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho,
b_o, W_xc, W_hc, b_c)
print("Hidden State (h_t):")
print(h_t)
print("\nMemory Cell (C_t):")
print(C_t)
```

5.11.2 Attention-based LSTM

GQ. Explain attention-based LSTM in detail. **(8 Marks)**

Long Short-Term Memory (LSTM) extensions that integrate an attention mechanism are referred to as attention-based LSTMs. The model may selectively focus on various input sequence segments while making predictions or producing output thanks to the attention mechanism. The LSTM performs better on tasks involving sequential data, such as machine translation, language modeling, and sentiment analysis, thanks to this attention mechanism, which also enhances the LSTM's capacity to manage long-range dependencies.

Key Concepts of Attention-based LSTM

1. **Context Vector :** The context vector in an attention-based LSTM represents the weighted sum of the input sequence items. The context vector's importance in creating the attention vector is determined by the attention weights, which are computed by the attention mechanism. The context vector gives the LSTM more details, enabling it to concentrate on important segments of the input sequence.
2. **Attention Scores :** The importance of each input element to the LSTM's current hidden state is quantified by the attention scores. Typically, a compatibility function is used to determine these scores by contrasting the hidden state with each input element. Higher attention scores denote more relevant content.
3. **Softmax Function :** The attention scores are frequently standardized using the softmax function in order to obtain the attention weights. The attention weights must add up to 1, which is an acceptable probability distribution, according to the softmax function.
4. **Context-Dependent Weights :** Based on the current hidden state, the attention mechanism dynamically modifies the attention weights for each time step. This enables the LSTM to give various input items varied weights for every time step.

Equations for Attention-based LSTM

1. **Attention Scores (e_t) Calculation**

$$e_t = \text{score}(h_t, x_t) = v_a * \tanh(W_a * h_t + U_a * x_t)$$

where:

h_t : Hidden state at time step t.

x_t : Input at time step t.

W_a, U_a : Weight matrices for the hidden state and input, respectively.

v_a : A learnable weight vector.

Attention Weights (a_t) Calculation:

$$a_t = \text{softmax}(e_t)$$

2. **Context Vector (c_t) Calculation**

$$c_t = \Sigma(a_t * x_t)$$

3. **Combined Input (z_t) Calculation**

$$z_t = [h_t; c_t]$$

4. LSTM Update Equations

The regular LSTM still uses the same LSTM update equations for the input gate (i_t), forget gate (f_t), candidate cell (C_tilde), output gate (o_t), memory cell (C_t), and hidden state (h_t).

GQ. Write a code for attention-based LSTM in detail. **(6 Marks)**

Code for Attention based LSTM in tensorflow is as follows:

```
import tensorflow as tf

class AttentionLSTMCell(tf.keras.layers.Layer):
    def __init__(self, units):
        super(AttentionLSTMCell, self).__init__()
        self.units = units
        self.attention = tf.keras.layers.Attention()

        self.lstm_cell = tf.keras.layers.LSTMCell(units)

    def call(self, inputs, states):
        h_prev, C_prev = states

        # Perform attention mechanism
    context, attention_weights = self.attention([h_prev, inputs])

        # Concatenate attention context with the current input
        lstm_input = tf.concat([inputs, context], axis=-1)

        # LSTM cell update
        h_t, C_t = self.lstm_cell(lstm_input, states)

        return h_t, [h_t, C_t]

# Example usage
input_dim = 10
hidden_dim = 16

# Create the Attention-based LSTM cell
attention_lstm_cell = AttentionLSTMCell(hidden_dim)
```

```
# Dummy input and initial states
inputs = tf.random.normal(shape=(1, 5, input_dim))
initial_hidden_state = tf.random.normal(shape=(1, hidden_dim))
initial_memory_cell = tf.random.normal(shape=(1, hidden_dim))

# Run the Attention-based LSTM cell
outputs, states = tf.keras.layers.RNN(attention_lstm_cell)(inputs, initial_state=[initial_hidden_state,
initial_memory_cell])
```

In this code sample, we use TensorFlow to define an Attention-based LSTM cell. The attention mechanism and LSTM update equations are implemented by the AttentionLSTMCell class. The hidden state and input sequence elements are combined via the attention mechanism, and the LSTM cell then processes the concatenated input to create the updated hidden state and memory cell.

5.12 GATED RECURRENT UNIT (GRU)

5.12.1 Gated Recurrent Unit (GRU)

GQ. Explain Gated Recurrent Unit (GRU) in detail. **(8 Marks)**

A modification of the conventional Long Short-Term Memory (LSTM) architecture known as the Gated Recurrent Unit (GRU) was created in order to solve some of the LSTM's complications while preserving its capacity to identify long-term dependencies in sequential data. In order to teach the recurrent unit when to update its internal state and when to forget specific pieces of information, the GRU introduces gating mechanisms that regulate the flow of information within the recurrent unit. Compared to LSTM, this design makes the architecture simpler and more computationally efficient.

Key Components of Gated Recurrent Unit (GRU)

- **Update Gate (z_t) :** The update gate determines how much of the previous hidden state to retain (1 - forget) and how much of the new candidate activation to consider (update).
- **Reset Gate (r_t) :** The reset gate determines how much of the previous hidden state should be forgotten (reset) to account for the new input.
- **Candidate Activation (h_tilde_t) :** The candidate activation represents the new information that can be added to the hidden state.

Equations for Gated Recurrent Unit (GRU)

The equations for a single time step t in a GRU are as follows :

Reset Gate (r_t) Calculation

r_t = sigmoid(W_{xr} * x_t + W_{hr} * h_{t-1} + b_r)

Update Gate (z_t) Calculation

z_t = sigmoid(W_{xz} * x_t + W_{hz} * h_{t – 1} + b_z)

andidate Activation (h_tilde_t) Calculation

h_tilde_t = tanh(W_{xh} * x_t + r_t * (W_{hh} * h_{t - 1}) + b_h)

Hidden State (h_t) Calculation:

h_t = (1 - z_t) * h_{t – 1} + z_t * h_tilde_t

where : x_t : Input at time step t.

h_{t – 1} : Hidden state at the previous time step.

W_{xr}, W_{hr}, W_{xz}, W_{hz}, W_{xh}, W_{hh}: Weight matrices for different inputs and the hidden state.

b_r, b_z, b_h: Bias vectors for the reset gate, update gate, and candidate activation, respectively.

☞ Advantages of Gated Recurrent Unit (GRU) over LSTM

GQ. List advantages and use cases of GRU. **(6 Marks)**

The GRU offers several advantages over LSTM :

1. **Simpler Architecture :** GRU's simpler architecture, which has fewer gating mechanisms than LSTM, makes it easier to use and more efficient computationally.
2. **Faster Training :** GRU can frequently be trained more quickly than LSTM due to its simpler design, especially on smaller datasets.
3. **Less Prone to Overfitting :** GRU may be less susceptible to overfitting because of its smaller number of parameters, especially when training data is scarce.
4. **Similar Performance to LSTM :** GRU frequently exhibits similar performance to LSTM, delivering equivalent outcomes on a range of tasks involving sequential data.

Use Cases for Gated Recurrent Unit (GRU)

Applications including natural language processing, speech recognition, machine translation, sentiment analysis, and time series prediction frequently make use of GRUs. They are especially helpful for jobs that require capturing long-term dependencies in sequential data while maintaining computational efficiency, such as dealing with medium-sized datasets. However, it's crucial to experiment and select the best architecture based on the details of the given task and dataset.

Chapter Ends...

□□□

Module 6

CHAPTER 6

Recent Trends and Applications

Syllabus

Generative Adversarial Network (GAN): Architecture
Applications: Image Generation, DeepFake

6.1 GENERATIVE ADVERSARIAL NETWORK (GAN): ARCHITECTURE

Deep Generative Models

- A generative model is a powerful way of learning any kind of data distribution using unsupervised learning. It has achieved tremendous success in last few years.
- All types of generative models aim at learning true data distribution of the training set and then they generate new data points with some variations.
- But it is not always possible to learn the exact distribution of our data either implicitly or explicitly, so we try to model a distribution which is as similar as possible to the true data distribution.
- For this, we use the power of neural networks to learn a functions which can approximate the model distribution to the true distribution.
- Two of the most commonly used and efficient approaches are Variational Autoencoders (VAE) and Generative Adversarial Networks (GAN).
- VAE aims at maximizing the lower bound of the data log-likelihood and GAN aims at achieving an equilibrium between Generator and Discriminator.

6.1.1 Generative Adversarial Networks

GQ. Explain different type of Generative adversarial models.

- Generative Adversarial Networks don't work with any explicit density estimation, but it is based on game theory approach with an objective to find Nash equilibrium, between the two networks, Generator and Discriminator.
- The idea is to sample from a simple distribution like Gaussian and then learn to transform this noise to data distribution using universal function approximators such as neural networks.
- This is achieved by adversarial training of these two networks. A generator model G learns to capture the data distribution and a discriminator model D estimates the probability that a sample came from the data distribution rather than model distribution.
- In this game, the generator tries to fool the discriminator by generating real images as far as possible and the discriminator tries not to get fooled by the generator by improving its discriminative capability.

6.1.2 Generative Adversarial Networks (GANs) can be Broken Down into Three Parts

(i) **Generative :** To learn a generative model, which describes how data is generated in terms of a probabilistic model.

(ii) **Adversarial :** The training of a model is done in an adversarial setting.

(iii) **Networks :** Use deep neural networks as the artificial intelligent (AI) algorithms for training purpose.

We mention the image, showing the basic architecture of GAN.

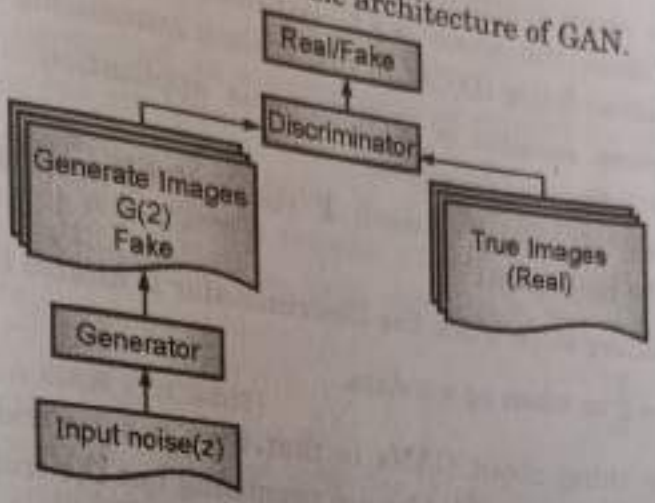


Fig. 6.1.1 : Building block of Generative Adversarial Network

We define a prior on input noise variables p(z) and then the generator maps this to data distribution using a complex differentiable function with parameters θg.

In addition to this, we have another network called Discriminator which takes in input x and using another differentiable function with parameters θd outputs a single scalar value denoting the probability that x comes from the true data distribution P data(x). The objective function of the GAN is defined as :

$$\min_G \max_D V(D, G) = E_{x \sim p\,data\,(x)} [\log D(x)] + E_{z \sim p_z(z)} [\log (1 - D(G(z)))]$$

Where,

G = Generator

D = Discriminator,

P data (x) = distribution of real data

p (z) = distribution of generator

x = sample from p data (x)

Z = sample from p (z)

D (x) = Discriminator network

G (z) = Generator network

- In the above equations, if the input to the Discriminator comes from true data distribution then D(x) should output 1 to maximize the above objective function w.r.t.

D, where as if the image has been generated from the Generator then D (G (z)) should output 1 to minimize the objective function w.r.t. G.

- The latter implies that G should generate such realistic images which can fool D.
- We maximize the above function w.r.t. parameters of Discriminator using Gradient Ascent and minimize the same w.r.t. parameters of Generator using Gradient Descent. We maximize E [log (D(G(z)))] rather than minimising E [log (1 – D(G(z)))].
- The training process consists of simultaneous application of stochastic Gradient descent on discriminator and generator.
- While training, we alternate between k steps of optimizing D and one step of optimizing G on the mini-batch.
- The process of training stops when the Discriminator is unable to distinguish eg and ρ data i.e. $D(x, \theta d) = \frac{1}{2}$ or when eg = ρ data.
- One of the simple thing about GANs is that they can be trained even with small training data. But the results of GANs are promising but the training procedure is not trivial especially setting up the hyper parameters of the network.
- Also, GANs are difficult to optimize as they don't converge easily.

Remarks

(i) Here we saw why the Adversarial networks are better at producing realistic images.

(ii) But there are problems with GANs such as stabilizing their training which is an active area of research.

(iii) But GANs are really powerful and they are used in a variety of tasks such as high quality image and video-generation, text to image translation, image enhancement, reconstruction of 3D models of objects from images, music generation, cancer drug discovery etc.

(iv) Generative models are going to be very helpful for graphic designing, designing of attractive user-Interface etc.

6.1.3 Different Types of GANs

- GANs are a very active topic of research and there are many different type of GAN implementation.
- Some of the important ones that are actively being used are described below :

(i) Vanilla GAN

- This is the simplest type of GAN. Here, the Generator and Discriminator are simple multi-layer perceptrons.
- In vanilla GAN, the algorithm tries to optimize the mathematical equation using stochastic gradient descent.

- Vanilla GANs has two networks called generator network and a discriminator network. Both the networks are trained at the same time and compete or battle against each-other in a minimax play.

(iii) Conditional GAN (CGAN)

- CGAN can be described as a deep learning method in which some conditional parameters are put into place.
- In CGAN, an additional parameter 'y' is added to the Generator for generating the corresponding data. Labels are also put into the input to the Discriminator in order for the Discriminator to help distinguish the real data from the fake generated data.

(iii) Deep Convolution GAN (DCGAN)

- DCGAN is one of the most popular and hence also the most successful implementation of GAN.
- It is composed of convNets in place of multi-layer perceptrons. The convNets are implemented without max-pooling, which is in fact replaced by convolution stride. Also, the layers are not fully connected.

(iv) Laplacian Pyramid GAN (LAPGAN)

- The Laplacian pyramid is a linear invertible image representation consisting of a set of band-pass images, spaced an octave apart, plus a low-frequency residual.
- This approach uses multiple number of Generator and Discriminators networks and different levels of Laplacian pyramid. This image is mainly used because it produces very high quality images.
- The image is down-sampled at first at each layer of the pyramid and then it is again up-scaled at each layer in a backward pass where the image requires some noise from the conditional GAN at these layers until it reaches its original size.

(v) Super Resolution GAN (SRGAN)

- SRGAN as the name suggests is a way of designing a GAN in which a deep neural network is used along with an adversarial network in order to produce higher resolution images.
- This type of GAN is particularly useful in optimally up-scaling native low-resolution images to enhance its details minimizing errors while doing so.

(vi) Cycle GAN

- Training a model for image-to-image translation typically requires a large dataset of paired examples.
- These datasets can be difficult and expensive to prepare, and in some cases impossible, such as photographs of paintings by long dead artists.
- The cycle GAN is a technique that involves the automatic training of image-to-image translation models without paired examples.
- The models are trained in an unsupervised manner using a collection of images from the source and target domain that do not need to be related in any way.
- This simple technique achieves visually impressive results on a range of application domains, most notably translating photographs of horses to zebra, and the reverse. Thus we observe that
  (a) Image-to-image translation involves the controlled modification of an image and requires large datasets of paired images that are complex to prepare or sometimes don't exist.
  (b) Cycle GAN is a technique for training unsupervised image translation models via GAN architecture using unpaired collections of images from two different domains.
  (c) Cycle CAN has been demonstrated on a range of applications including season-translation, object transfiguration, style transfer, and generating photos from paintings.

6.1.3.1 Cycle GAN Model Architecture

- The architecture of the cycle GAN appears to be complex.
- Consider the problem where we want to translate images from summer to winter and winter to summer.
- We have two collections of photographs and they are unpaired, meaning they are photos of different locations at different times,

  Collections 1 : Photos of summer landscapes,

  Collection2 : Photos of winter landscapes.
- We develop an architecture of two GANs, and each GAN has a discriminator and generator model, i.e. there are four models in total in the architecture.
- The first GAN will generate photos of winter given photos of summer, and the second GAN will generate photos of summer given photos of winter.

  GAN 1 : Translate photos of summer (collection 1) to winter (collection 2).

  GAN 2 : Translate photos of winter (collection 2) to summer (collection 1).
- Each GAN has a conditional generator model that will synthesize an image given an input image. And each GAN has a discriminator model to predict how likely the generated image is supposed to be from the target image collection.

- The discriminator and generator models for a GAN are trained under normal adversarial loss like a standard GAN model.

We summarise the generator and discriminator models from GAN1 as follows :

Generator Model 1

(i) Input : Takes photos of summer (collection 1).

(ii) output : Generates photos of winter (collection 2).

Discriminator model 1

(i) Input : Takes photos of winter from collection 2 and output from Generator model 1

(ii) output : likelihood of image is from collection 2.

Similarly form GAN 2 as follows

Generator Model 2

(i) Input : Takes photos of winter (collections 2).

(ii) Output : Generates photos of summer.

Discriminator model 2

(i) Input : Takes photos of summer from collection 1 and output from Generator Model 1

(ii) Output : likelihood of image is from collection 1.

- Each of the GANs are also updated using cycle consistency loss.
- Cycle consistency loss compares an input photo to the cycle GAN to the generated photo and calculate the difference between the two e.g. using the 'L1 norm' or summed absolute difference in pixel values.
- There are two ways in which cycle consistency loss is calculated and used to update the generator models each training iteration.
- The first GAN (GAN 1) will take an image of a summer landscape, generate image of a winter landscape, which is provided as input to the second GAN (GAN 1), which in turn will generate an image of a summer landscape.
- The cycle consistency loss calculates the difference between the image input to GAN 1 and the image output by GAN2 and the generator models are updated accordingly to reduce the difference in the images.
- This is a forward-cycle for cycle consistency loss. The same process is related in reverse for a backward cycle consistency loss from generator 2 to generator 1 and comparing the original photo of winter to the generated photo of winter.

Forward cycle consistency loss

(i) Input photo of summer (collection 1) to GAN 1.

(ii) Output photo of winter from GAN1

(iii) Input photo of winter from GAN 1 to GAN 2.

(iv) Compare photo of summer (collection 1) to photo of summer from GAN 2.

Backward Cycle Consistency Loss

(i) Input photo of winter (collection 2) to GAN 2.

(ii) Output photo of summer from GAN 2.

(iii) Input photo of summer from GAN 2 to CAN 1.

(iv) Output photo of winter from GAN 1.

(v) Compare photo of winter (collection 2) to photo of winter from GAN 1.

6.1.4 Applications of Cycle GAN

The cycle GAN approach is presented with many impressive applications.

(i) **Style Transfer :** Style Transfer refers to the learning of artistic style from one domain, often paintings, and applying the artistic style to another domain, such as photographs.

(ii) **Object Transfiguration :** Object transfiguration refers to the transformation of objects from one class, such as dogs into another class of objects, such as cats.

- The cycle GAN transforms photographs of horses into zebras and the reverse photographs of zebras into horses.
- This makes sense if both horse and zebra look similar in size and structure, except for their coloring.

(iii) **Season Transfer**

- Season transfer refers to the transformation of photographs taken in one season, such as summer, to another season, such as winter.
- The cycle GAN demonstrated on translating photographs, of winter landscapes to summer landscapes and the reverse.

Style GAN

- Style GAN is a Generative Adversarial Network (GAN), and uses an alternative architecture for generative adversarial networks, borrowing from style transfer literature, (in particular, the use of adaptive instance nor) malisation).
- Style GAN is 'an open-source' for experimenting or just to try out this machine learning code. Style GAN depends upon NVIDIA's CUDA software and GPUs as well

6.2 APPLICATIONS : IMAGE GENERATION, DEEP FAKE

Q. What is image classification and why is it important? How does image classification work?

- Image Classification is one of the most fundamental tasks in computer vision. Image classification has revolutionized and propelled technological advancements in the AI field, from the automobile industry to medical analysis and automated perception in robots.
- Image classification (or Image recognition) is a subdomain of computer vision in which an algorithm looks at an image and assigns it a tag from a collection of predefined tags or categories that it has been trained on.
- Vision is responsible for 80-85 percent of our perception of the world, and we, as human beings, trivially perform classification daily on whatever data we come across. Therefore, emulating a classification task with the help of neural networks is one of the first uses of computer vision that researchers thought about.
- A computer visualizes an image in the form of pixels. In its view, the image is just an array of matrices, with the size of the matrix dependent on the image resolution. Image processing for the computer is thus the analysis of this mathematical data with the help of algorithms.
- The algorithms break down the image into a set of its most prominent features, reducing the workload on the final classifier. These features give the classifier an idea of what the image represents and what class it might be put into.
- The feature extraction process forms the most crucial step in classifying an image as all further steps depend on it. Classification, particularly supervised classification, also depends largely on the data fed to the algorithm. A well-balanced classification dataset works wonders as compared to a bad dataset with class-wise data imbalance and poor quality of images and annotations.
  - Image classification is a subdomain of computer vision dealing with categorizing and labeling groups of pixels or vectors within an image using a collection of predefined tags or categories that an algorithm has been trained on.
  - We can distinguish between supervised and unsupervised classification.
  - In supervised classification, the classification algorithm is trained on a set of images along with their corresponding labels.
  - In unsupervised classification, the algorithm uses only raw data for training.
  - To build reliable image classifiers you need enough diverse datasets with accurately labelled data.
  - Image classification with CNN works by sliding a kernel or a filter across the input image to capture relevant details in the form of features.

- Other machine learning image classification algorithms include K-Nearest Neighbors, Support Vector Machines, and Random Forests.
- The most important image classification metrics include Precision, Recall, and F1 Score.

- In order to classify a set of data into different classes or categories, the relationship between the data and the classes into which they are classified must be well understood.
- Generally, classification is done by a computer, so, to achieve classification by a computer, the computer must be trained. Sometimes it never gets sufficient accuracy with the results obtained, so training is a key to the success of classification.
- To improve the classification accuracy, inspired by the ImageNet challenge, the proposed work considers classification of multiple images into the different categories (classes) with more accuracy in classification, reduction in cost and in a shorter time by applying parallelism using a deep neural network model.
- The image classification problem requires determining the category (class) that an image belongs to. The problem is considerably complicated by the growth of categories' count, if several objects of different classes are present in the image and if the semantic class hierarchy is of interest, because an image can belong to several categories simultaneously.
- Fuzzy classes present another difficulty for probabilistic categories' assignment. Moreover, a combination of different classification approaches has shown to be helpful for the improvement of classification accuracy.
- Deep convolutional neural networks provide better results than existing methods in the literature due to advantages such as processing by extracting hidden features, allowing parallel processing and real time operation.
- The concept of convolutions in the context of neural networks begins with the idea of layers consisting of neurons with a local receptive field, i.e., neurons which connect to a limited region of the input data and not the whole
- Image processing is now routinely used by a wide range of individuals who have access to digital cameras and computers. With a minimum investment, one can readily enhance contrast, detect edges, quantify intensity, and apply a variety of mathematical operations to images.
- Although these techniques can be extremely powerful, the average user often digitally manipulates images with abandon, seldom understanding the most basic principles behind the simplest image-processing routines.
- Although this may be acceptable to some individuals, it often leads to an image that is significantly degraded and does not achieve the results that would be possible with some knowledge of the basic operations of an image-processing system.

- Classification between the objects is an easy task for humans but it has proved to be a complex problem for machines. The rise of high-capacity computers, the availability of high quality and low-priced video cameras, and the increasing need for automatic video analysis has generated an interest in object classification algorithms.
- A simple classification system consists of a camera fixed high above the zone of interest, where images are captured and consequently processed. Classification includes image sensors, image pre-processing, object detection, object segmentation, feature extraction and object classification.
- A classification system consists of a database that contains predefined patterns which are compared with a detected object to classify it to a proper category.
- Image classification is an important and challenging task in various application domains, including biomedical imaging, biometry, video surveillance, vehicle navigation, industrial visual inspection, robot navigation, and remote sensing.

The classification process consists of the following steps :

A. **Pre-processing :** Atmospheric correction, noise removal, image transformation, main component analysis, etc.;

B. Detection and extraction of an object, including detection of position and other characteristics of a moving object image obtained by a camera; while inextraction, estimating the trajectory of the detected object in the image plane;

C. **Training :** Selection of the particular attribute which best describes the pattern;

D. **Classification of the object :** this step categorizes detected objects into predefined classes by using a suitable method that compares the image patterns with the target patterns.

6.2.1 Image Classification Approach

I. **Digital data :** An image is captured by using a digital camera or any mobile phone camera.

II. **Pre-processing :** Improvement of the image data, which includes obtaining a normalized image, enhancing contrast, obtaining a gray-scale image, binary image, resizing image, complementing binary image, removing noise, getting the image boundary.

III. **Feature extraction :** The process of measuring, calculating or detecting the features from the image samples. The two most common types of feature extraction are (i) geometric feature extraction and (ii) color feature extraction.

IV. **Selection of preparing information :** Selection of the specific property which best portrays the given example, e.g., info image, output image, after preprocessing train dataset name.

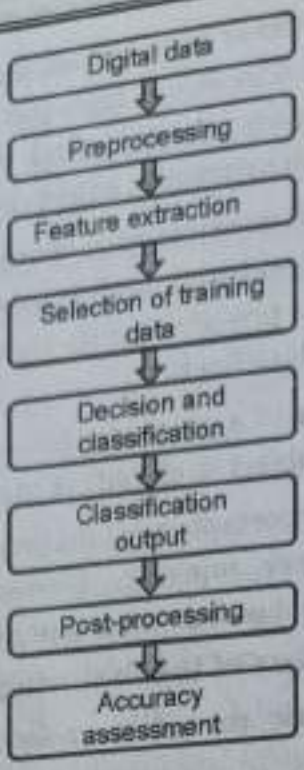


Fig. 6.2.1 : Steps for image classification

V. **Decision and classification :** Categorizes recognized items into predefined classes by utilizing a reasonable strategy that contrasts the image designs and the objective examples.

VI. **Accuracy evaluation :** Precision appraisal is acknowledged to distinguish conceivable well springs of mistakes and as a pointer utilized as a part of correlations. See Fig. 6.2.1.

6.2.2 Implementation

GQ. How deep learning can be used for image classification?

Deep Convolutional Neural Networks

- **Deep learning :** Deep learning is a new area of machine learning research, which has been presented with the goal of getting machine learning nearer to one of its unique objectives. Deep learning is an artificial intelligence function that copies the task of the human brain and is used in processing data and creating patterns for decision making.
- Deep learning for images is simply using more attributes extracted from the image rather than only its signature. However, it is done automatically in the hidden layers and not as input (as it is the case in NN), as can see in Fig. 6.2.2. The neurons in the first layer pass input data to the network. Similarly, the last layer is called the output layer. The layers in-between the input and output layers are called hidden layers. In this example, the network has only one hidden layer shown in blue.

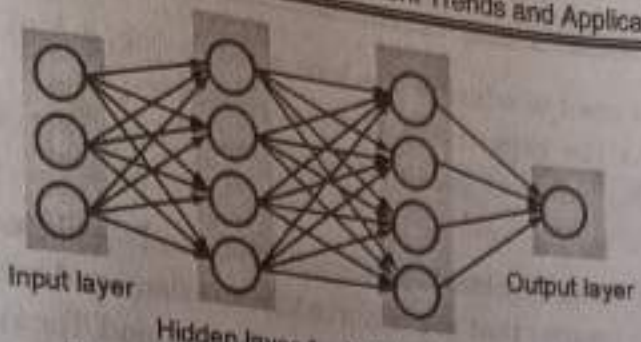


Fig. 6.2.2 : Neural network representation

- The networks which have many hidden layers tend to be more accurate and are called deep networks; hence machine learning algorithms which use these deep networks are called deep learning.
- A typical convolutional network is a sequence of convolution and pooling pairs, followed by a few fully connected layers. A convolution is like a small neural network that is applied repeatedly, once at each location on its input. As a result, the network layers become much smaller but increase in depth. Pooling is the operation that usually decreases the size of the input image. Max pooling is the most common pooling algorithm, and has proven to be effective in many computers vision tasks.
- Suppose we try to teach a computer to recognize images and classify them into one of these 10 categories; see Fig. 6.2.3.

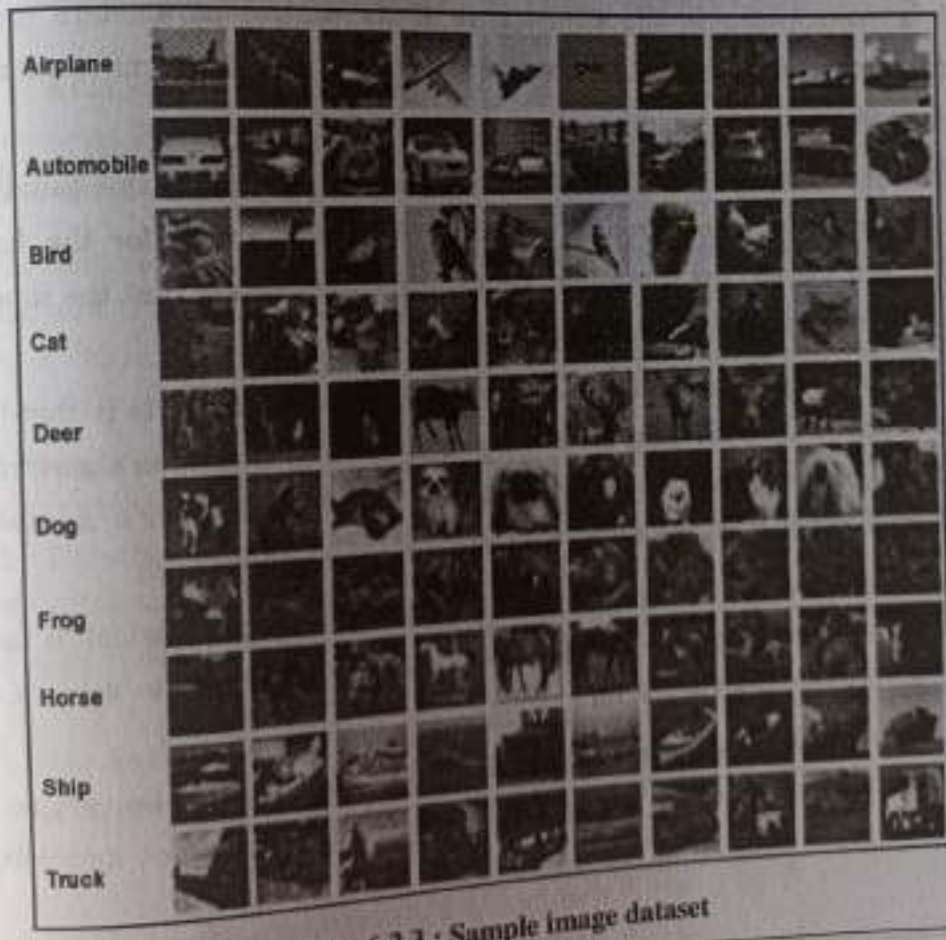


Fig. 6.2.3 : Sample image dataset

- To do so, we first need to educate the how a cat, a dog, a bird, etc., looks like before the computer has the capacity to perceive another picture. The more felines the computer sees, the better it gets at perceiving felines. This is known as regulated learning.
- It can convey this errand by marking the images, the PC will begin perceiving designs exhibited in feline images that are different from others and will begin assembling its own particular discernment. One can utilize Python and TensorFlow to compose the program.
- TensorFlow is an open source, profound learning system made by Google that gives engineers granular control over every neuron (known as a "hub" in TensorFlow) so it can alter the weights and accomplish ideal execution.
- TensorFlow has numerous developed libraries (a few of which shall be utilized for image ordering) and has an astonishing network, so only needs to discover open source usage for basically any profound learning point.

6.3 TUNING OF CONTROL ALGORITHMS

- Tuning a control loop is the adjustment of its control parameters (proportional band/gain, integral gain/reset, derivative gain/rate) to the optimum values for the desired control response.
- A tuning system for the process control loop, the tuning system fine-tuning the field device controller and the process controller by determining, for the field device controller and the process controller, such control parameters that the interaction of the controllers provides desired process variability.
- The objective of algorithm tuning is to find the best point or points in that hypercube for your problem. One can then use those points in an optimization algorithm to zoom in on the best performance. One can repeat this process with a number of well performing methods and explore the best one can achieve with each.
- PID controller is one of the most popular closed-loop controllers which is used in the automation industry. By fine-tuning 3 constants, you are able to achieve a system which is almost free from any errors.
- On the other hand, Genetic Algorithms are being heavily used by various machine learning enthusiasts around the world for building an efficient and robust deep learning network.

6.3.1 What's a Control System ?

GQ. What are control systems?

- In simple terms, control system takes some sets of inputs, regulates them to derive the desired output and then directs them.
- Control system is usually of two types :
- (1) Open-loop and (2) Closed-loop.
- The only difference being that, in closed-loop, the error is sent into the controller as a feedback signal. In this article, we'll be focusing on closed-loop control systems.

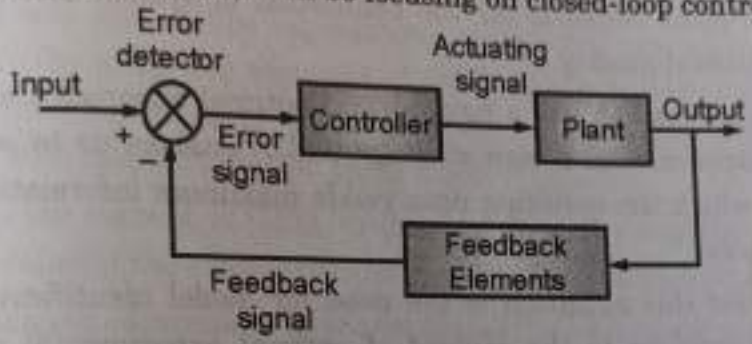


Fig. 6.3.1 : Skeleton of a closed-loop control system

6.3.2 What is a PID Controller ?

GQ. What is a PID controller?

- In a PID controller, we calculate an error $e(t)$ as the difference between the desired set-point and the current value (process variable) and pass it as a feedback signal. The error $e(t)$ is then corrected based on the proportional (p), integral (i) and derivative (d) terms.
- In the chemical industry, PI controllers are widely used because of their simplicity and robust structure. However, it is still difficult to find good controller settings even though PI controllers only have a few adjustable parameters. There are basically two different methods for the calculation of controller settings.
- One of them is based on open-loop experiments whereas the other one uses closed-loop tests.
- Most tuning methods are based on information about process dynamics obtained from open-loop experiments. However, this strategy is usually time consuming and expensive.

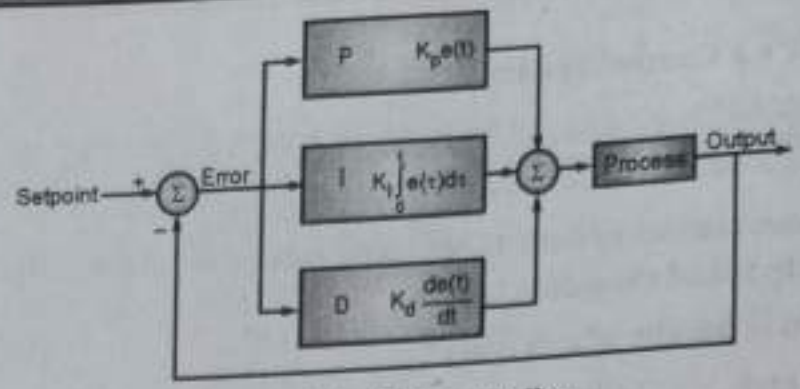


Fig. 6.3.2 : PID controller

- Therefore, there is a scope for the determination of experimental conditions using closed-loop set-point response for the estimation of system model parameters with maximum statistical quality.
- For this purpose, one can use a model-based optimal experimental design approach. The optimal experimental design strategy (OED) enables us to select experimental conditions for which the resulting data yields maximum information with respect to model parameters.
- The challenge for this approach is the need for model identification by closed-loop experiments. According to the concept of optimal experimental design, parameter estimation is applied in order to match the model to areal process.
- Therefore, the calculation of controller settings consists of two steps.
- First, to identify the system behavior using closed-loop set-point response and after that, to calculate the controller settings using an optimization strategy.

6.3.3 System Identification Strategy

- Controller tuning requires a good process model. Therefore, the controller tuning strategy consists of two steps (Fig. 6.3.3). In the first step, we identify the process behaviour experimentally by using closed-loop set-point response. After this, based on the results achieved in the first step, we numerically calculate the controller settings.
- We assume that each process can be described by a first or second order with time delay model. That means that we have three or four model parameters that have to be determined. These are the process gain K, the time constants T_1, T_2 and the time delay, τ.
- Usually, an open-loop experiment is used to identify the process behavior. Here an experiment is executed with just a P-controller. By this method, we are able to reduce the experimental time and decrease the number of necessary experiments as well as the required amount of measured data.

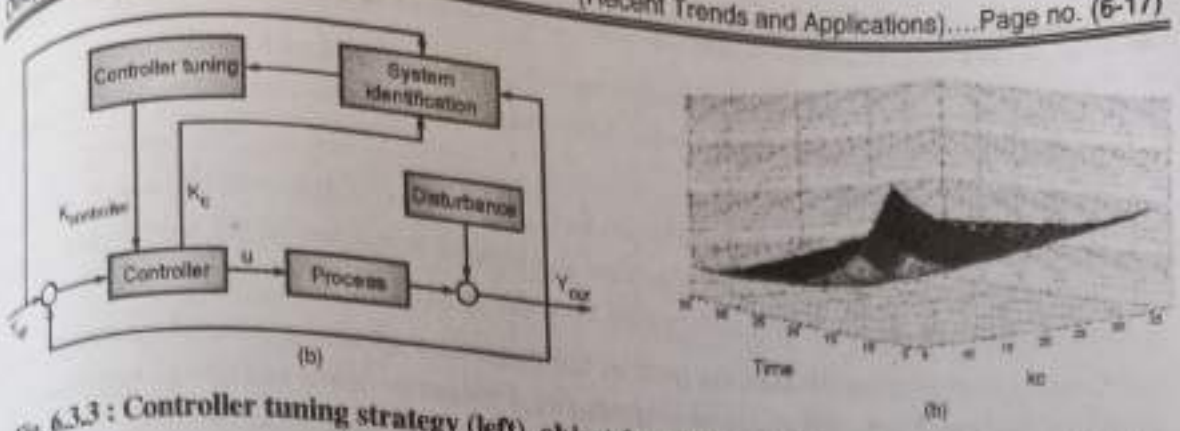


Fig. 6.3.3 : Controller tuning strategy (left), object function for OED over the design variable k_c for different measurement times (right)

- For a closed-loop system the experimental conditions are represented by controller gain ***kc***. Here, by using the approach of OED we aim to determine the optimal value of the controller gain $k_{c,opt}$ for which the experimental data yields maximum information with respect to model parameters.
- According to the concept of OED, in the first step, the experiment is planned. Here, the optimal value of the control gain throughout the experiment is determined.
- Next, the planned experiment is executed and the measured data is analyzed. In the last step, the quality of the model is validated and a new set of model parameters is calculated (Fig. 6.3.4).
- These three iteration steps are executed sequentially without interruption until the required model accuracy is achieved. Important is here, that the design of a new experiment is based on the current information about model structure and parameter set obtained in the prior experiments.

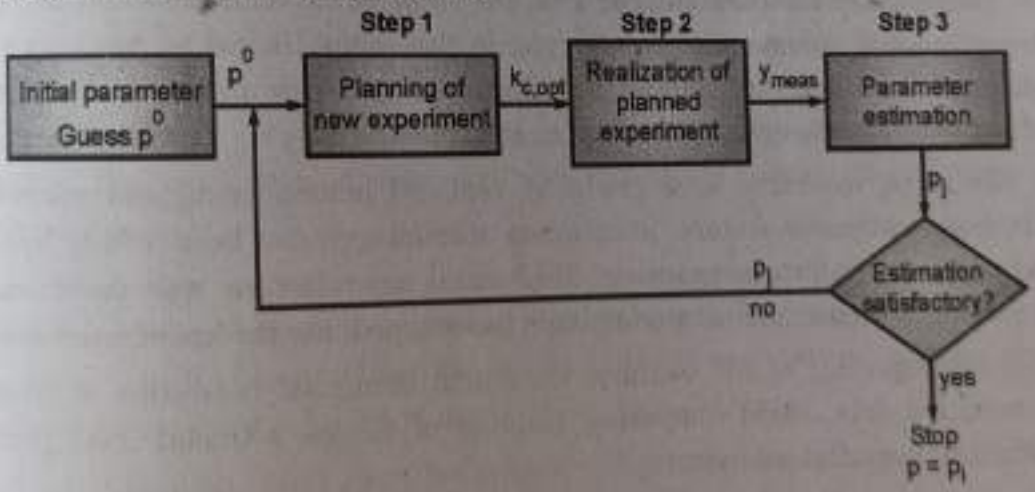


Fig. 6.3.4 : Model-based system identification strategy using closed-loop response

- The strategy consists of two steps. In the first step, we identify the process behaviour using a closed-loop set-point response. For this purpose, we first run experiments with just a P-controller.

- In the proposed procedure, the design variable for the optimal experimental design is represented by the controller gain k_c. After this, we calculate the controller settings by using an optimization strategy.
- Here, we use the model achieved in the first step for the calculation of optimal controller parameters. One can also use the second order model for calculation of the controller settings for a first order with time delay process.
- Usually, a very time-consuming part of the controller tuning procedure is the process identification. Here, using closed-loop set-point response leads to shorter experimental time compared to open-loop experiments.
- Furthermore, well designed experiments enable us to reduce the system identification effort even more. In summary, the sum of both benefits leads to reduced commissioning time in comparison to conventional methods.

6.4 FAULT DETECTION

6.4.1 Development of Mechanical Manufacture Industry

- Globally speaking, mechanical manufacture industry has experienced four stages of development. The first stage is machine manufacturing age. In the late 18th century, Industrial Revolution characterized by the invention of steam engine and machine tool brought manufacture industry into the age in which machines replaced manual manufacture.
- From the beginning of 1900s to 1960s, the second industrial revolution happened, and manufacture industry stepped into the stage of electrification and automation. Streamline and volume-produce emerged in this stage. Based on the upgrading of industry 2.0, electric information technology was applied in mechanical manufacture industry and electric information era has come.
- At this stage, machines have gradually replaced human being and micro-electric technology, computer science, automation technology have been widely applied in mechanical manufacture industry. Mechanical manufacture was developing into integration. Now mechanical manufacture has stepped into the age of intelligence.
- From the beginning of 21st century, the fourth industrial revolution is integrating internet, big data, cloud computing, internet of things, artificial intelligence into mechanical manufacture industry.
- With the progress and development of science and technology, artificial intelligence technology is being increasingly applied to mechanical manufacture and automation.
- Artificial intelligence technology builds production model through computer simulation system and makes comprehensive data analysis to make relevant precautious measures in case of emergency, which guarantees the orderly production

system, reduces the possible capital loss of manufacturing enterprises, and also greatly improves the production efficiency and accuracy of manufacturing.

- The process of mechanical design, manufacturing and automation is relatively complex, and a large amount of data calculation is required in this process.
- For example, a large number of formulas are needed in the process of modeling and demonstration to calculate and deduce, and if the process is completely dependent on manual calculation, on the one hand, it is easy to calculate wrongly, on the other hand, it also takes a lot of time and effort, which is not conducive to the entire production process.
- Luckily, artificial intelligence can automatically classify and categorize information to improve the accuracy of calculation, and therefore, the subsequent errors or failures can be effectively avoided.
- In addition, artificial intelligence can also diagnose mechanical failure. In the method of fault diagnosis based on Expert System Theory, firstly, the data being monitored by machines are input into the system through the human-machine interface.
- Then the reasoning machine obtains the corresponding diagnostic results through forward inference engine and puts forward expert opinions.
- Finally, the most similar cases in history are obtained by intelligent searching, and the similarity is calculated based on the historical cases to diagnose mechanical faults.
- This advantage of artificial intelligence technology can also be reflected in the maintenance of equipment.
- Artificial intelligence in equipment maintenance is mainly predictive maintenance, which is done by collecting the actual operation data of parts of the equipment and then comparing them with that of the intelligent training model, timely warning and reminding related personnel to maintain. This technology not only improves the safety of the production system, but also effectively reduces the downtime, and improves production efficiency as well.
- Fault detection is one of the key activities of quality assurance. Fault detection plays a vital role in thinning out the software time and price of building although, there are numerous detection techniques that are available in software engineering there's a necessity for constant software fault detection methodology.
- Machine Learning has four techniques like supervised, unsupervised, semi-supervised, and reinforcement learning are discussed.
- In line with the survey, to detect the fault, a mixture of classification and reduction machine learning techniques.

6.4.2 Machine Learning-based Scheme for Multi-class Fault Detection in Turbine Engine Disks

- Fault detection in rotating engine components is a research topic widely applied in many practical engineering areas like aviation safety, complex dynamic networks, and vehicular networks.
- Detection of faults in turbine engine components is needed in the aviation industry' s efforts to improve the working status of the turbine engine disk and ensure its efficient running whose failures may lead to catastrophic events. Particularly, engine failures can jeopardize the security of an aircraft in flight.
- Common current techniques for engine inspection are magnetic particle inspection, eddy currents, liquid penetrant testing, radiography testing and borescope inspection. The advantages of these techniques are the capability to detect normal and unusual discontinuities and the flexibility to be used on various materials such as rubbers, plastics, and metals.
- On the other hand, the drawbacks are its inspections costs, and its operating is programmed using a schedule based on a certain number of flight hours at regular intervals.
- Induction thermography is another inspection approach for aircraft engine that utilizes induction heating for faults detection inconductive materials with the aid of infrared cameras to detect the presence of temperature drops caused by flaws.
- The results showed that this technique requires a direct path from the infrared camera to the flaw to achieve good detections. Otherwise, a small diameter bolt hole can hardly be inspected.

6.4.3 Automated Fault Management with Machine Learning

- The digitization of industry and critical infrastructures, brought to life through virtualized cloud platforms, requires a new approach to fault management altogether. In this blog post, the second in our series about automated fault management, we take a good look at the latest machine learning techniques used to detect and predict faults in cloud systems.
- Based on machine learning techniques, fault detection and fault prediction functions make an integral component of a modern day automated fault management system. As we made the case in our previous post, automating fault detection for management systems using ML, machine learning techniques play an important role in automating these functions.
- In this post, we describe how different machine learning techniques can be applied in automated fault management systems to both detect faults and anomalies, and also predict faults that will eventually occur.

6.4.4 Components of a Fault Management System

The Fig. 6.4.1 captures the key functions included in a fault management system and how they relate to each other.

Basic Machine Learning Components

The two major types of machine learning - supervised and unsupervised learning - have different applications and, as such, address different aspects of problems that are faced by today's fault detection methods.

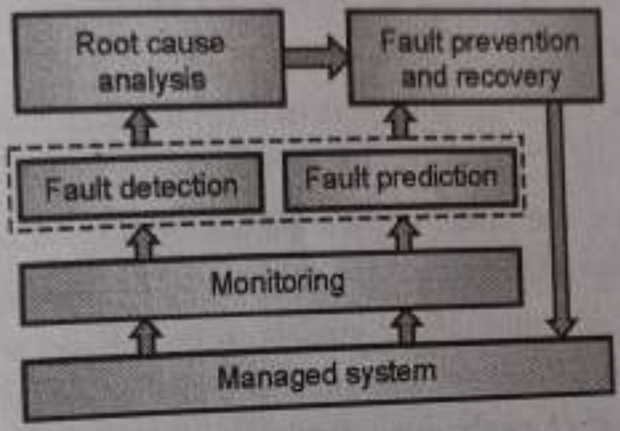


Fig. 6.4.1

- Most machine-learning-based solutions share the same set of core components: a data component that serves the data that would be used to train and evaluate the machine learning model, a preprocessing component that would adapt the data into a format usable for using with the models and the training/inference pipelines where the actual machine learning magic happens.

Monitoring

- Like most machine learning solutions, successful fault detection and prediction require a large amount of data on which to train or fit the models. Such data may already exist as historical monitoring data in most systems with functioning monitoring components.
- However, it is important to avoid the pitfalls of a human-knows-best approach in selecting metrics to monitor.
- Specifically, the goal of a monitoring systems for ML-based techniques should be to collect as many metrics as possible, as frequently as possible for as long as possible, while keeping the impact on the monitored system minimal.

Preprocessing

One or more of data preprocessing steps are applied to the data depending on the data itself and the requirement of models to be trained. The most common ones include :

- **Data synchronization** where metrics collected from/through several agents are aligned in time with each other.
- **Data cleansing** where either unnecessary data is removed (e.g. unusable metrics, such as non-numeric or ephemeral data, empty samples, etc.) and missing data is generated (e.g. imputation of missing data through interpolation).
- **Gaugification** where counter-type metrics (metrics that increase all the time) are converted to gauge-type metrics (metrics whose values can both increase and decrease), through e.g. the process of differencing.
- **Normalization** where the values of the metrics are scaled such that all metrics have comparable magnitudes (e.g. through min-max normalization and standardization).
- **Features selection** where the relevant metrics are identified for use in training the models.
- Of all the above steps, the latter is perhaps the most technical one in the sense that there are several different techniques to apply for different applications.
- In general, features selection allows models to be trained with only a subset of the metrics from the original dataset, making the training process faster and its resource requirements lower. If properly done, it also improves the accuracy of the model by removing noise from the training data.
- In our experiments, we were able to reduce the number of features of one dataset by an order of magnitude by using a process called recurrent feature elimination. For another dataset, we were able to reduce the number of features by a factor of 6 using PCA, while still maintaining a classification accuracy of more than 99% by the trained models.

6.4.5 Basic Python Implementation

We'll begin by importing Python libraries. For learning purposes, there are countless datasets available. We're going to use https://www.kaggle.com/c/digit-recognizer for this article.

```
from sklearn.model_selection import train_test – split
from dbn.tensorflow import supervisedDBNClassification
import numpy as np
import pandas as pd
form sklearn.metrics.classification import accuracy_score
```

Then we'll upload the CSV file and use the sklearn package to create a DBN model. Also, divide the test set and training set into 25% and 75% respectively. The output was then forecast and saved in y pred. Finally, we calculated the Accuracy score and displayed it on the screen.

```
digits = pd.read_csv("train.csv")
from sklearn.preprocessing import standradscaler
```

```
X = np.array(digits.drop(["label"], axis = 1))
y = np.array(digits["label"])
ss = standardScaler()
X = ss.fit – transform(X)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25)
classifier = SupervisedDBNClassification(hidden_layers_structure = [256, 256], learing_rate_rbm=0.05,
learning_rate=0.1, n_epochs_rbm = 10, n_iter_backprop=100, batch_size=32,
activation_function='relu,
dropout_p = 0.2)
classifier.fit(x – train, y_train)
y_pred = classifier.predict(x_test)
print("nAccuracy of prediction: %f' % accuracy_score(x_test, y_pred))
```

Output

```
Accuracy of prediction : 93.3%
```

Deep Generative Models (DGM)

- Deep generative models (DGM) are neural networks with many hidden layers trained to approximate complicated, high dimensional probability distributions using a large number of samples.
- When trained successfully, we can use the DGM to estimate the likelihood of each observation and to create new samples form the underlying distribution.
- The literature on DGMs has become vast and is growing rapidly. Developing DGMs has become one of the most hotly researched fields in artificial intelligence in recent years.
- Some advances have even reached the public sphere, for example, the recent successes in generating realistic looking images, voices or movies, so-called deep fakes.
- Despite these successes, several mathematical and practical issues limit the broader use of DGMs : Given a specitic dataset it remains challenging to design and train a DGM and even more challenging to find out why a particular model is or is not effective.
- Some examples of Generative models :
  - (i) Naïve Bayes,
  - (ii) Bayesian Networks
  - (iii) Markov-random fields
  - (iv) Hidden Markov Models (HMMs)
  - (v) Latent Dirichlet's Allocation (LDA)
  - (vi) Generative Adversarial Networks (GANs)

(vii) Autoregressive Model.

(viii) Generative models are widely used in many subfields of AI and machine learning. Recent advances in parametrizing these models using deep neural networks, combined with progress in stochastic optimization methods, have enabled modeling of complex, high-dimensional data including images, text, and speech.

Discriminator Network

- The discriminator in a GAN (Generative Adversarial network) is simply a classifier. It tries to distinguish real data from the data created by the generator.
- It could use any network architecture appropriate to the type of data it is classifying

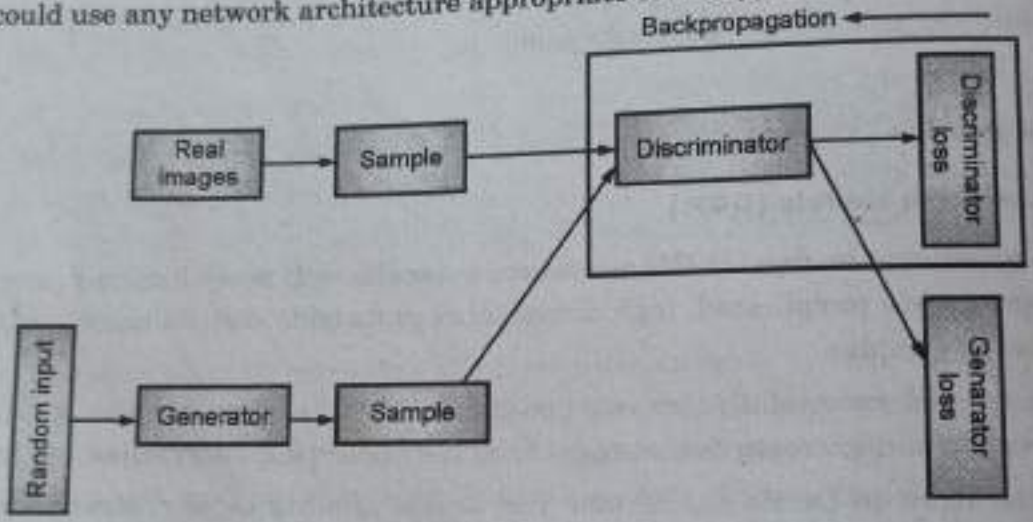


Fig. 6.4.2 : Backpropagation in discriminator training

Discriminator Training Data

The discriminator's training data comes from two sources :

(i) **Real data :** instances, such as real pictures of people. The discriminator uses these instances as positive examples during training.

(ii) **Fake data :** instances created by the generator. The discriminator use these instances as negative examples during training.

In the above Fig. 6.4.2, the two "sample" boxes represent these two data sources feeding into the discriminator.

During discriminator training the generator does not train. Its weights remain constant while it produces examples for the discriminator to train on.

Training the Discriminator

The discriminator connects to two loss functions. During discriminator training, the discriminator ignores the generator loss and just uses the discriminator loss.

During discriminator training

1. The discriminator classifies both real data and fake data from the generator.
2. The discriminator loss penalizes the discriminator for misclassifying a real instance as fake or a fake instance as real.
3. The discriminator updates its weights through 'backpropagation' from the discriminator loss through the discriminator network.

Using the Discriminator to Train the Generator

- To train a neural net, we alter the net's weights to reduce the error or loss of its output
- In our GAN, the generator is not directly connected to the loss that we are trying to affect.
- The generator feeds into the discriminator net, and the 'discriminator' produces the output we are trying to affect.
- The generator loss penalizes the generator for producing a sample that the discriminator network classifies as fake.
- This extra chunk of network must be included in backpropagation.
- Backpropagation adjusts each weight in the right direction by calculating the weight's impact on the output – how the output would change if you changes the weight. But the impact of a generator weight depends upon the impact of the discriminator weights it feeds into. So backpropagation starts at the output and flows back through the discriminator into the generator.
- At the same time we don't want the discriminator to change during generator training. Trying to hit a moving target would make a hard problem even harder for the generator.
- Hence, we train the generator with the following procedure :
  1. Sample random noise.
  2. Produce generator output from sampled random noise.
  3. Get discriminator "Real" or "Fake" classification for generator output.
  4. Calculate loss from discriminator classification.
  5. Backpropagation through both the discriminator and generator to obtain gradients.
  6. use gradients to change only the generator weights.

  This is one iteration of generator training.

Generator Network

- The generator part of a GAN learns to create fake data by incorporating feedback from the discriminator. It learns to make the discriminator classify its output as real.

- Generator training requires tighter integration between the generator and the discriminator than discriminator training requires. The portion of the GAN that trains the generator includes :
  (i) random input
  (ii) generator network, which transforms the random input into a data instance,
  (iii) discriminator network, which classifies the generated data
  (iv) discriminator output
  (v) generator loss, which penalizes the generator for failing to fool the discriminator.

(i) Random Input

- Neural networks need some form of input. Normally we input data that we want to do something with, like an instance that we want to classify or make a prediction about.
- The main problem is : if we input for a network, that outputs entirely new data instances ?
- In its most basic form, a GAN takes random noise as its input. The generator then transforms this noise into a meaningful output.
- By introducing noise, we can get the GAN to produce a wide variety of data, sampling from different places in the target distribution.
- Experiments suggest that the distribution of the noise does not matter much, so we can choose something that is easy to sample from, like a uniform distribution.
- For convenience the space from which the noise is sampled is usually of smaller dimension than the dimensionality of the output space.

(II) GAN Training

CAN contains two separately trained networks, its training algorithm must address two complications :

(a) GANs must judge two different kinds of training (generator and discriminator)
(b) GAN convergence is hard to identify.

(iii) Alternate Training

The generator and the discriminator have different training processes.

GAN training proceeds in alternating periods :

1. The discriminator trains for one or more epochs.
2. The generator trains for one or more epochs.
3. Repeat steps 1 and 2 to continue to train the generator and discriminator networks.

- The generator is kept constant during the discriminator training phase.
- As discriminator training tries to figure out how to distinguish real data from fake, it has to learn how to recognize the generator's flaws.

That is altogether a different problem for a thoroughly trained generator than it is for an untrained generator that produces random output.

- Similarly, we keep the discriminator constant during the generator training phase. Otherwise the generator would be trying to hit a moving target and might never converge.
- It is this important concept that allows GANs to tackle otherwise intractable generative problems. Here, we get a foothold in the difficult generative problem by starting with a much simpler classification problem.
- Conversely, if you cannot train a classifier to tell the difference between real and generated data even for the initial random generator output, you cannot get the GAN training started.

(iv) Convergence

- As the generator improves with training, the discriminator performance gets worse because the discriminator cannot easily tell the difference between real and fake.
- If the generator succeeds perfectly, then the discriminator has a 50% accuracy in effect, the discriminator flips a coin to make its prediction.
- This progression poses a problem for convergence of the GAN as a whole : the discriminator feedback gets less meaningful over time.
- If the GAN continues training past the point when the discriminator is giving completely random feedback, then the generator starts to train on junk feedback, and its own quality may collapse.
- For a GAN, convergence is often a fleeting, rather than stable state.

Loss Functions

- GANs try to replicate a probability distribution. Therefore they use loss function and they reflect the distance between the distribution of the data generated by the GAN and the distribution of the real data.
- To calculate the difference between tow distributions in GAN loss functions, we mention two common GAN – loss functions :

  (1) Minimax loss (2) Wasserstein loss
- A GAN can have two loss functions : one for generator training and one for discriminator training.
- Here we shall see how two loss-functions work together to reflect a distance measure between probability distributions. In the loss schemes, the generator and discriminator losses derive from a single measure of distance between probability distributions.

- In both of these schemes, the generator can only affect one term in the distance measure : the term that reflects the distribution of the fake data.
- So during generator training we drop the other terms, which reflect the distribution of the real data.

The generator and discriminator losses derive from a single formula :

1. Minimax loss

Here the generator tries to minimize the following function while the discriminator tries to maximize it :

$$E_x [\log (D(x))] + E_z [\log (1 - D(G(z)))]$$

In this function :

(i) D (x) is the discriminator's estimate of the probability that the real data instance x is real.

(ii) E_x is the expected value over all real data instances,

(iii) G (Z) is the generator's output when given noise Z.

(iv) D (G(z)) is the discriminator's estimate of the probability that a fake instance is real.

(v) E_z is the expected value over all random inputs to the generator (in effect, the expected values over all generated fake instances G(z)).

(vi) This formula derives from the 'cross-entropy' between the real and generated distributions.

The generator cannot directly affect the 'log (D(x))' term in the function, so, for the generator, minimizing the loss is equivalent to minimizing log (1 – D(G(z))).

Modified Minimax Loss

The above minimax loss function can cause the GAN to get stuck in the early stages of GAN training when the discriminator's job is very easy. Hence there is modification of generator loss so that the generator tries to maximize log D(G(z)).

Wasserstein Loss

- This loss function depends on a modification of the GAN scheme and in which the discriminator does not classify instances.
- For each instance it outputs a number. The number does not have to be less than one or greater than zero, hence we cannot use 0.5 as a threshold to decide whether an instance is real or fake.
- Discriminator training just tries to make the output bigger for real instances than for fake instances.
- It is because it cannot really discriminate between real and fake, Wasserstein GAN (WGAN) discriminator is actually called a 'critic' instead of a "discriminator".

- This distinction has actually theoretical importance, but for practical purposes we can treat it as an acknowledgement that the inputs to the loss functions don't have to be probabilities.
- The loss functions themselves are deceptively simple :

Critic loss : D(x) – D (G(z))

- The discriminator tries to maximize this function. In other words, it tries to maximize the difference between its output on real instances and its output on fake instances.
- Generator loss D (G(z))

The generator tries to maximize this function. In other words, it tries to maximize the discriminator's output for its fake instances.

In these functions :

(i) D(x) is the critic output for a real instance,

(ii) G(z) is the generator's output when given noise Z.

(iii) D (G(z)) is the critic's output for a fake instance.

(iv) The output of critic D does not have to be between 1 and 0.

(v) The formulas derive from the 'earth mover distance' between the real and generated distributions.

The theoretical justification for the Wasserstein GAN (or WGAN) requires that the weights throughout the GAN be clipped so that they remain within a constrained range.

Benefits

(i) Wasserstein GANs are less vulnerable to getting stuck than minimax-based GANs,

(ii) They avoid problems with vanishing gradients.

(iii) The earth mover distance has the advantage of being a true metric : a measure of distance in a space of probability distributions.

(iv) Cross-entropy is not a metric in this sense.

Problems of GANs

GANs have a number of common failure modes. We mention some of the important problems people face.

1. Vanishing Gradients

- It is found that if discriminator is too good, then generator training can fail to 'vanishing gradients'.
- In effect, an optimal discriminator does not provide enough information for the generator to make progress.

Attempts to Remedy

(i) **Wasserstein loss :** The Wasserstein loss is designed to prevent vanishing gradients even when you train the discriminator optimality.

(ii) **Modified minimax loss :** The original GAN proposed a 'modification' to minimax loss to deal with vanishing gradients

2. Mode Collapse

- Generally GAN is supposed to produce a wide variety of output. For example, we want a different face for evey random input to the face generator. If a generator produces a plausible output, the generator may learn to produce only that output.
- Generally, the generator is always trying to find the one output that seems most plausible to the discriminator.
- If the generator starts producing the same output (or a small set of outputs) over and over again, the discriminator's best strategy is to learn to always reject the output. But if the next generation of discriminator gets stuck in a local minimum and does not find the best strategy, then it is too easy for the next generator iteration to find the most plausible output for the current discriminator.
- Each iteration of generator over-optimises for a particular discriminator, and the discriminator never manages to learn its way out of the trap. Hence, the generators rotate through a small set of output types.
- This form of GAN failure is called 'mode-collopse'.

Attempts to Remedy : The following mentioned approaches try to force the generator to broaden its scope by preventing it from optimizing for a single fixed discriminator :

(i) **Wasserstein loss :** The 'Wasserstein loss' alleviates mode collapse by allowing to train the discrimmator to optimality without worrying about vanishing gradients. If the discriminator does not get stuck in local minima, it learns to reject the output that the generator stabilizes on. So the generator has to try something new.

(ii) **Unrolled GANs :** 'Unrolled GANs' use a generator loss functions that incorporates not only the current discriminator's classifications, but also the output of future discriminator versions. So the generator cannot over-optimise for a single discriminator.

GAN Variations

(i) Progressive GANs

- In a progressive GAN, the generator's first layers produce very low resolution images, and subsequent layers add details.
- This technique allows the GAN to train more quickly than comparable non-progressive GANs, and produces higher resolution images.

(ii) Image – to-image translation

- Image – to-image translation GANs can take an image as input and map it to a generated output image with different properties.
- For examples, we can take a mask image with blob of colour in the shape of a car, and the GAN can fill in the shape with photorealistic car details.
- Similarly, one can train an image-to-image GAN to take sketches of handbags and turn them into phtorealistic images of handbags.
- In these above cases, the loss is a weighted combination of the usual discriminator-based loss and a pixel-wise loss that penalises the generator for departing from the source image.

(iii) Super-resolution

- Super resolution GANs increase the resolution of images adding where necessary to fill in blurry area.
- Given the blurry image, a GAN produces the sharper image. The GAN generated images looks very similar to the original image.

(iv) Face Inpainting

- GANs have been used for the 'semantic image in painting' task.
- In the inpainting task, chunks of an image are blacked out and the system tries to fill in the missing chunks.

(v) Test-to-Speech

We note that not all GANs produce images. For example, researchers have also used GANs to produce synthesized speech from text input.

Applications of GAN networks

- GANs have very specific use cases. After training, the generative model can be used to create new plausible samples on demand.
- We divide the applications into the following areas :
  1. Generate examples for image datasets.
  2. Generate photographs of Human faces
  3. Generate Realistic photographs
  4. Generate cartoon characters.
  5. Image-to-image Translation
  6. Text-to-image translation
  7. Semantic-Image-to-photo-translation
  8. Face frontal view generation

9. Generate new Human poses
10. Photos to Emojis,
11. Photograph Editing
12. Face Aging
13. Photo Blending
14. Super Resolution
15. Photo In painting
16. Clothing Translation
17. Video prediction
18. Object Generation.

We illustrate each application step-by-step.

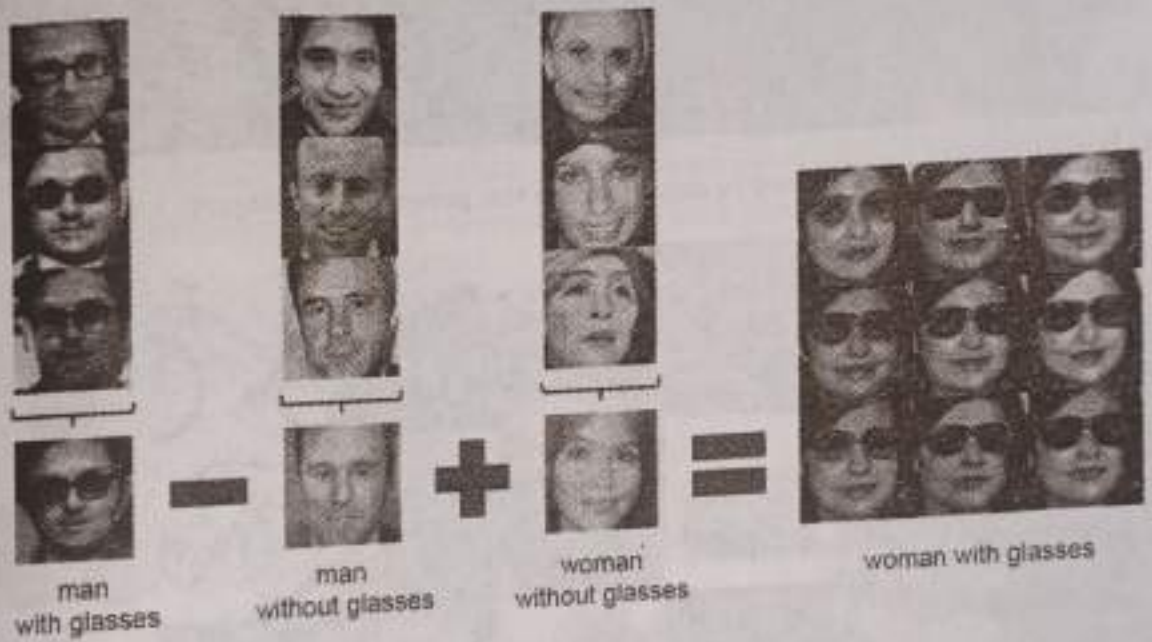
1. Generate examples for image datasets

Generating new plausible samples was the application where GANs were used to generate new plausible examples for the MNTST handwritten digit dataset, the (IFAR – 10 small) object photograph dataset, (IFAR-10 small).

This was also demonstration of how to train stable GANs at scale.

Demonstration of models for generating new examples of bedrooms.

Also there is demonstration of the ability to perform vector arithmetic with the input to the GANs with generated faces.



2. Generate photographs of human faces

- Tero Kerras demonstrated the generation of plausible photographs of human faces they are fair looking.
- The face generations were trained on celebrity examples, meaning, that there are elements of existing celebrities in the generated faces, making them seem familiar.

Their method were also used to demonstrate the generation of objects and scenes.

3. Generate Realistic photographs

Andrew Brock demonstrated the generation of synthetic photographs with their technique BigGAN that are practically indistinguishable from real photographs

4. Generate cartoon characters

Yanghua Jin in 2017 demonstrated the training and use of a GAN for generating faces of animal characters (anime-i.e. Japanese comic book characters).

5. Image-to-image translation

This is a bit of catch-all tasks, for those that present GANs that can do many image translation tasks.

Examples include translation tasks such as :

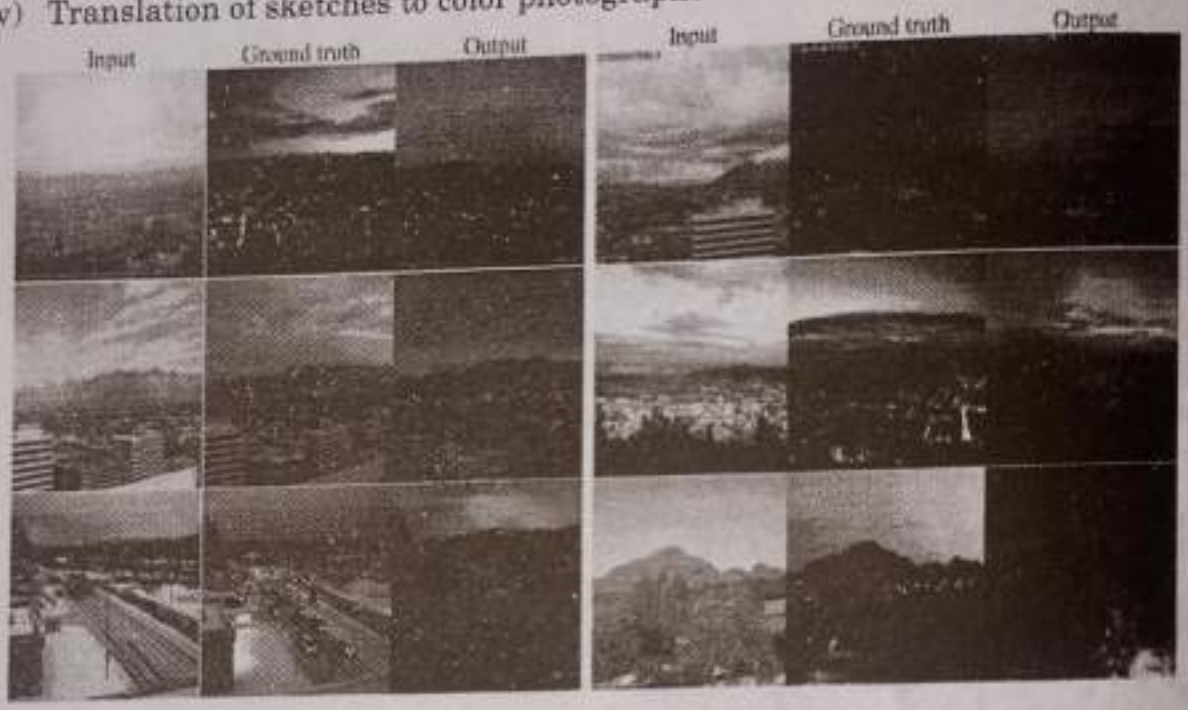
(i) translation of semantic images to photographs of cityscapes and buildings.

(ii) Translation of satellite photographs to Google maps.

(iii) Translation of photos from day to night.

(iv) Translation of black and white photographs to colour.

(v) Translation of sketches to color photographs.



The examples below demonstrates four image translation cases :

(i) Translation from photograph to artistic painting style.

(ii) Translation from horse to zebra.

(iii) Translation of photograph from summer to winter

(iv) Translation of satellite photograph to Google Maps view.

It also provides many other examples, such as :

(i) Translation of painting to photographs,

(ii) Translation of sketch to photograph.

(iii) Translation of apple to orange.

(iv) Translation of photograph to artistic painting

6. Text-to-Image translation : (text 2 image)

Hari Zhang demonstrated the use of GANs, specifically their stack GAN to generate realistic looking photographs form textual descriptions of simple objects like birds and flowers.

Chapter Ends ...

❑❑❑