UNIT -1

What is a Cost Function?

It is a **function** that measures the performance of a model for any given data. **Cost Function** quantifies the error between predicted values and expected values and presents it in the form of a single real number.

After making a hypothesis with initial parameters, we calculate the Cost function. And with a goal to reduce the cost function, we modify the parameters by using the Gradient descent algorithm over the given data. Here's the mathematical representation for it:

Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$

Parameters: θ_0, θ_1

Cost Function: $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$

Goal: $\underset{\theta_0, \theta_1}{\text{minimize}}\ J(\theta_0, \theta_1)$

What is Gradient Descent?

Gradient descent is an optimization algorithm used in machine learning to minimize the cost function by iteratively adjusting parameters in the direction of the negative gradient, aiming to find the optimal set of parameters.

The cost function represents the discrepancy between the predicted output of the model and the actual output. The goal of gradient descent is to find the set of parameters that minimizes this discrepancy and improves the model's performance.

The algorithm operates by calculating the gradient of the cost function, which indicates the direction and magnitude of steepest ascent. However, since the objective is to minimize the cost function, gradient descent moves in the opposite direction of the gradient, known as the negative gradient direction.

By iteratively updating the model's parameters in the negative gradient direction, gradient descent gradually converges towards the optimal set of parameters that yields the lowest cost.

The learning rate, a hyperparameter, determines the step size taken in each iteration, influencing the speed and stability of convergence.

Gradient descent can be applied to various machine learning algorithms, including linear regression, logistic regression, neural networks, and support vector machines. It provides a general framework for optimizing models by iteratively refining their parameters based on the cost function.

Example of Gradient Descent

Let’s say you are playing a game where the players are at the top of a mountain, and they are asked to reach the lowest point of the mountain. Additionally, they are blindfolded. So, what approach do you think would make you reach the lake?

Take a moment to think about this before you read on.

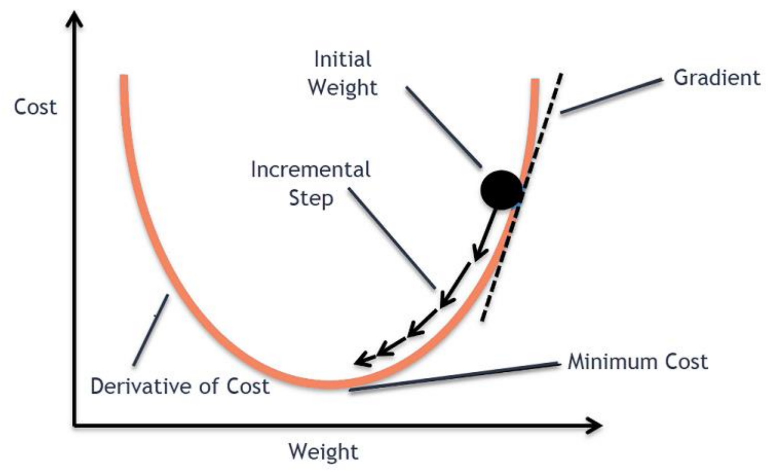
The best way is to observe the ground and find where the land descends. From that position, take a step in the descending direction and iterate this process until we reach the lowest point.

Finding the lowest point in a hilly landscape. (Source: Fisseha Berhane)

Gradient descent is an iterative optimization algorithm for finding the local minimum of a function.

To find the local minimum of a function using gradient descent, we must take steps proportional to the negative of the gradient (move away from the gradient) of the function at the current point. If we take steps proportional to the positive of the gradient (moving towards the gradient), we will approach a local maximum of the function, and the procedure is called **Gradient Ascent.**

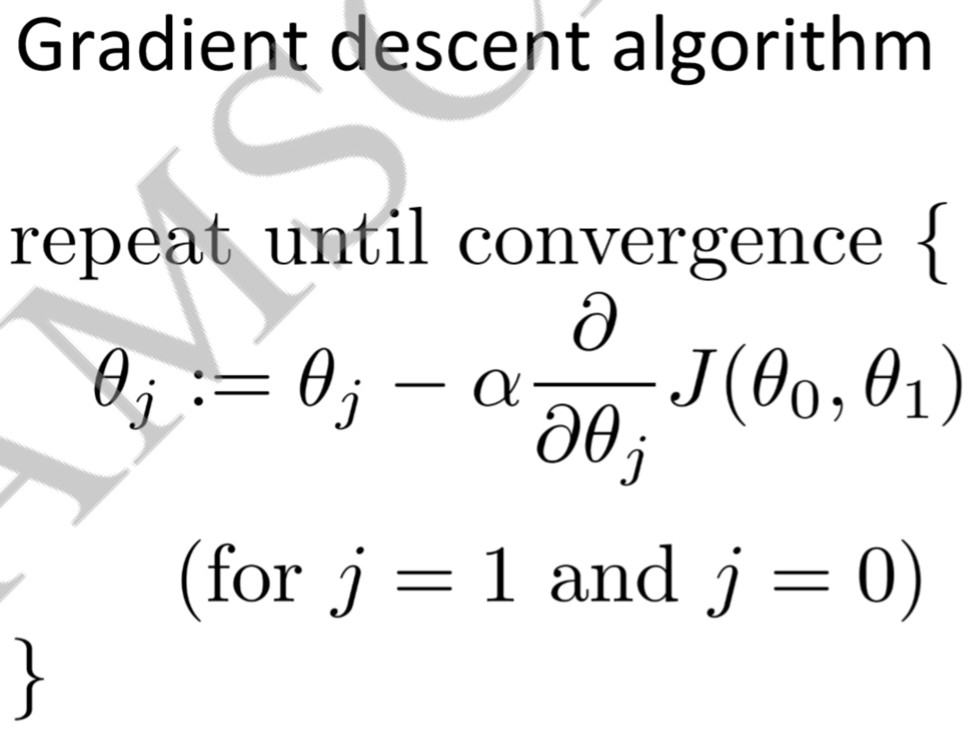
Gradient descent was originally proposed by **CAUCHY** in 1847. It is also known as steepest descent.



Source: Clairvoyant

The goal of the gradient descent algorithm is to minimize the given function (say cost function). To achieve this goal, it performs two steps iteratively:

1. **Compute the gradient** (slope), the first order derivative of the function at that point
2. **Make a step (move) in the direction opposite to the gradient**, opposite direction of slope increase from the current point by alpha times the gradient at that point



Gradient descent algorithm

```
repeat until convergence {
    $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$
        (for $j = 1$ and $j = 0$)
}
```

Alpha is called **Learning rate** – a tuning parameter in the optimization process. It decides the length of the steps.

How Does Gradient Descent Work?

1. Gradient descent is an optimization algorithm used to minimize the cost function of a model.
2. The cost function measures how well the model fits the training data and is defined based on the difference between the predicted and actual values.
3. The gradient of the cost function is the derivative with respect to the model's parameters and points in the direction of the steepest ascent.
4. The algorithm starts with an initial set of parameters and updates them in small steps to minimize the cost function.
5. In each iteration of the algorithm, the gradient of the cost function with respect to each parameter is computed.
6. The gradient tells us the direction of the steepest ascent, and by moving in the opposite direction, we can find the direction of the steepest descent.
7. The size of the step is controlled by the learning rate, which determines how quickly the algorithm moves towards the minimum.
8. The process is repeated until the cost function converges to a minimum, indicating that the model has reached the optimal set of parameters.
9. There are different variations of gradient descent, including batch gradient descent, stochastic gradient descent, and mini-batch gradient descent, each with its own advantages and limitations.
10. Efficient implementation of gradient descent is essential for achieving good performance in machine learning tasks. The choice of the learning rate and the number of iterations can significantly impact the performance of the algorithm.

Types of Gradient Descent

The choice of gradient descent algorithm depends on the problem at hand and the size of the dataset. Batch gradient descent is suitable for small datasets, while stochastic gradient descent is more suitable for large datasets. Mini-batch gradient descent is a good compromise between the two and is often used in practice.

Batch Gradient Descent

Batch gradient descent updates the model's parameters using the gradient of the entire training set. It calculates the average gradient of the cost function for all the training examples and updates the parameters in the opposite direction. Batch gradient descent guarantees convergence to the global minimum, but can be computationally expensive and slow for large datasets.

Stochastic Gradient Descent

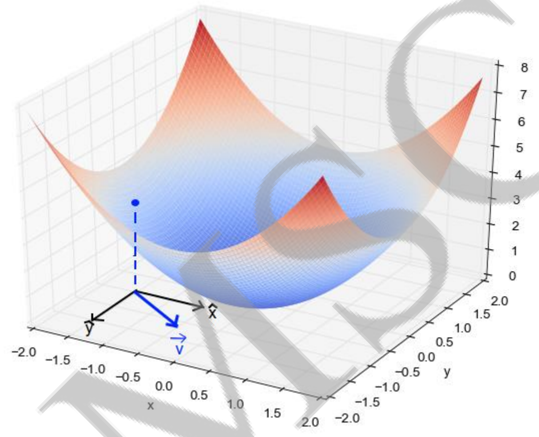
Stochastic gradient descent updates the model's parameters using the gradient of one training example at a time. It randomly selects a training example, computes the gradient of the cost function for that example, and updates the parameters in the opposite direction. Stochastic gradient descent is computationally efficient and can converge faster than batch gradient descent. However, it can be noisy and may not converge to the global minimum.

Mini-Batch Gradient Descent

Mini-batch gradient descent updates the model's parameters using the gradient of a small subset of the training set, known as a mini-batch. It calculates the average gradient of the cost function for the mini-batch and updates the parameters in the opposite direction. Mini-batch gradient descent combines the advantages of both batch and stochastic gradient descent, and is the most commonly used method in practice. It is computationally efficient and less noisy than stochastic gradient descent, while still being able to converge to a good solution.

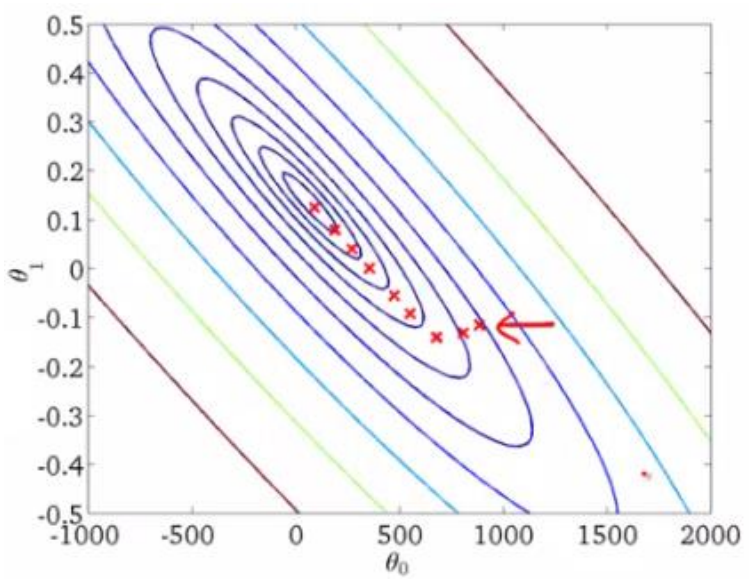
Plotting the Gradient Descent Algorithm

When we have a single parameter (theta), we can plot the dependent variable cost on the y-axis and theta on the x-axis. If there are two parameters, we can go with a 3-D plot, with cost on one axis and the two parameters (thetas) along the other two axes.



cost along z-axis and parameters(thetas) along x-axis and y-axis (source: Research gate)

It can also be visualized by using **Contours.** This shows a 3-D plot in two dimensions with parameters along both axes and the response as a contour. The value of the response increases away from the center and has the same value along with the rings. The response is directly proportional to the distance of a point from the center (along a direction).



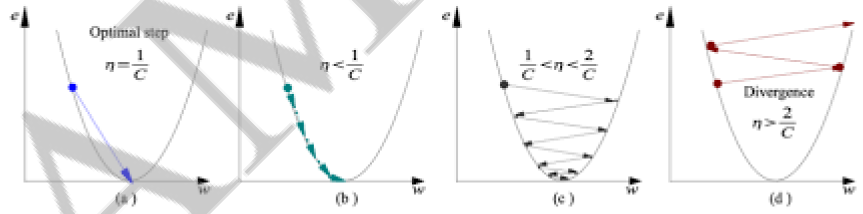
Gradient descent using Contour Plot. (source: Coursera)

Alpha – The Learning Rate

We have the direction we want to move in, now we must decide the size of the step we must take.

It must be chosen carefully to end up with local minima.

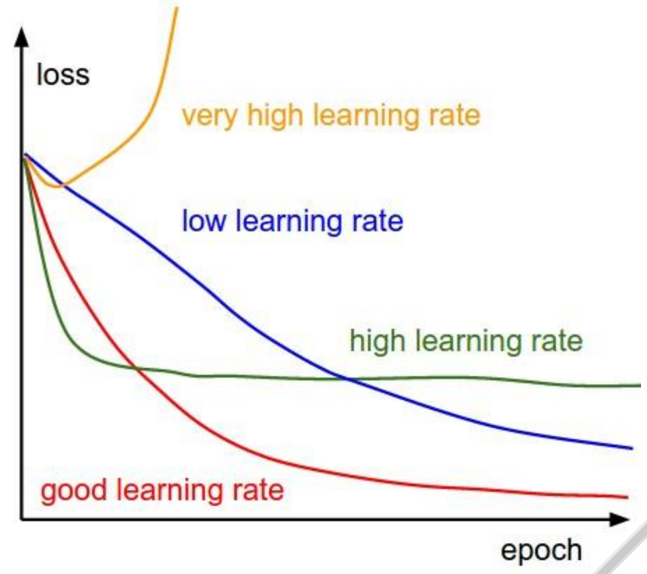
- If the learning rate is too high, we might **OVERSHOOT** the minima and keep bouncing, without reaching the minima
- If the learning rate is too small, the training might turn out to be too long



Source: Coursera

1. a) Learning rate is optimal, model converges to the minimum
2. b) Learning rate is too small, it takes more time but converges to the minimum

3. c) Learning rate is higher than the optimal value, it overshoots but converges ($1/C < \eta < 2/C$)
4. d) Learning rate is very large, it overshoots and diverges, moves away from the minima, performance decreases on learning

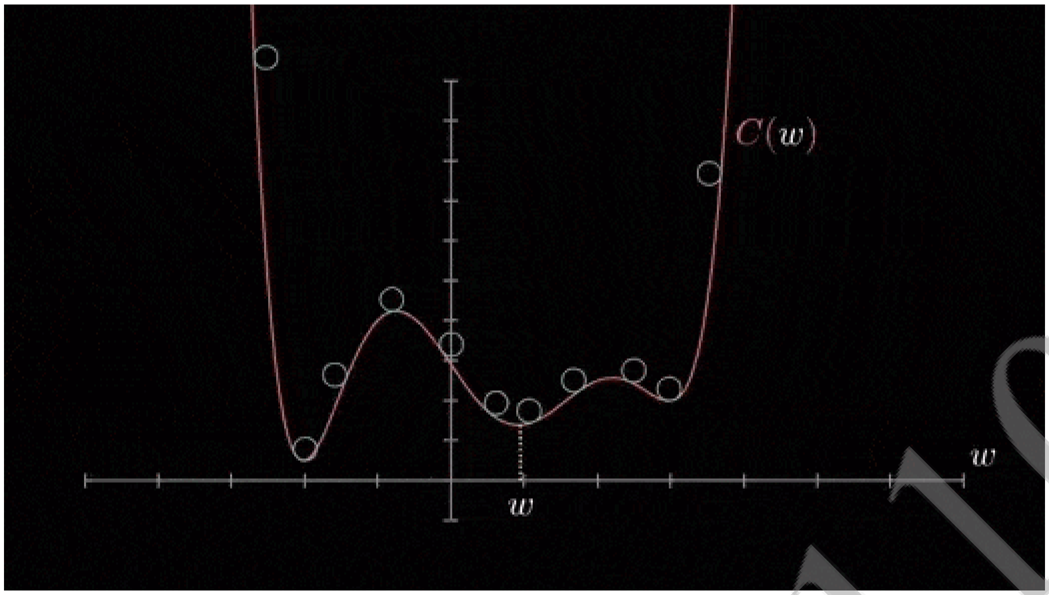


Source: researchgate

Note: *As the gradient decreases while moving towards the local minima, the size of the step decreases. So, the learning rate (alpha) can be constant over the optimization and need not be varied iteratively.*

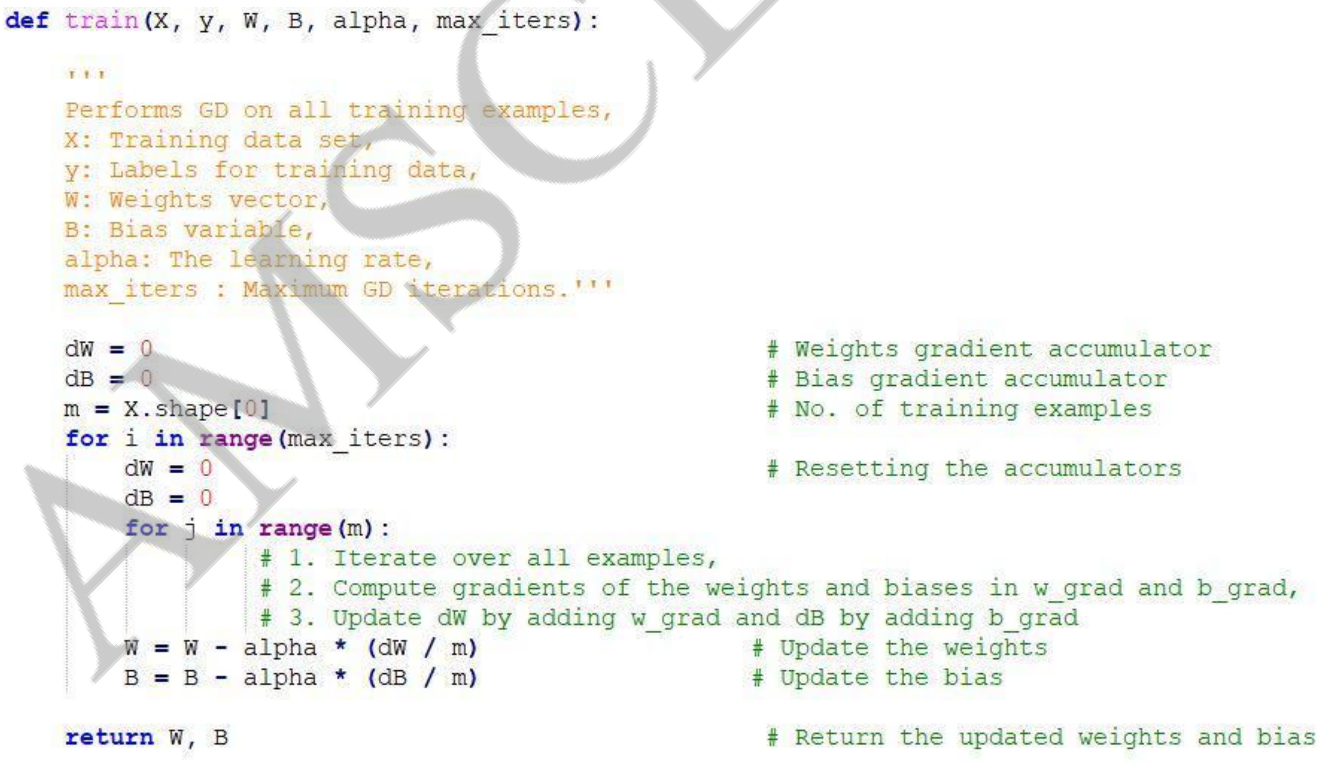
Local Minima

The cost function may consist of many minimum points. The gradient may settle on any one of the minima, which depends on the initial point (i.e initial parameters(theta)) and the learning rate. Therefore, the optimization may converge to different points with different starting points and learning rate.



Convergence of cost function with different starting points (Source: Gfycat)

Code Implementation of Gradient Descent in Python



```python
def train(X, y, W, B, alpha, max_iters):

    '''
    Performs GD on all training examples,
    X: Training data set,
    y: Labels for training data,
    W: Weights vector,
    B: Bias variable,
    alpha: The learning rate,
    max_iters : Maximum GD iterations.'''

    dW = 0                                          # Weights gradient accumulator
    dB = 0                                          # Bias gradient accumulator
    m = X.shape[0]                                  # No. of training examples
    for i in range(max_iters):
        dW = 0                                      # Resetting the accumulators
        dB = 0
        for j in range(m):
            # 1. Iterate over all examples,
            # 2. Compute gradients of the weights and biases in w_grad and b_grad,
            # 3. Update dW by adding w_grad and dB by adding b_grad
        W = W - alpha * (dW / m)                   # Update the weights
        B = B - alpha * (dB / m)                   # Update the bias

    return W, B                                     # Return the updated weights and bias
```

Challenges of Gradient Descent

While gradient descent is a powerful optimization algorithm, it can also present some challenges that can affect its performance. Some of these challenges include:

1. Local Optima: Gradient descent can converge to local optima instead of the global optimum, especially if the cost function has multiple peaks and valleys.
2. Learning Rate Selection: The choice of learning rate can significantly impact the performance of gradient descent. If the learning rate is too high, the algorithm may overshoot the minimum, and if it is too low, the algorithm may take too long to converge.
3. Overfitting: Gradient descent can overfit the training data if the model is too complex or the learning rate is too high. This can lead to poor generalization performance on new data.
4. Convergence Rate: The convergence rate of gradient descent can be slow for large datasets or high-dimensional spaces, which can make the algorithm computationally expensive.
5. Saddle Points: In high-dimensional spaces, the gradient of the cost function can have saddle points, which can cause gradient descent to get stuck in a plateau instead of converging to a minimum.

To overcome these challenges, several variations of gradient descent have been developed, such as adaptive learning rate methods, momentum-based methods, and second-order methods. Additionally, choosing the right regularization method, model architecture, and hyperparameters can also help improve the performance of gradient descent.

NNDL-unit 2 - Notes

Neural Network and Deep Learning (Anna University)

UNIT II

2 Associative Memory and Unsupervised Learning Networks

Syllabus

Training Algorithms for Pattern Association-Autoassociative Memory Network-Heteroassociative Memory Network-Bidirectional Associative Memory (BAM) - Hopfield Networks - Iterative Autoassociative Memory Networks-Temporal Associative Memory Network - Fixed Weight Competitive Nets - Kohonen Self - Organizing Feature Maps - Learning Vector Quantization - Counter propagation Networks - Adaptive Resonance Theory Network.

Contents

2.1 Training Algorithms for Pattern Association

- Pattern association is the process of memorizing input-output patterns in a hetero-associative network architecture, or input patterns only in an auto-associative network, in order to recall the patterns when a new input pattern is presented.
- Pattern association learns associations between input patterns and output patterns. It is widely used in distributed memory modeling. It is one of the more basic two-layer networks.
- Its architecture consists of two sets of units, the input units and the output units. Each input unit connects to each output unit via weighted connections. The connections are only allowed from input units to output units.
- The effect of a unit u_i in the input layer on a unit u_j in the output layer is determined by the product of the activation a_i of u_i and the weight of the connection from u_i to u_j. The activation of a unit u_j in the output layer is given by : $\mathrm{SUM}(w_{ij} \times a_i)$
- A pattern association can be trained to respond with a certain output pattern when presented with an input pattern. The connection weights can be adjusted in order to change the input/output behavior. The learning rule is what specifies now a network changes it weights for a given input/output association.
- The most commonly used learning rules with pattern associators are Hebb rule and the delta rule.

2.1.1 Hebb Rule

- Hebb rule is the simplest and most common method of determining weights for an associative memory neural net. It can be used with patterns are represented as either binary or bipolar vectors
- Hebb's Law states that if neuron i is near enough to excite neuron j and repeatedly participates in its activation, the synaptic connection between these two neurons is strengthened and neuron j becomes more sensitive to stimuli from neuron i.
- According to Hebb rule, weight vector is found to increase proportionately to the product of the input and learning signal.
- Hebb's Law can be represent in the form two rules :
  1. If two neurons on either side of a connection are activated synchronously, then the weight of that connection is increased.
  2. If two neurons on either side of a connection are activated asynchronously, then the weight of that connection is decreased.
- Fig. 2.1.1 shows Hebbian learning in a neural network.

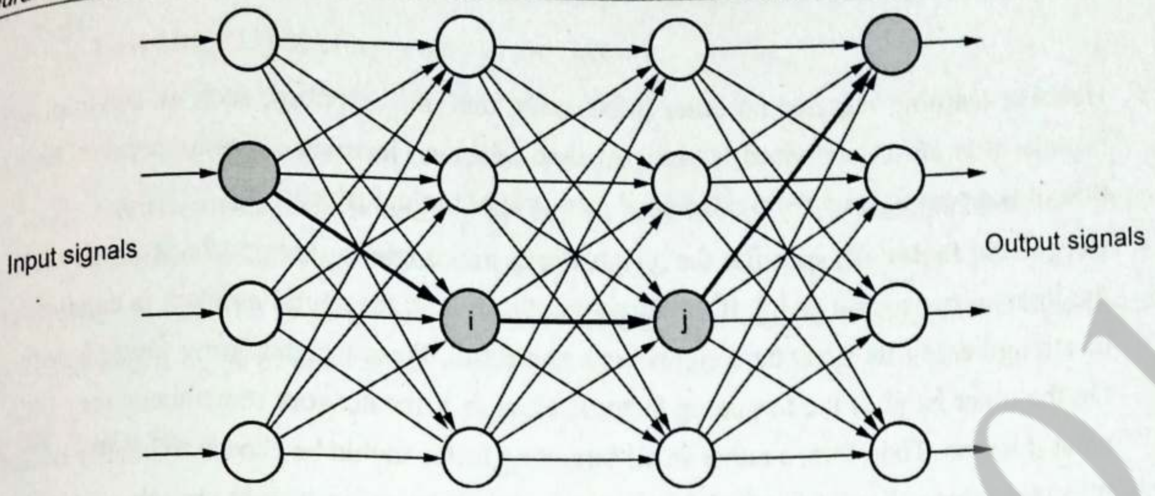


Fig. 2.1.1 Hebbian learning in a neural network

- Using Hebb's Law, we can express the adjustment applied to the weight w_{ij} at iteration p in the following form :

$$\Delta W_{ij}(p) = F[y_j(p), x_i(p)],$$

- where $F[y_j(p), x_i(p)]$ is a function of both postsynaptic and presynaptic activities.
- As a special case, we can represent Hebb's law

$$\Delta W_{ij}(p) = \alpha y_i(p)\, x_i(p)$$

where α is the learning rate parameter.

- This equation is referred to as the activity product rule. It shows how a change in the weight of the synaptic connection between a pair of neurons is related to a product of the incoming and outgoing signals.
- Hebbian learning implies that weights can only increase. In other words, Hebb's Law allows the strength of a connection to increase, but it does not provide a means to decrease the strength. Thus, repeated application of the input signal may drive the weight w_{ij} into saturation.
- The w_{ij} stands for the weight of the connection from neuron j to neuron i. Fig. 2.1.2 shows Two connected neurons (w_{ij}).

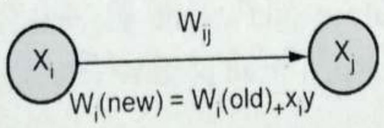


Fig. 2.1.2 Two connected neurons

- To resolve this problem, we might impose a limit on the growth of synaptic weights. It can be done by introducing a non-linear forgetting factor into Hebb's Law.

- Hebbian learning requires no other information than the activities, such as labels or error signals: it is an unsupervised learning method. Hebbian learning is not a concrete learning rule, it is a postulate on the fundamental principle of biological learning.
- **Forgetting factor (∅)** specifies the weight decay in a single learning cycle. It usually falls in the interval between 0 and 1. If the forgetting factor is 0, the neural network is capable only of strengthening its synaptic weights, and as a result, these weights grow towards infinity. On the other hand, if the forgetting factor is close to 1, the network remembers very little of what it learns. Therefore, a rather small forgetting factor should be chosen, typically between 0.01 and 0.1, to allow only a little 'forgetting' while limiting the weight growth.
- Fig. 2.1.3 shows flow chart of Hebb training algorithm.

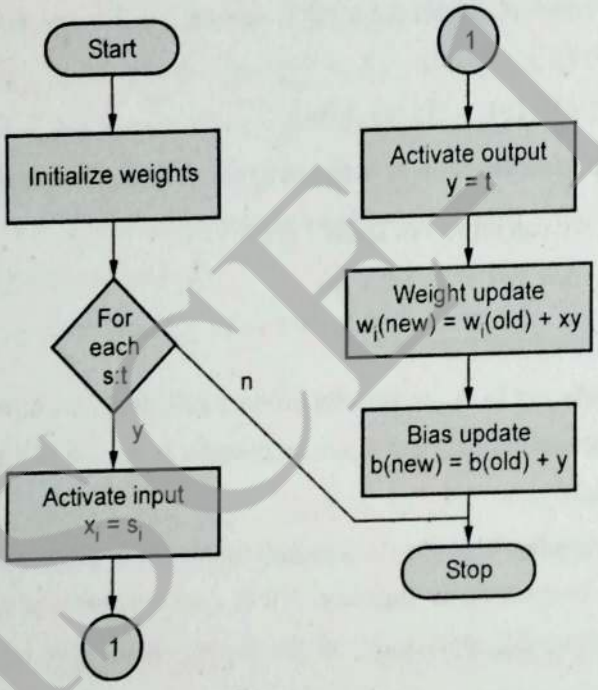

Start
Initialize weights
For each s:t
n
y
Activate input $x_i = s_i$
1
1
Activate output $y = t$
Weight update $w_i(new) = w_i(old) + xy$
Bias update $b(new) = b(old) + y$
Stop

Fig. 2.1.3 Flow chart of Hebb training algorithm

- **Generalized Hebbian learning algorithm :**
  1. **Initialization :** Set initial synaptic weights and thresholds to small random values, say in an interval [0, 1]. Also assign small positive values to the learning rate parameter α and forgetting factor ∅.
  2. **Activation :** Compute the neuron output at iteration p

$$y_j(p) = \sum_{i=1}^{n} x_i(p)\, w_{ij}(p) - \theta_j$$

where n is the number of neuron inputs, and θ_j is the threshold value of neuron j.

3. **Learning :** Update the weights in the network

$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p)$$

where $\Delta w_{ij}(p)$ is the weight correction at iteration p.

4. **Iteration :** Increase iteration p by one, go back to Step 2 and continue until the synaptic weights reach their steady-state values

- Hebb rule can be used for pattern association, pattern categorization, pattern classification and over a range of other areas.

2.1.2 Delta Rule

- An important generalization of the perceptron training algorithm was presented by Widrow and Hoff as the least mean square learning procedure also known as the delta rule.
- The learning rule was applied to the "**adaptive linear element**" also named Adaline.
- The perceptron learning rule uses the output of the threshold function for learning. The delta rule uses the net output without further mapping into output values –1 or +1.
- Fig. 2.1.4 shows adaline.

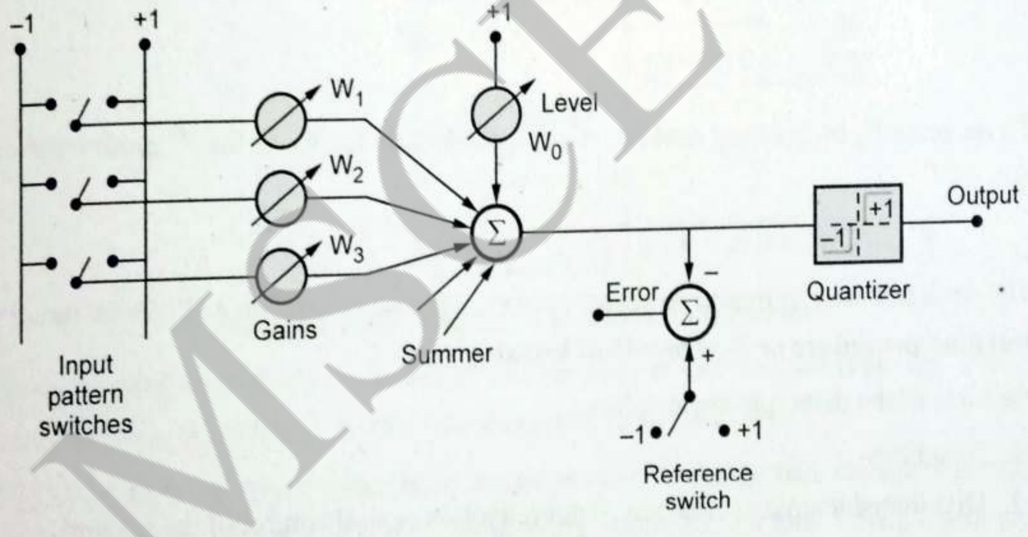


Fig. 2.1.4 Adaline

- If the input conductances are denoted by w_i, where i = 0, 1, 2,.....n, and input and output signals by x_i and y, respectively, then the output of the central block is defined to be :

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

Where $\theta = w_0$

- In a simple physical implementation this device consists of a set of controllable resistors connected to a circuit which can sum up currents caused by the input voltage signals.

Usually the central block the summer is also followed by a quantizer which outputs either + 1 or − 1, depending on the polarity of the sum.

- The problem is to determine the coefficients w_i where $i = 0, 1, ..., n$, in such a way that the input output response is correct for a large number of arbitrarily chosen signal sets.
- If an exact mapping is not possible the average error must be minimized, for instance, in the sense of least squares.
- An adaptive operation means that there exists a mechanism by which the w_i can be adjusted, usually iteratively to attain the correct values.
- For the Adaline, Widrow introduced the delta rule to adjust the weights.
- For the p^{th} input-output pattern, the error measure of a single-output Adaline can be expressed as,

$$E_p = (t_p - o_p)^2$$

where, t_p = Target output

o_p = Actual output of the Adaline

- The derivation of E_p with respect to each weight w_i is

$$\frac{\partial E_p}{\partial w_i} = -2\,(t_p - o_p)\,x_i$$

- To decrease E_p by gradient descent, the update formula for w_i on the p^{th} input-output pattern is

$$\Delta_p w_i = \eta\,(t_p - o_p)\,x_i$$

- The delta rule tries to minimize squared errors, it is also referred to as the **least mean square learning procedure or Widrow-Hoff learning rule**.
- Features of the delta rule are as follows :
  1. Simplicity
  2. Distributed learning : Learning is not reliant on central control of the network.
  3. Online learning : Weights are updated after presentation of each pattern.

2.2 Associative Memory Network

- One of the primary functions of the brain is associative memory. Learning can be considered as a process of forming associations between related patterns. The associative memory is composed of a cluster of units which represent a simple model of a real biological neuron.
- An associative memory, also known as Content-Addressable Memory (CAM) can be searched for a value in a single memory cycle rather than using a software loop.
- Associative memories can be implemented using networks with or without feedback. Such

Usually the central block the summer is also followed by a quantizer which outputs either + 1 or − 1, depending on the polarity of the sum.

- The problem is to determine the coefficients w_i where $i = 0, 1.....n$, in such a way that the input output response is correct for a large number of arbitrarily chosen signal sets.
- If an exact mapping is not possible the average error must be minimized, for instance, in the sense of least squares.
- An adaptive operation means that there exists a mechanism by which the w_i can be adjusted, usually iteratively to attain the correct values.
- For the Adaline, Widrow introduced the delta rule to adjust the weights.
- For the pth input-output pattern, the error measure of a single-output Adaline can be expressed as,

$$E_p = (t_p - o_p)^2$$

where, t_p = Target output

o_p = Actual output of the Adaline

- The derivation of E_p with respect to each weight w_i is

$$\frac{\partial E_p}{\partial w_i} = -2\,(t_p - o_p)\,x_i$$

- To decrease E_p by gradient descent, the update formula for w_i on the pth input-output pattern is

$$\Delta_p w_i = \eta\,(t_p - o_p)\,x_i$$

- The delta rule tries to minimize squared errors, it is also referred to as the **least mean square learning procedure or Widrow-Hoff learning rule.**
- Features of the delta rule are as follows :
  1. Simplicity
  2. Distributed learning : Learning is not reliant on central control of the network.
  3. Online learning : Weights are updated after presentation of each pattern.

2.2 Associative Memory Network

- One of the primary functions of the brain is associative memory. Learning can be considered as a process of forming associations between related patterns. The associative memory is composed of a cluster of units which represent a simple model of a real biological neuron.
- An associative memory, also known as Content-Addressable Memory (CAM) can be searched for a value in a single memory cycle rather than using a software loop.
- Associative memories can be implemented using networks with or without feedback. Such

associative neural networks are used to associate one set of vectors with another set of vectors, say input and output patterns.

- The aim of an associative memory is, to produce the associated output pattern whenever one of the input patterns is applied to the neural network. The input pattern may be applied to the network either as input or as initial state and the output pattern is observed at the outputs of some neurons constituting the network.
- Associative memories belong to class of neural network that learn according to a certain recording algorithm. They require information a priori and their connectivity matrices most often need to be formed in advance. Writing into memory produces changes in the neural interconnections. Reading of the stored info from memory named recall, is a transformation of input signals by the network.
- All memory information is spatially distributed throughout the network. Associative memory enables a parallel search within a stored data. The purpose of search is to output one or all stored items that matches the search argument and retrieve it entirely or partially.
- The Fig. 2.2.1 shows a block diagram of an associative memory.

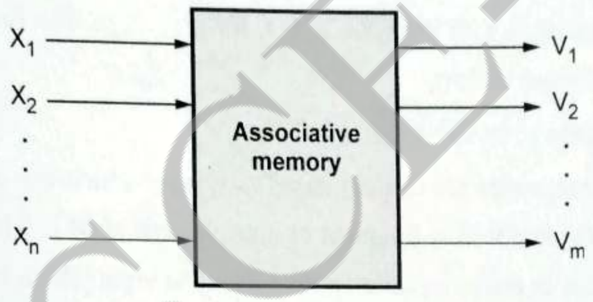


Fig. 2.2.1 Block diagram of an associative memory

- In the initialization phase of the associative memory no information is stored; ? because the information is represented in the w weights they are all set to zero.
- The advantage of neural associative memories over other pattern storage algorithms like lookup tables of hash codes is that the memory access can be fault tolerant with respect to variation of the input pattern.
- In associative memories many associations can be stored at the same time. There are different schemes of superposition of the memory traces formed by the different associations. The superposition can be simple linear addition of the synaptic changes required for each association (like in the Hopfield model) or nonlinear.
- The performance of neural associative memories is usually measured by a quantity called information capacity, that is, the information content that can be learned and retrieved, divided by the number of synapses required.

- An associative memory is a content-addressable structure that maps specific input representations to specific output representations. It is a system that "associates" two patterns (X, Y) such that when one is encountered, the other can be recalled.
- Associative network memory can be static or dynamic.
- Static : networks recall an output response after an input has been applied in one feed-forward pass and theoretically without delay. They were termed instantaneous.
- Dynamic : memory networks produce recall as a result of output/input feedback interaction, which requires time.
- There are two classes of associative memory : auto-associative and hetero-associative.
- Whether auto- or hetero-associative, the net can associate not only the exact pattern pairs used in training, but is also able to obtain associations if the input is similar to one on which it has been trained.

2.2.1 Auto-associative Memory

- Auto-associative networks are a special subset of the hetero-associative networks, in which each vector is associated with itself, i.e. $y^i = x^i$ for i=1, ..., m. The function of such networks is to correct noisy input vectors.
- Fig. 2.2.2 shows auto-associative memory.
- Auto-associative memories are content based memories which can recall a stored sequence when they are presented with a fragment or a noisy version of it. They are very effective in de-noising the input or removing interference from the input which makes them a promising first step in solving the cocktail party problem.
- The simplest version of auto-associative memory is linear associator which is a two-layer feed-forward fully connected neural network where the output is constructed in a single feed-forward computation.

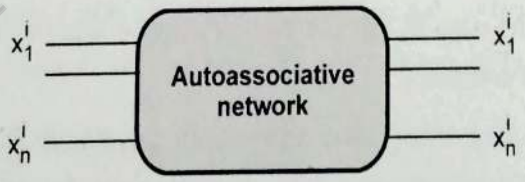


Fig. 2.2.2 Auto-associative memory

- Artificial neural networks can be used as associative memories. One of the simplest artificial neural associative memory is the linear associator. The Hopfield model and Bidirectional Associative Memory (BAM) models are some of the other popular artificial neural network models used as associative memories.

2.2.2 Hetero-associative Memory Network

- Hetero-associative networks map "m" input vectors $X^1, X^2, \dots, X^m$ in n-dimensional space to m output vectors $y^1, y^2, \dots, y^m$ in k-dimensional space, so that $X^i \rightarrow y^i$.
- If $\| \tilde{X} - X^i \|^2 < \epsilon$ then $\tilde{x} - y^i$. This should be achieved by the learning algorithm, but becomes very hard when the number m of vectors to be learned is too high.
- Fig. 2.2.3 shows block diagram of hetero-associative network.

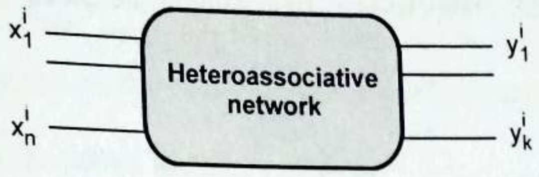


Fig. 2.2.3 Auto-associative memory

- Fig. 2.2.4 shows the structure of a hetero-associative network without feedback.

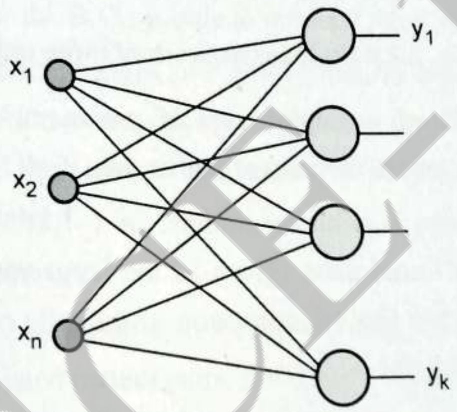


Fig. 2.2.4 Hetero-associative network without feedback

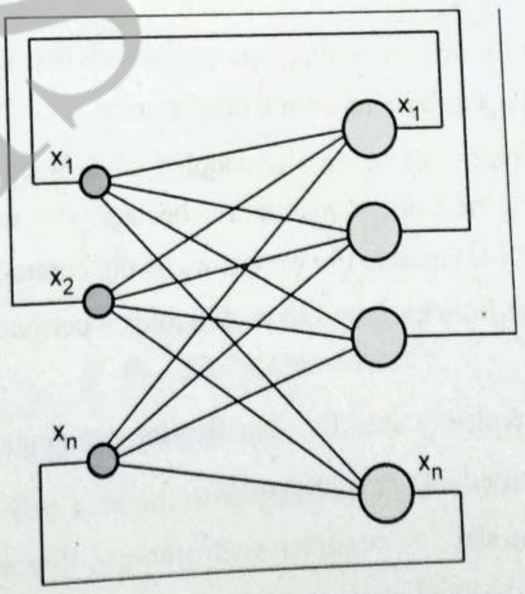


Fig. 2.2.5 Hetero-associative network without feedback

2.2.3 The Hopfield Network

- The **Hopfield** model is a single-layered recurrent network. Like the associative memory, it is usually initialized with appropriate weights instead of being trained.
- Hopfield Neural Network (HNN) is a model of auto-associative memory. It is a single layer neural network with feedbacks. Fig. 2.2.6 shows Hopfield network of three units. The Hopfield network is created by supplying input data vectors, or pattern vectors, corresponding to the different classes. These patterns are called class patterns.

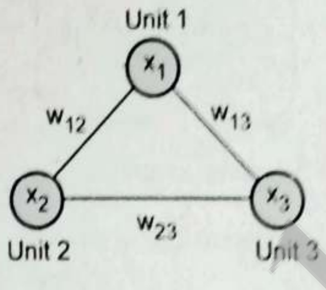


Fig. 2.2.6 Hopfield network of three units

- Hopfield model consists of a single layer of processing elements where each unit is connected to every other unit in the network other than itself.
- The output of each neuron is a binary number in $\{-1,1\}$. The output vector is the state vector. Starting from an initial state (given as the input vector), the state of the network changes from one to another like an automaton. If the state converges, the point to which it converges is called the attractor.
- In its simplest form, the output function is the sign function, which yields 1 for arguments ≥ 0 and -1 otherwise.
- The connection weight matrix W of this type of network is square and symmetric. The units in the Hopfield model act as both input and output units.
- A Hopfield network consists of "n" totally coupled units. Each unit is connected to all other units except itself. The network is symmetric because the weight w_{ij} for the connection between unit i and unit j is equal to the weight w_{ij} of the connection from unit j to unit i. The absence of a connection from each unit to itself avoids a permanent feedback of its own state value.
- Hopfield networks are typically used for classification problems with binary pattern vectors.
- Hopfield model is classified into two categories :
  1. Discrete Hopfield Model
  2. Continuous Hopfield Model

- In both discrete and continuous Hopfield network weights trained in a one-shot fashion and not trained incrementally as was done in case of Perceptron and MLP.
- In the discrete Hopfield model, the units use a slightly modified bipolar output function where the states of the units, i.e., the output of the units remain the same if the current state is equal to some threshold value.
- The continuous Hopfield model is just a generalization of the discrete case. Here, the units use a continuous output function such as the sigmoid or hyperbolic tangent function. In the continuous Hopfield model, each unit has an associated capacitor C_i and resistance r_i that model the capacitance and resistance of real neuron's cell membrane, respectively.

2.2.4 Bidirectional Associative Memory (BAM)

- BAM consists of two layers, x and y. Signals are sent back and forth between both layers until an equilibrium is reached. Equilibrium is reached if the x and y vectors no longer change. Given an x vector the BAM is able to produce the y vector and vice versa.
- BAM consists of bi-directional edges so that information can flow in either direction. Since the BAM network has bidirectional edges, propagation moves in *both directions*, first from one layer to another, and then back to the first layer. Propagation continues until the nodes are no longer changing values.
- Fig. 2.2.7 shows BAM network.
- Since the BAM also uses the traditional Hebb's learning rule to build the connection weight matrix to store the associated pattern pairs, it too has a severely low memory capacity.

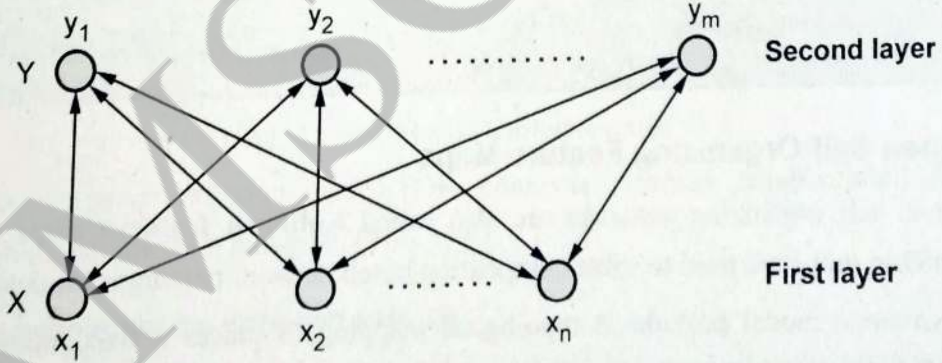


Fig. 2.2.7 BAM network

- BAM can be classified into two categories :
  1. **Discrete BAM :** The network propagates an input pattern X to the Y layer where the units in the Y layer will compute their net input.

2. **Continuous BAM :** The units use the sigmoid or hyperbolic tangent output function. The units in the X layer have an extra external input Ii, while the units in the Y layer have an extra external input J_j for i = 1, 2, ..., m and j = 1, 2, ..., n.

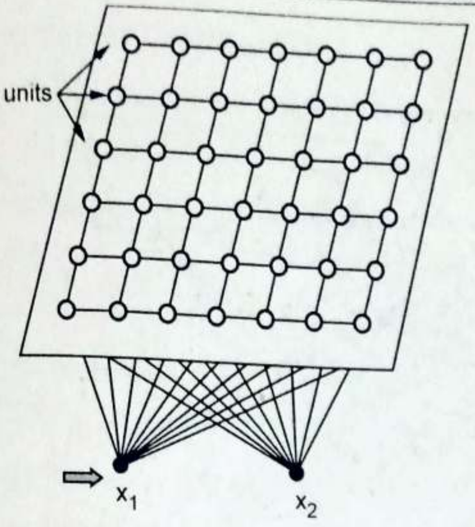
These extra external inputs lead to a modification in the computation of the net input to the units.

2.2.5 Difference Between Auto-associative Memory and Hetero-associative Memory

Auto-associative memory	Hetero-associative memory
The inputs and output vectors s and t are the same	The inputs and output vectors s and t are different.
Recalls a memory of the same modality as the one that evoked it	Recalls a memory that is different in character from the input
A picture of a favorite object might evoke a mental image of that object in vivid detail	A particular smell or sound, for example, might evoke a visual memory of some past event
An auto-associative memory retrieves the same pattern	Hetero-associative memory retrieves the stored pattern
Example : color correction, color constancy	Example : 1. Space transforms : Fourier, 2. Dimensionality reduction : PCA

2.3 Kohonen Self-Organizing Feature Maps

- Kohonen self organizing networks are also called **Kohonen features** maps or **topology preserving maps** are used to solve competition based network paradigm for data clustering.
- The Kohonen model provides a topological mapping. It places a fixed number of input patterns from the input layer into a higher-dimensional output or Kohonen layer.
- Training in the Kohonen network begins with the winner's neighbourhood of a fairly large size. Then, as training proceeds, the neighbourhood size gradually decreases.
- Fig. 2.3.1 shows a simple Kohonen self organizing network with 2 inputs and 49 outputs. The learning feature map is similar to that of competitive learning networks.



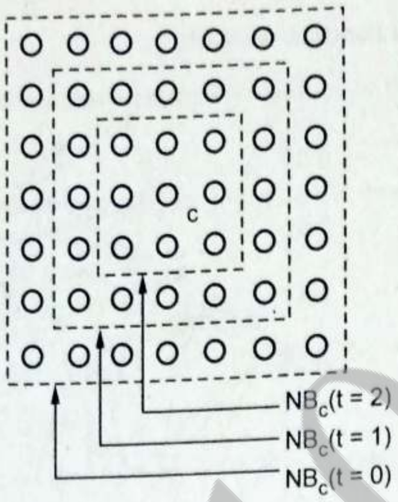


Fig. 2.3.1 Simple Kohonen self organizing network

- A similarity measure is selected and the winning unit is considered to be the one with the largest activation. For this **Kohonen features** maps all the weights in a neighborhood around the winning units are also updated. The neighborhood's size generally decreases slowly with each iteration.
- Step for how to train a Kohonen self organizing network is as follows :

For n-dimensional input space and m output neurons :

1. Choose random weight vector w_i for neuron i, $i = 1, ..., m$
2. Choose random input x
3. Determine winner neuron k : $|| w_k - x || = \min_i || w_i - x ||$ (Euclidean distance)
4. Update all weight vectors of all neurons i in the neighborhood of neuron (k : $w_i := w_i + \eta \cdot \varphi(i,k) \cdot (x - w_i)$) ($w_i$ is shifted towards x)
5. If convergence criterion met, STOP. Otherwise, narrow neighborhood function and learning parameter η and go to (2).

Competitive learning in the Kohonen network

- To illustrate competitive learning, consider the Kohonen network with 100 neurons arranged in the form of a two-dimensional lattice with 10 rows and 10 columns. The network is required to classify two-dimensional input vectors - each neuron in the network should respond only to the input vectors occurring in its region.
- The network is trained with 1000 two-dimensional input vectors generated randomly in a square region in the interval between -1 and +1. The learning rate parameter a is equal to 0.1.

1. **Initial random weights**

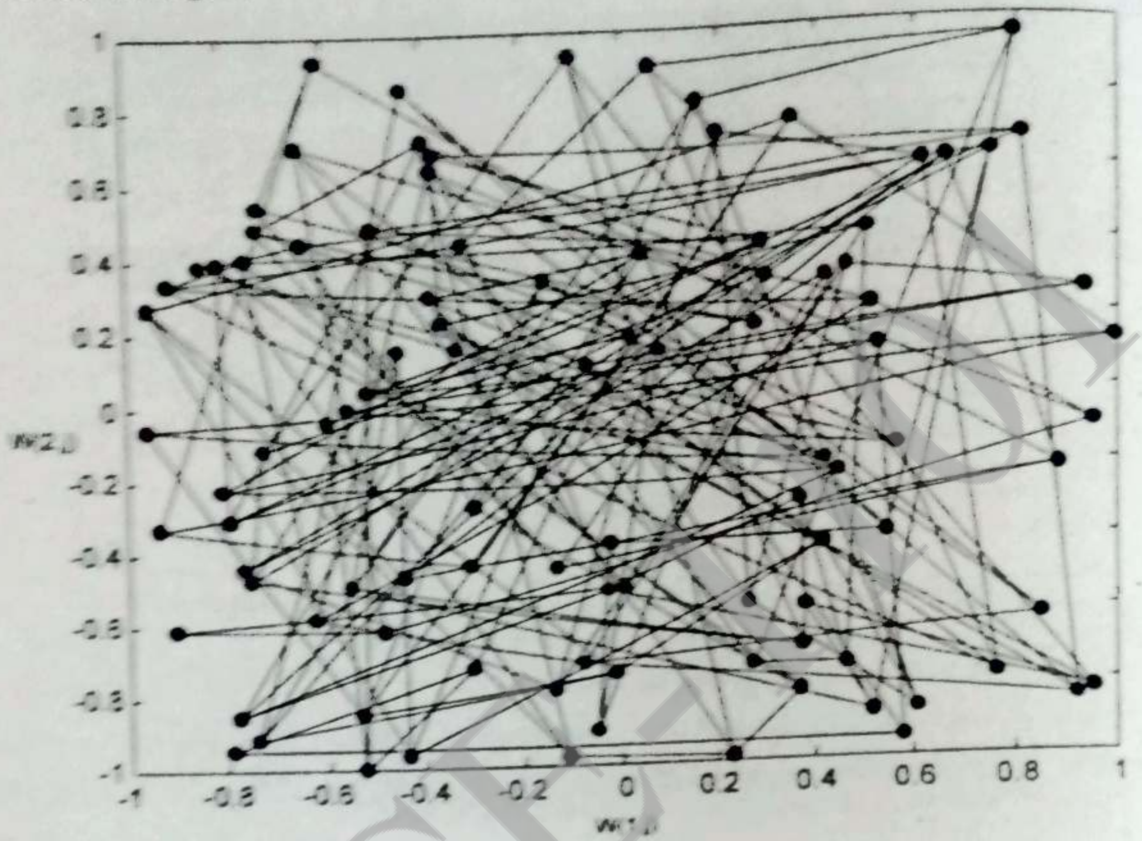
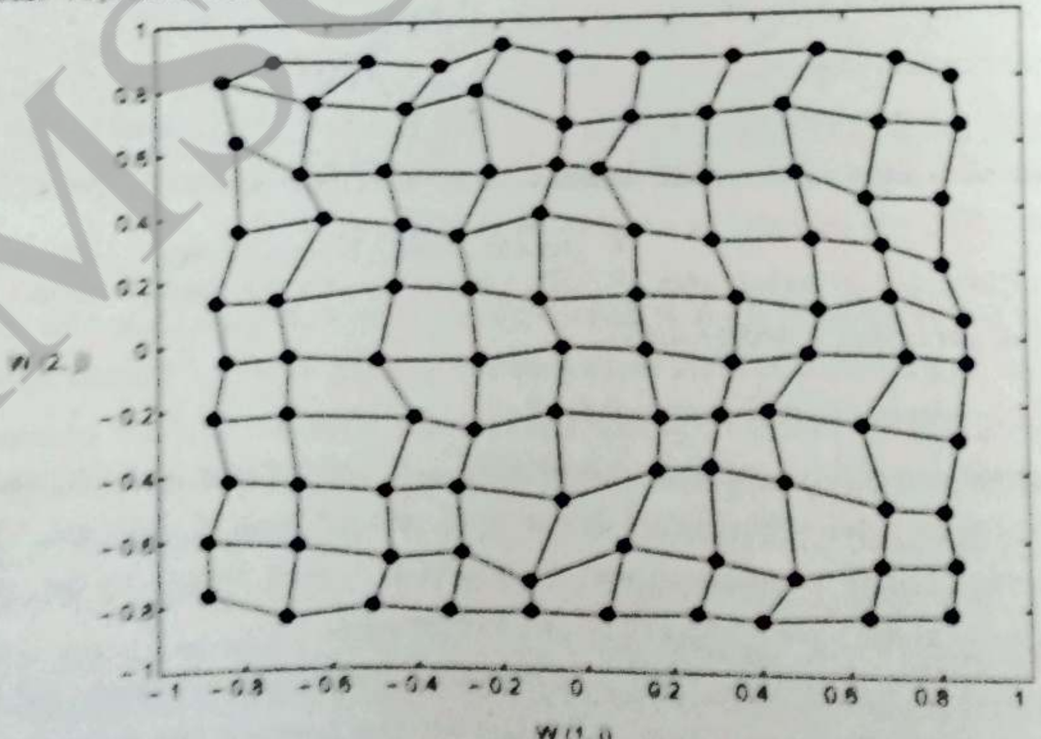


Fig. 2.3.2

2. **Network after 10,000 iterations**



2.4 Learning Vector Quantization

- Learning Vector Quantization (LVQ) is adaptive data classification method. It is based on training data with desired class information.
- LVQ uses unsupervised data clustering techniques to preprocesses the data set and obtain cluster centers.
- Fig. 2.4.1 shows the network representation of LVQ.
- Here input dimension is 2 and the input space is divided into six clusters. The first two clusters belong to class 1, while other four clusters belong to class 2.
- THE LVQ learning algorithm involves two steps :
  1. An unsupervised learning data clustering method is used to locate several cluster centers without using the class information.
  2. The class information is used to fine tune the cluster centers to minimize the number of misclassified cases.
- The number of cluster can either be specified a priori or determined via a cluster technique capable of adaptively adding new clusters when necessary. Once the clusters are obtained, their classes must be labeled before moving to second step. Such labeling is a achieved by **voting method**.

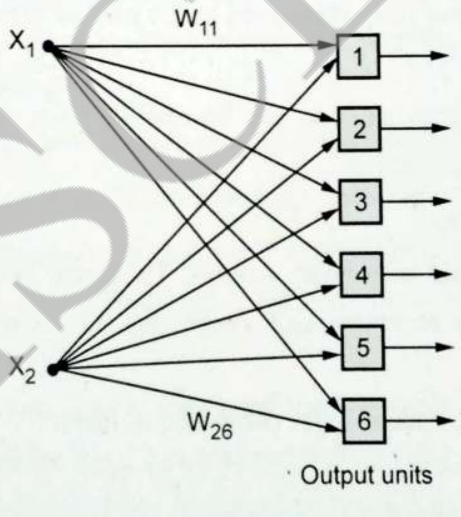


Fig. 2.4.1 LVQ

➤ **Learning method :**

- The weight vector (w) that is closest to the input vector (x) must be found. If x belongs to the same class, we move w towards x; otherwise we move w away from the input vector x.
  - **Step 1 : Initialize the cluster centers by a clustering method.**
  - **Step 2 : Label each cluster by the voting method.**

- **Step 3 : Randomly select a training input vector x and find k such that || x – w_k || is a minimum.**
- **Step 4 : If x and w_k belongs to the same class, update w_k by**

$$\Delta W_k = N(x - W_k)$$

Otherwise update w_k by

$$\Delta W_k = -\eta (X - W_k)$$

- **Step 5 : If the maximum number of iterations is reached, stop. Otherwise return to step 3.**

2.5 Counter Propagation Networks

- The counter propagation network is a hybrid network. It consists of an outstar network and a competitive filter network. It was developed in 1986 by Robert Hecht-Nielsen.
- Counter propagation networks multilayer networks based on a combination of input, clustering and output layers. This network can be used to compress data, to approximate functions or to associate patterns.
- CPN is an unsupervised winner-take-all competitive learning network.
- The hidden layer is a Kohonen network with unsupervised learning and the output layer is a Grossberg (outstar) layer fully connected to the hidden layer. The output layer is trained by the Widrow-Hoff rule.
- The counter propagation network can be applied in a data compression approximation functions or pattern association.
- Three major components :
  1. Instar : Hidden node with input weights. The instar is a single processing element that shares its general structure and processing functions with many other processing elements
  2. Competitive layer : Hidden layer composed of instars
  3. Outstar : A structure
- Counter propagation networks training include two stages :
  1. Input vectors are clustered. Clusters are formed using dot product metric or Euclidean norm metrics.
  2. Weights from cluster units to outputs units are made to produce the desired response.
- **Counter Propagation Operation :**
  1. Present input to network
  2. Calculate output of all neurons in Kohonen layer

3. Determine winner (neuron with maximum output)
4. Set output of winner to 1 (others to 0)
5. Calculate output vector

- Counter propagation networks are of two types :
  1. Full counter propagation
  2. Forward counter propagation

1. Full counter-propagation network (CPN).

- The Full CPN allows to produce a correct output even when it is given an input vector that is partially incomplete or incorrect.
- Full counter-propagation was developed to provide an efficient method of representing a large number of vector pairs, x : y by adaptively constructing a lookup table.
- It produces an approximation x* : y* based on input of an x vector or input of a y vector only, or input of an x:y pair, possibly with some distorted or missing elements in either or both vectors.
- In first phase, the training vector pairs are used to form clusters using either dot product or euclidean distance. If dot product is used, normalization is a must.
- This phase of training is called as In star modeled training. The active units here are the units in the x-input, z-cluster and y-input layers. The winning unit uses standard Kohonen learning rule for its weigh updation.
- During second phase, the weights are adjusted between the cluster units and output units.
- In this phase, we can find only the J unit remaining active in the cluster layer.
- The weights from the winning cluster unit J to the output units are adjusted, so that vector of activation of units in the y output layer, y*, is approximation of input vector y; and x* is an approximation of input vector x.
- The architecture of CPN resembles an instar and outstar model.
- The model which connects the input layers to the hidden layer is called Instar model and the model which connects the hidden layer to the output layer is called Outstar model.
- The weights are updated in both the Instar (in first phase) and Outstar model (second phase). The network is fully interconnected network

2. Forward Counter Propagation Network :

- It may be used if the mapping from x to y is well defined, but the mapping from y to x is not. In this network, after competition only one unit in that layer will be active and send a signal to the output layer.

- **Possible drawback of counter propagation networks :**
  1. Training a counter propagation network has the same difficulty associated with training a Kohonen network.
  2. Counter propagation networks tend to be larger than back propagation networks. If a certain number of mappings are to be learned, the middle layer must have that many numbers of neurons.

2.6 Adaptive Resonance Theory Network

- Gail Carpenter and Stephen Grossberg (Boston University) developed the Adaptive Resonance learning model. How can a system retain its previously learned knowledge while incorporating new information.
- Adaptive resonance architectures are artificial neural networks that are capable of stable categorization of an arbitrary sequence of unlabeled input patterns in real time. These architectures are capable of continuous training with non-stationary inputs.
- Some models of Adaptive Resonance Theory are :
  1. ART1 - Discrete input.
  2. ART2 - Continuous input.
  3. ARTMAP - Using two input vectors, transforms the unsupervised ART model into a supervised one.
- Various others : Fuzzy ART, Fuzzy ARTMAP (FARTMAP), etc…
- The primary intuition behind the ART model is that object identification and recognition generally occur as a result of the interaction of 'top-down' observer expectations with 'bottom-up' sensory information.
- The basic ART system is an **unsupervised learning model**. It typically consists of a comparison field and a recognition field composed of neurons, a vigilance parameter, and a reset module. However, ART networks are able to grow additional neurons if a new input cannot be categorized appropriately with the existing neurons.
- ART networks tackle the stability-plasticity dilemma :
  1. **Plasticity :** They can always adapt to unknown inputs if the given input cannot be classified by existing clusters.
  2. **Stability :** Existing clusters are not deleted by the introduction of new inputs.
  3. **Problem :** Clusters are of fixed size, depending on ρ.
- Fig. 2.6.1 shows ART-1 Network.

- ART-1 networks, which receive binary input vectors. Bottom-up weights are used to determine output-layer candidates that may best match the current input.
- Top-down weights represent the "prototype" for the cluster defined by each output neuron. A close match between input and prototype is necessary for categorizing the input.

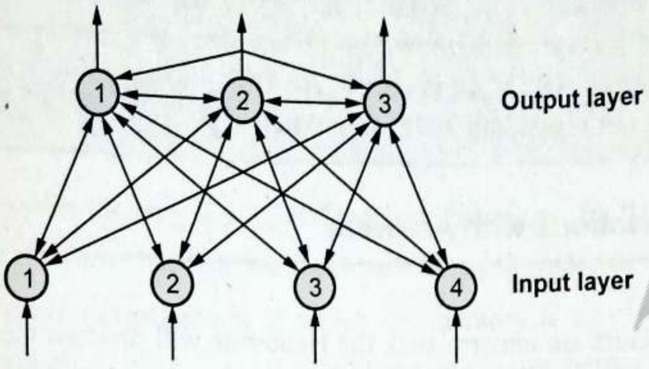


Fig. 2.6.1 ART 1 network

- Finding this match can require multiple signal exchanges between the two layers in both directions until "resonance" is established or a new neuron is added.
- The basic ART model, ART1, is comprised of the following components :
  1. The short term memory layer : F1 - Short term memory.
  2. The recognition layer : F2 - Contains the long term memory of the system.
  3. Vigilance Parameter : ρ - A parameter that controls the generality of the memory. Larger ρ means more detailed memories, smaller ρ produces more general memories.

Types of ART :

Type	Remarks
ART 1	It is the simplest variety of ART networks, accepting only binary inputs.
ART 2	Extends network capabilities to support continuous inputs.
ART 3	ART 3 builds on ART-2 by simulating rudimentary neurotransmitter regulation of synaptic activity by incorporating simulated sodium (Na+) and calcium (Ca2+) ion concentrations into the system's equations, which results in a more physiologically realistic means of partially inhibiting categories that trigger mismatch resets.
Fuzzy ART	Fuzzy ART implements fuzzy logic into ART's pattern recognition, thus enhancing generalizability

ARTMAP	It is also known as Predictive ART, combines two slightly modified ART-1 or ART-2 units into a supervised learning structure where the first unit takes the input data and the second unit takes the correct output data, then used to make the minimum possible adjustment of the vigilance parameter in the first unit in order to make the correct classification.
Fuzzy ARTMAP	Fuzzy ARTMAP is merely ARTMAP using fuzzy ART units, resulting in a corresponding increase in efficiency.

2.7 Two Marks Questions with Answers

Q.1 What is recall ?

Ans. : If the input vectors are uncorrelated, the Hebb rule will produce the correct weights and the response of the net when tested with one of the training vectors will be perfect recall

Q.2 Explain learning vector quantization.

Ans. : LVQ is adaptive data classification method. It is based on training data with desired class information. LVQ uses unsupervised data clustering techniques to preprocesses the data set and obtain cluster centers.

Q.3 What is meant by associative memory ?

Ans. : An associative memory can be considered as a memory unit whose stored data can be identified for access by the content of the data itself rather than by an address or memory location. Associative memory is often referred to as Content Addressable Memory (CAM).

Q.4 Define auto associative memory.

Ans. : This is a single layer neural network in which the input training vector and the output target vectors are the same. The weights are determined so that the network stores a set of patterns. If vector "t" is the same as " s", the net is auto-associative.

Q.5 What is Hebbian learning ?

Ans. : Hebb rule is the simplest and most common method of determining weights for an associative memory neural net. It can be used with patterns are represented as either binary or bipolar vectors.

Q.6 What is Bidirectional Associative Memory (BAM) ?

Ans. : Bidirectional associative memory first proposed by **Bart Kosko**, is a hetero-associative network. It associates patterns from one set, set A, to patterns from another set, set B and vice versa. Like a Hopfield network, the BAM can generalize and also produce correct outputs despite corrupted or incomplete inputs.

Q.7 List the problems of BAM network.

Ans. :

1. Storage capacity of the BAM : The maximum number of associations to be stored in the BAM should not exceed the number of neurons in the smaller layer.
2. Incorrect convergence : The BAM may not always produce the closest association

Q.8 What is content-addressable memory ?

Ans. :

- A content-addressable memory is a type of memory that allows for the recall of data based on the degree of similarity between the input pattern and the patterns stored in memory.
- It refers to a memory organization in which the memory is accessed by its content as opposed to an explicit address like in the traditional computer memory system.
- Therefore, this type of memory allows the recall of information based on partial knowledge of its contents

Q.9 What are the delta rule for pattern association ?

Ans. :

- When the input vectors are linearly independent, the delta rule produces exact solutions.
- Whether the input vectors are linearly independent or not, the delta rule produces a least squares solution, i.e., it optimizes for the lowest sum of least squared errors.

Q.10 What is continuous BAM ?

Ans. : Continuous BAM transforms input smoothly and continuously into output in the range [0, 1] using the logistic sigmoid function as the activation function for all units.

Q.11 What are the delta rule for pattern association ?

Ans. :

- When the input vectors are linearly independent, the delta rule produces exact solutions.
- Whether the input vectors are linearly independent or not, the delta rule produces a least squares solution, i.e., it optimizes for the lowest sum of least squared errors.

Q.12 Which are the rules used in Hebb's law ?

Ans. : Rules :

1. If two neurons on either side of a connection are activated synchronously, then the weight of that connection is increased.
2. If two neurons on either side of a connection are activated asynchronously, then the weight of that connection is decreased.

Q.13 What do you mean counter propagation network ?

Ans. : Counter propagation networks multilayer networks based on a combination of input, clustering and output layers. This network can be used to compress data, to approximate functions or to associate patterns.

Q.14 What is Hopfield model ?

Ans. : The **Hopfield** model is a single-layered recurrent network. Like the associative memory, it is usually **initialized** with appropriate weights instead of being trained.

Q.15 Define Self-Organizing Map.

Ans. : The Self-Organizing Map is one of the most popular neural network models. It belongs to the category of competitive learning networks. The Self-Organizing Map is based on unsupervised learning, which means that no human intervention is needed during the learning and that little needs to be known about the characteristics of the input data.

Q.16 What is principle goal of the self-organizing map ?

Ans. : The principal goal of the Self-Organizing Map (SOM) is to transform an incoming signal pattern of arbitrary dimension into a one - or two-dimensional discrete map and to perform this transformation adaptively in a topologically ordered fashion.

Q.17 List the stages of the SOM algorithm.

Ans. :

1. Initialization - Choose random values for the initial weight vectors wj.
2. Sampling - Draw a sample training input vector x from the input space.
3. Matching - Find the winning neuron I(x) with weight vector closest to input vector.

$$\Delta w_{ji} = \eta(t)\, T_{jI(x)}(t)\, (x_i - w_{ji})$$

4. Updating - Apply the weight update equation
5. Continuation - Keep returning to step 2 until the feature map stops changing.

Q.18 Explain an essential ingredients and parameters of the SOM algorithm.

Ans. : An essential ingredients and parameters of the SOM algorithm are as follows :

1. Continuous input space of activation patterns that are generated in accordance with a certain probability distribution;
2. Topology of the network in the form of a lattice of neurons, which defines a discrete output space;
3. Time - varying neighborhood function hj, i(x)(n) that is defined around a winning neuron i(x);
4. Learning - rate parameter that starts at an initial value and then decreases gradually with time, but never goes to zero.

Q.19 How does counter-propagation nets are trained ?

Ans. : Counter-propagation nets are trained in two stages :

1. First stage : The input vectors are clustered. The clusters that are formed may be based on either the dot product metric or the Euclidean norm metric.
2. Second stage : The weights from the cluster units to the output units are adapted to produce the desired response.

Q.20 List the possible drawback of counter-propagation networks.

Ans. :

- Training a counter-propagation network has the same difficulty associated with training a Kohonen network.
- Counter-propagation networks tend to be larger than backpropagation networks. If a certain number of mappings are to be learned, the middle layer must have that many number of neurons.

Q.21 How forward-only differs form full counter-propagation nets ?

Ans. :

- In full counter-propagation, only the x vectors to form the clusters on the Kohonen units during the first stage of training.
- The original presentation of forward-only counter-propagation used the Euclidean distance between the input vector and the weight vector for the Kohonen unit.

Q.22 What is forward only counter-propagation ?

Ans. :

- Is a simplified version of the full counterpropagation
- Are intended to approximate $y = f(x)$ function that is not necessarily invertible.
- It may be used if the mapping from x to y is well defined, but the mapping from y to x is not.

Q.23 Define plasticity.

Ans. : The ability of a net to respond to learn a new pattern equally well at any stage of learning is called plasticity.

Q.24 List the components of ART1.

Ans. : Components are as follows :

1. The short term memory layer (F1)
2. The recognition layer (F2) : It contains the long term memory of the system.
3. Vigilance Parameter (ρ) : A parameter that controls the generality of the memory. Larger ρ means more detailed memories, smaller ρ produces more general memories.

❑❑❑

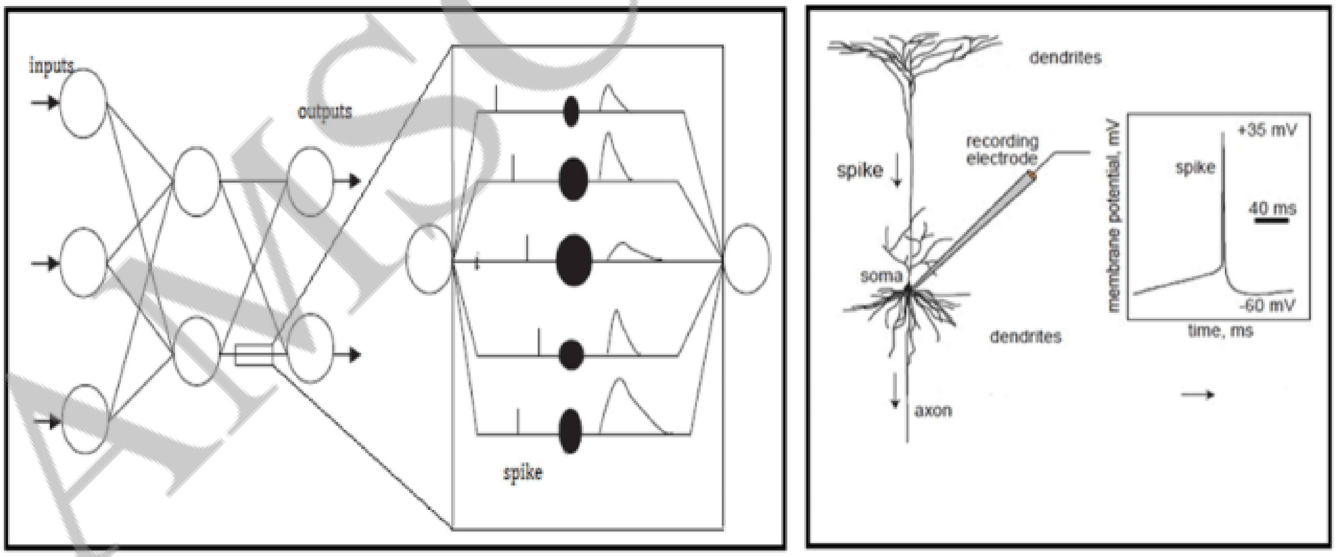
Spiking Neural Networks-Convolutional Neural Networks-Deep Learning Neural Networks-Extreme Learning Machine Model-Convolutional Neural Networks: The Convolution Operation – Motivation – Pooling – Variants of the basic Convolution Function – Structured Outputs – Data Types – Efficient Convolution Algorithms – Neuroscientific Basis – Applications: Computer Vision, Image Generation, Image Compression.

Spiking Neural Networks

What is Spiking Neural Network (SNN)?

Artificial neural networks that closely mimic natural neural networks are known as spiking neural networks (SNNs). In addition to neuronal and synaptic status, SNNs incorporate time into their working model. The idea is that neurons in the SNN do not transmit information at the end of each propagation cycle (as they do in traditional multi-layer perceptron networks), but only when a membrane potential – a neuron’s intrinsic quality related to its membrane electrical charge – reaches a certain value, known as the threshold.

The neuron fires when the membrane potential hits the threshold, sending a signal to neighboring neurons, which increase or decrease their potentials in response to the signal. A spiking neuron model is a neuron model that fires at the moment of threshold crossing.



SNN with connections and Biological Neuron

Artificial neurons, despite their striking resemblance to biological neurons, do not behave in the same way. Biological and artificial NNs differ fundamentally in the following ways:

- Structure in general
- Computations in the brain
- In comparison to the brain, learning is a rule.

Alan Hodgkin and Andrew Huxley created the first scientific model of a Spiking Neural Network in 1952. The model characterized the initialization and propagation of action potentials in biological neurons. Biological neurons, on the other hand, do not transfer impulses directly. In order to communicate, chemicals called neurotransmitters must be exchanged in the synaptic gap.

How Does Spiking Neural Network Work?

Key Concepts

What distinguishes a traditional ANN from an SNN is the information propagation approach. SNN aspires to be as close to a biological neural network as feasible. As a result, rather than working with continually changing time values as ANN does, SNN works with discrete events that happen at defined times. SNN takes a set of spikes as input and produces a set of spikes as output (a series of spikes is usually referred to as spike trains).

The general idea is as;

- Each neuron has a value that is equivalent to the electrical potential of biological neurons at any given time.
- The value of a neuron can change according to its mathematical model; for example, if a neuron gets a spike from an upstream neuron, its value may rise or fall.
- If a neuron's value surpasses a certain threshold, the neuron will send a single impulse to each downstream neuron connected to the first one, and the neuron's value will immediately drop below its average.
- As a result, the neuron will go through a refractory period similar to that of a biological neuron. The neuron's value will gradually return to its average over time.

Spike Based Neural Codes

Artificial spiking neural networks are designed to do neural computation. This necessitates that neural spiking is given meaning: the variables important to the computation must be defined in terms of the spikes with which spiking neurons communicate. A variety of neuronal information encodings have been proposed based on biological knowledge:

- ***Binary Coding:***

Binary coding is an all-or-nothing encoding in which a neuron is either active or inactive within a specific time interval, firing one or more spikes throughout that time frame. The finding that physiological neurons tend to activate when they receive input (a sensory stimulus such as light or external electrical inputs) encouraged this encoding.

- ***Rate Coding:***

Only the rate of spikes in an interval is employed as a metric for the information communicated in rate coding, which is an abstraction from the timed nature of spikes. The fact that physiological neurons fire more frequently for stronger (sensory or artificial) stimuli motivates rate encoding.

- ***Fully Temporal Codes***

The encoding of a fully temporal code is dependent on the precise timing of all spikes. Evidence from neuroscience suggests that spike-timing can be incredibly precise and repeatable. Timings are related to a certain (internal or external) event in a fully temporal code (such as the onset of a stimulus or spike of a reference neuron).

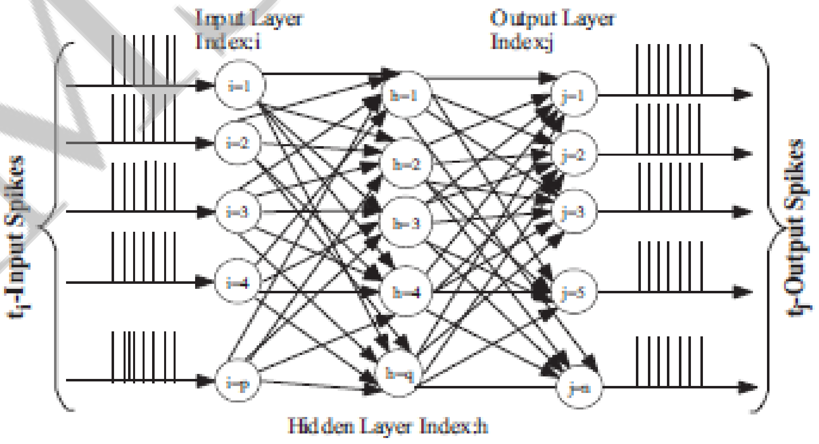
- ***Latency Coding***

The timing of spikes is used in latency coding, but not the number of spikes. The latency between a specific (internal or external) event and the first spike is used to encode information. This is based on the finding that significant sensory events cause upstream neurons to spike earlier.

SNN Architecture

Spiking neurons and linking synapses are described by configurable scalar weights in an SNN architecture. The analogue input data is encoded into the spike trains using either a rate-based technique, some sort of temporal coding or population coding as the initial stage in building an SNN.

A biological neuron in the brain (and a simulated spiking neuron) gets synaptic inputs from other neurons in the neural network, as previously explained. Both action potential production and network dynamics are present in biological brain networks.



Architecture of a multilayer spiking neural network.

The network dynamics of artificial SNNs are much simplified as compared to actual biological networks. It is useful in this context to suppose that the modelled spiking neurons have pure threshold dynamics (as opposed to refractoriness, hysteresis, resonance dynamics, or post-inhibitory rebound features).

When the membrane potential of postsynaptic neurons reaches a threshold, the activity of presynaptic neurons affects the membrane potential of postsynaptic neurons, resulting in an action potential or spike.

Learning Rules in SNN's

Learning is achieved in practically all ANNs, spiking or non-spiking, by altering scalar-valued synaptic weights. Spiking allows for the replication of a form of bio-plausible learning rule that is not possible in non-spiking networks. Many variations of this learning rule have been uncovered by neuroscientists under the umbrella term spike-timing-dependent plasticity (STDP).

Its main feature is that the weight (synaptic efficacy) connecting a pre-and post-synaptic neuron is altered based on their relative spike times within tens of millisecond time intervals. The weight adjustment is based on information that is both local to the synapse and local in time. The next subsections cover both unsupervised and supervised learning techniques in SNNs.

Application of Spiking Neural Networks

In theory, SNNs can be used in the same applications as standard ANNs. SNNs can also stimulate the central nervous systems of biological animals, such as an insect seeking food in an unfamiliar environment. They can be used to examine the operation of biological brain networks due to their realism.

Advantages and Disadvantages of SNN

Advantages

- SNN is a dynamic system. As a result, it excels in dynamic processes like speech and dynamic picture identification.
- When an SNN is already working, it can still train.
- To train an SNN, you simply need to train the output neurons.
- Traditional ANNs often have more neurons than SNNs; however, SNNs typically have fewer neurons.
- Because the neurons send impulses rather than a continuous value, SNNs can work incredibly quickly.
- Because they leverage the temporal presentation of information, SNNs have boosted information processing productivity and noise immunity.

Disadvantages

- SNNs are difficult to train.
- As of now, there is no learning algorithm built expressly for this task.

- Building a small SNN is impracticable.

Convolutional Neural Networks

What is a Neural Network?

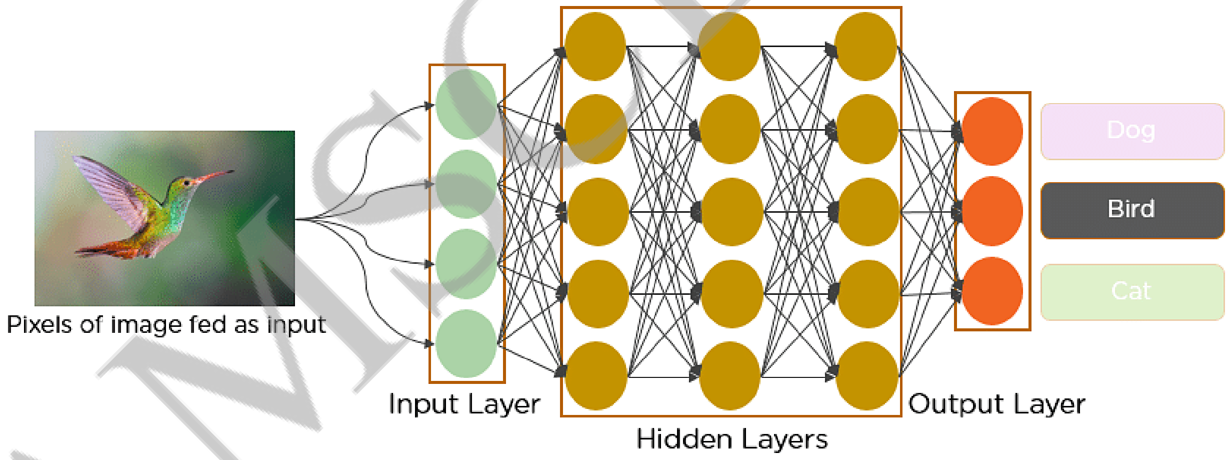
Neural networks are modeled after our brains. There are individual nodes that form the layers in the network, just like the neurons in our brains connect different areas.

Neural network with multiple hidden layers. Each layer has multiple nodes.

The inputs to nodes in a single layer will have a weight assigned to them that changes the effect that parameter has on the overall prediction result. Since the weights are assigned on the links between nodes, each node maybe influenced by multiple weights.

The neural network takes all of the training data in the input layer. Then it passes the data through the hidden layers, transforming the values based on the weights at each node. Finally it returns a value in the output layer.

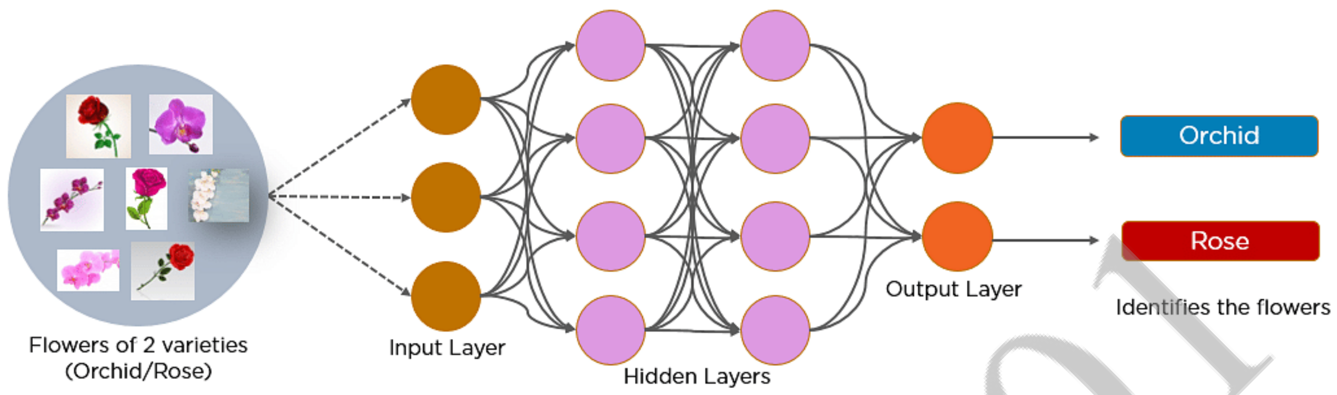
It can take some time to properly tune a neural network to get consistent, reliable results. Testing and training your neural network is a balancing process between deciding what features are the most important to your model.



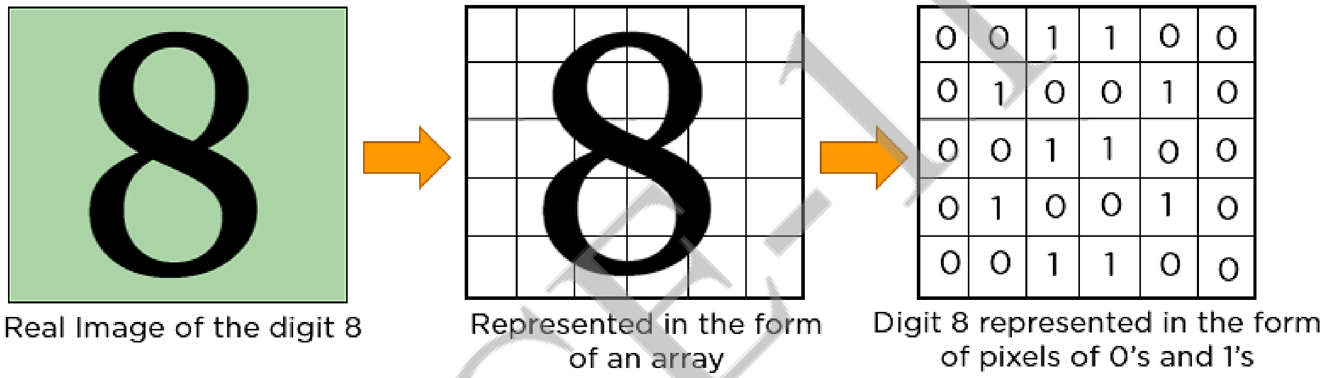
What is Convolutional Neural Network?

A convolutional neural network is a feed-forward neural network that is generally used to analyze visual images by processing data with grid-like topology. It's also known as a ConvNet. A convolutional neural network is used to detect and classify objects in an image.

Below is a neural network that identifies two types of flowers: Orchid and Rose.



In CNN, every image is represented in the form of an array of pixel values.


Real Image of the digit 8
Represented in the form of an array
Digit 8 represented in the form of pixels of 0's and 1's

0	0	1	1	0	0
0	1	0	0	1	0
0	0	1	1	0	0
0	1	0	0	1	0
0	0	1	1	0	0

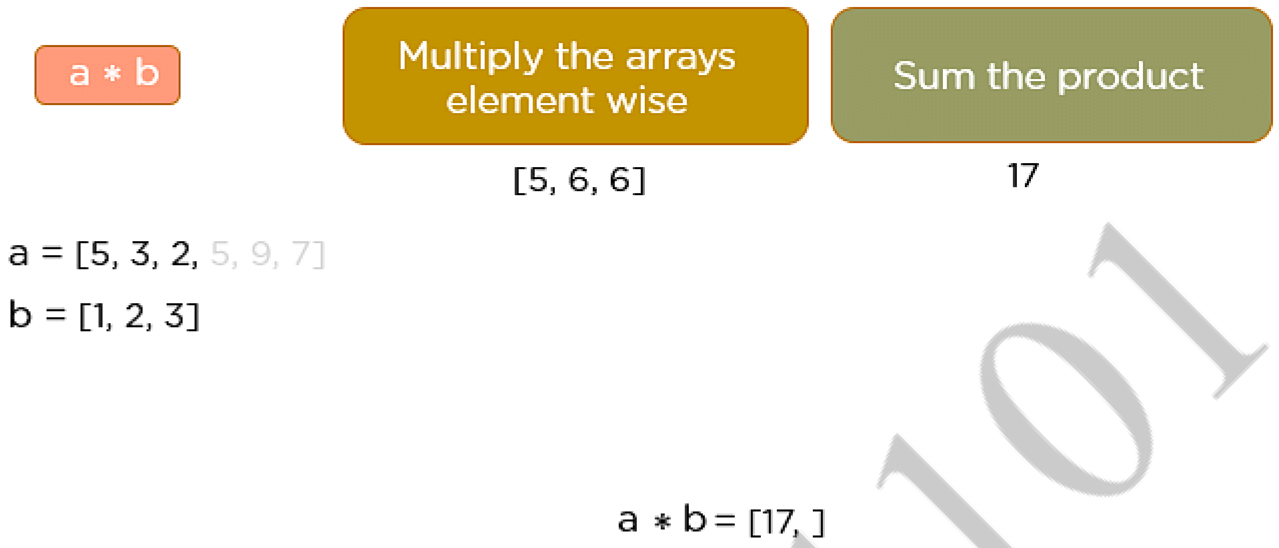
The convolution operation forms the basis of any convolutional neural network. Let's understand the convolution operation using two matrices, a and b, of 1 dimension.

a = [5,3,7,5,9,7]

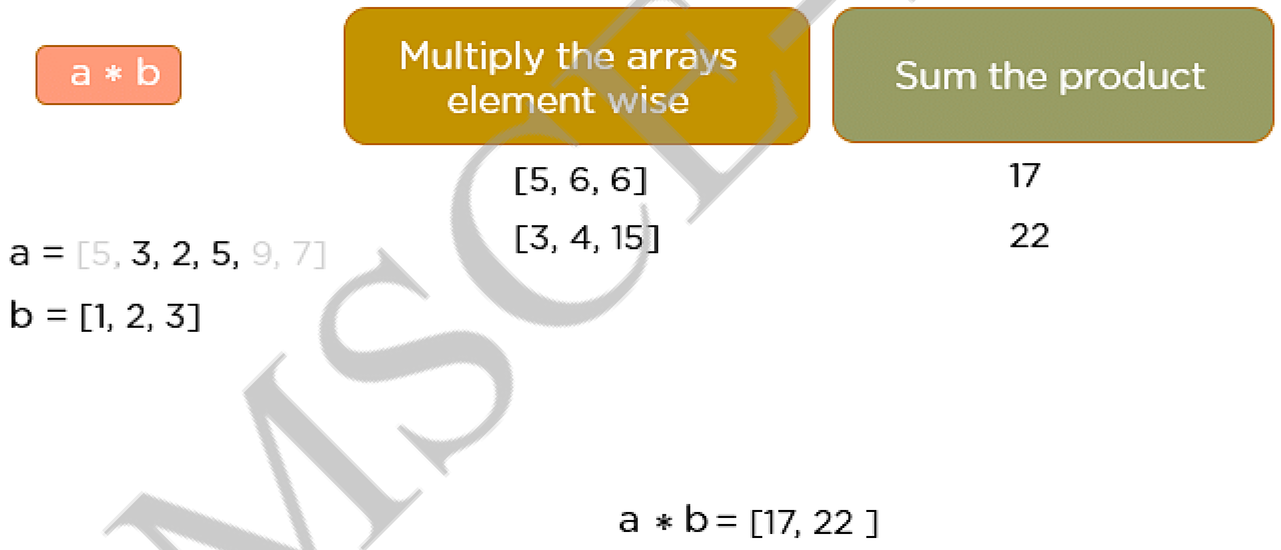
b = [1,2,3]

In convolution operation, the arrays are multiplied element-wise, and the product is summed to create a new array, which represents a*b.

The first three elements of the matrix a are multiplied with the elements of matrix b. The product is summed to get the result.



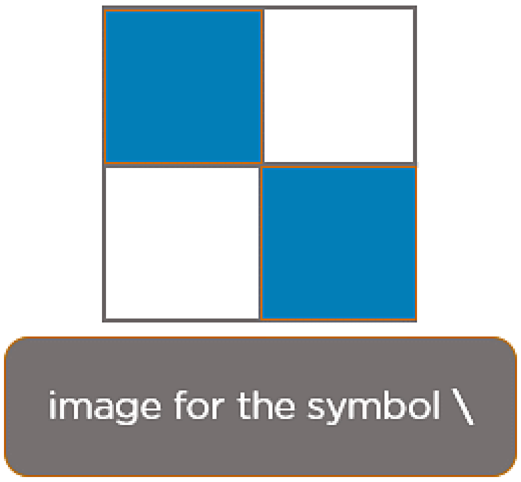
The next three elements from the matrix a are multiplied by the elements in matrix b, and the product is summed up.

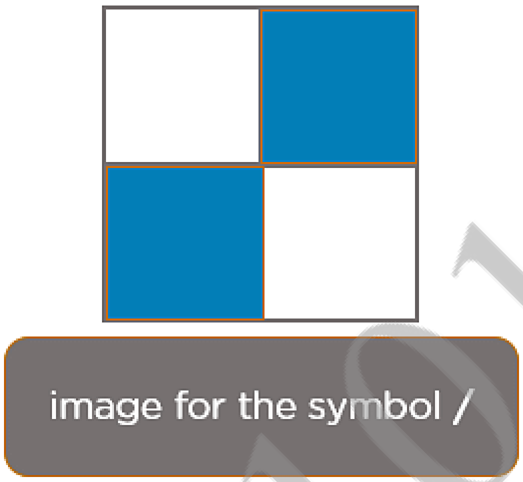


This process continues until the convolution operation is complete.

How Does CNN Recognize Images?

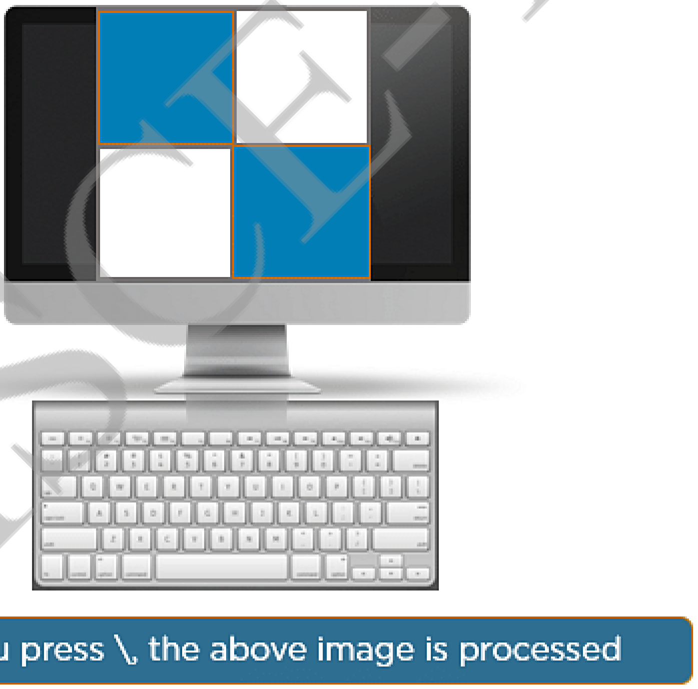
Consider the following images:



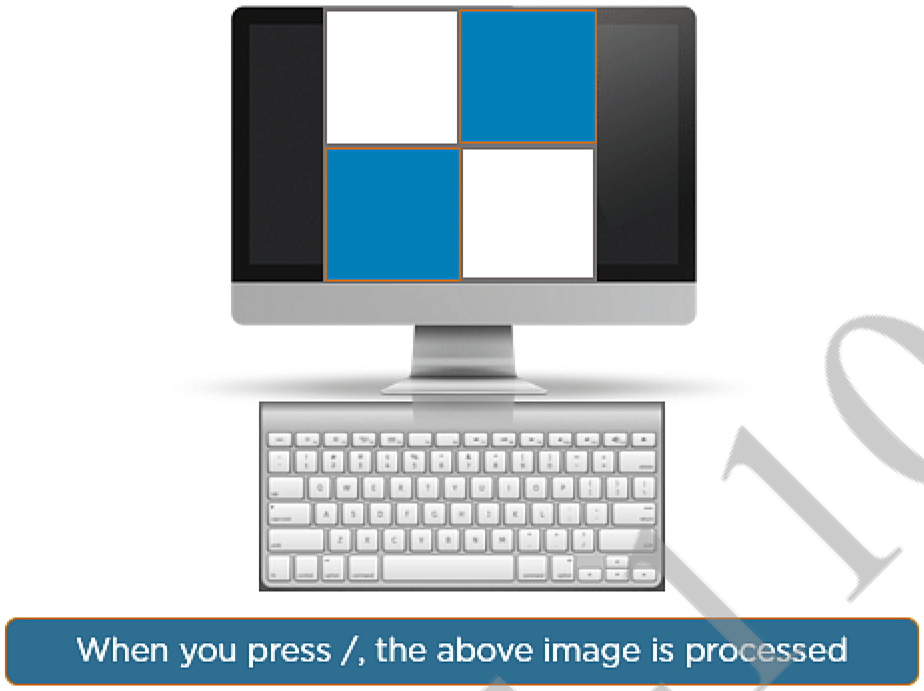


The boxes that are colored represent a pixel value of 1, and 0 if not colored.

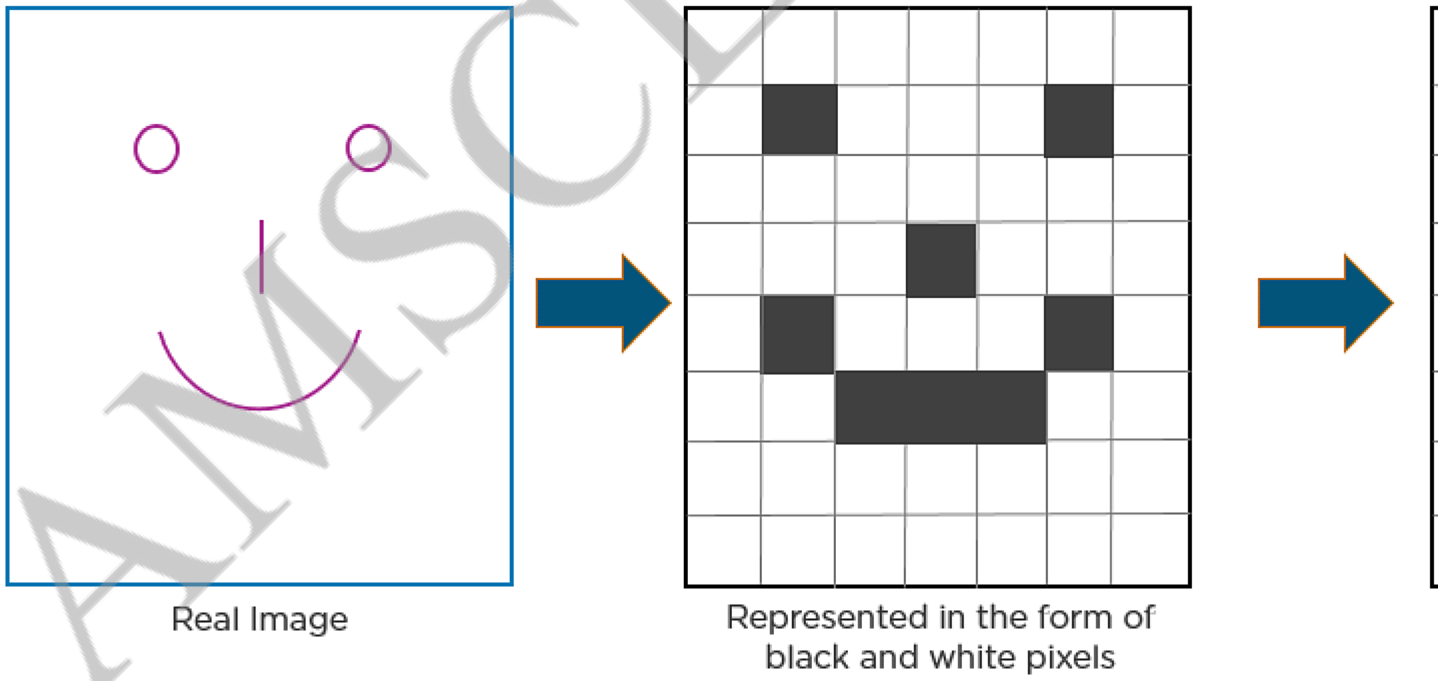
When you press backslash (\), the below image gets processed.



When you press forward-slash (/), the below image is processed:



Here is another example to depict how CNN recognizes an image:



As you can see from the above diagram, only those values are lit that have a value of 1.

Layers in a Convolutional Neural Network

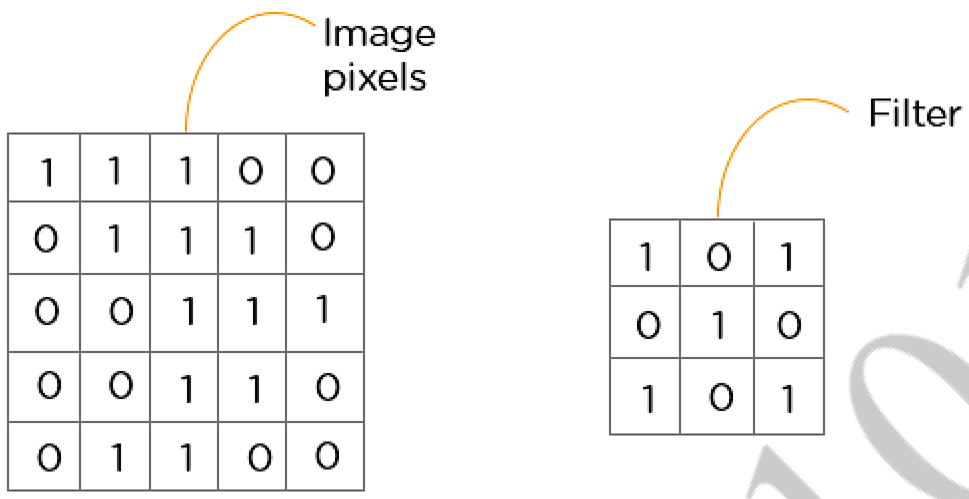
A convolution neural network has multiple hidden layers that help in extracting information from an image. The four important layers in CNN are:

1. Convolution layer
2. ReLU layer
3. Pooling layer
4. Fully connected layer

Convolution Layer

This is the first step in the process of extracting valuable features from an image. A convolution layer has several filters that perform the convolution operation. Every image is considered as a matrix of pixel values.

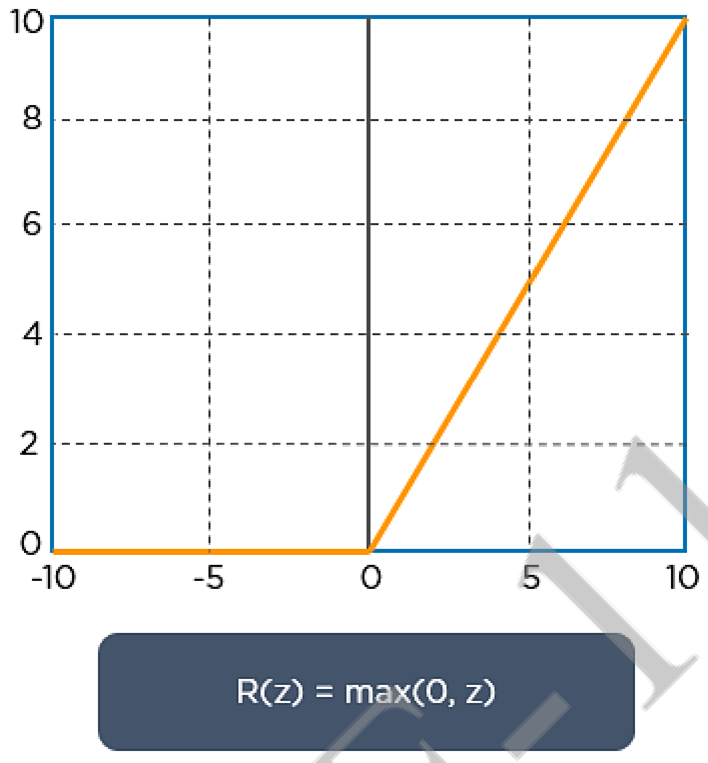
Consider the following 5x5 image whose pixel values are either 0 or 1. There's also a filter matrix with a dimension of 3x3. Slide the filter matrix over the image and compute the dot product to get the convolved feature matrix.



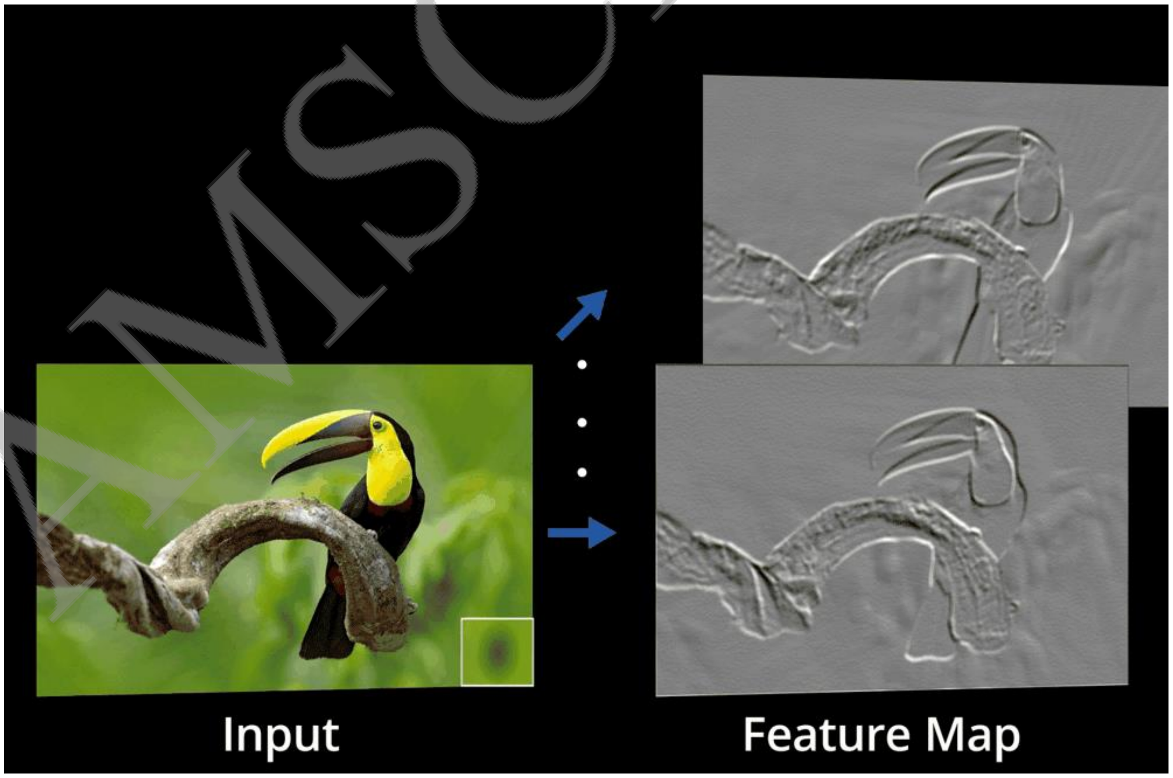
ReLU layer

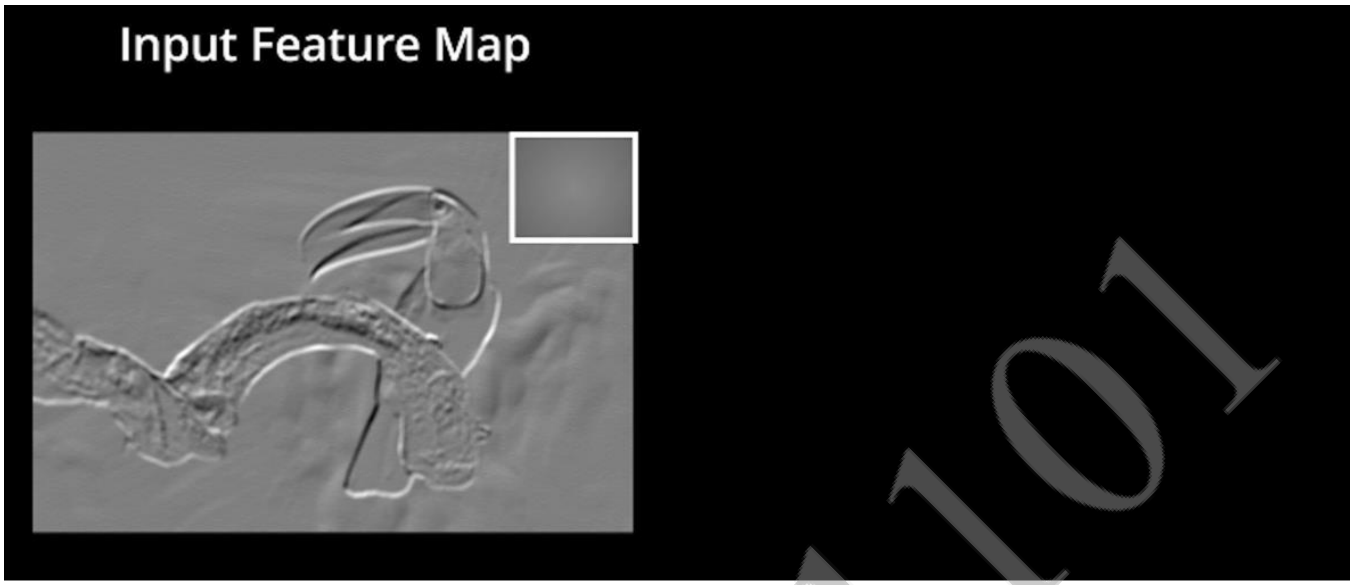
ReLU stands for the rectified linear unit. Once the feature maps are extracted, the next step is to move them to a ReLU layer.

ReLU performs an element-wise operation and sets all the negative pixels to 0. It introduces non-linearity to the network, and the generated output is a rectified feature map. Below is the graph of a ReLU function:



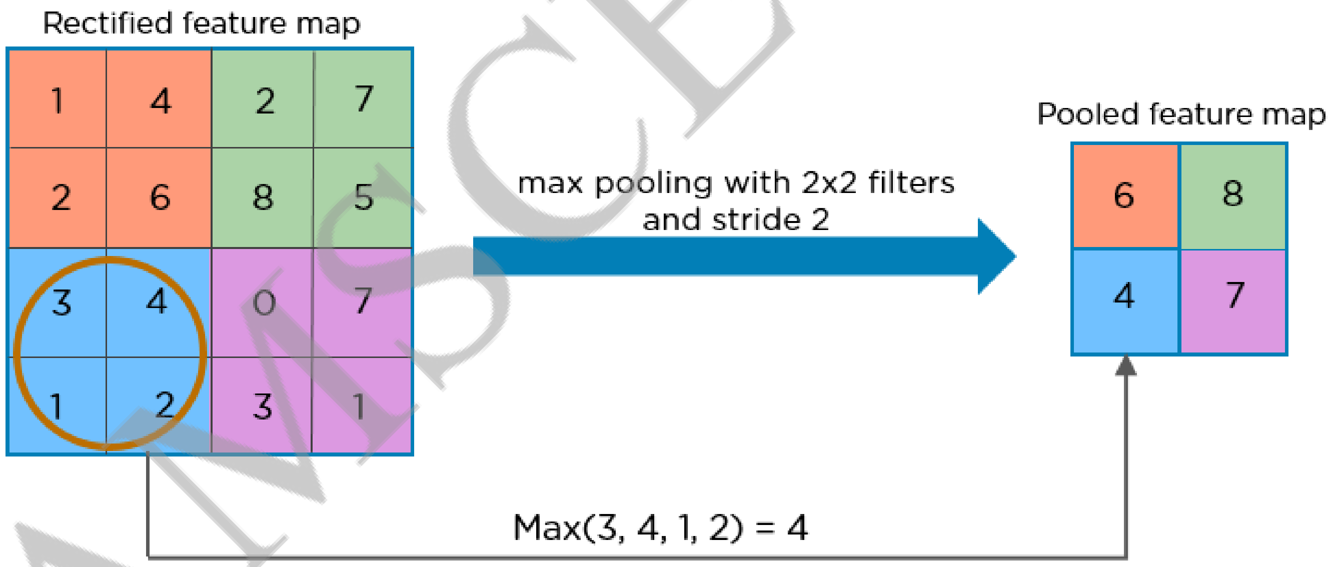
The original image is scanned with multiple convolutions and ReLU layers for locating the features.



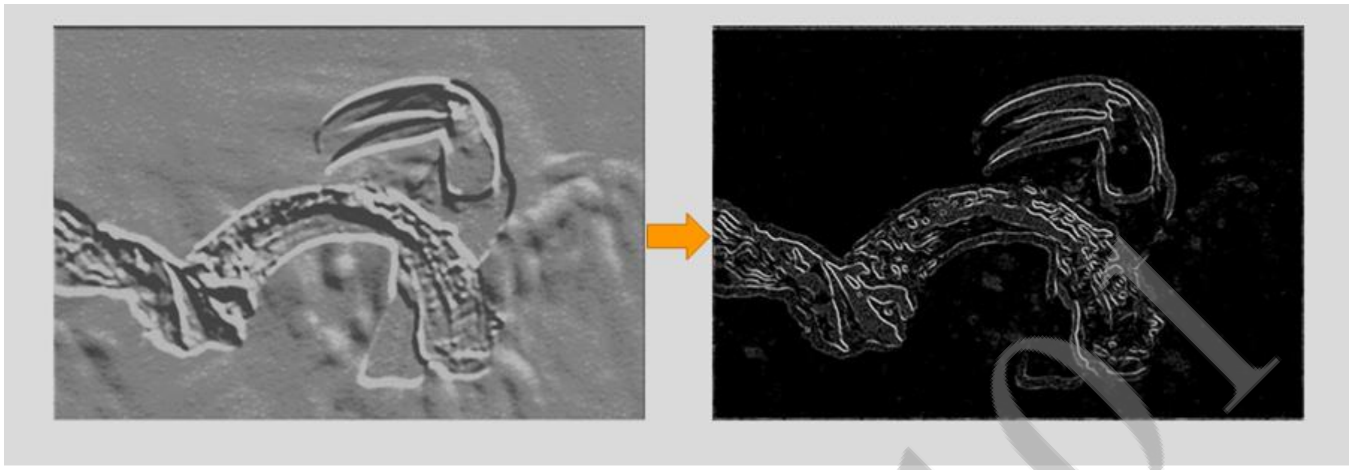


Pooling Layer

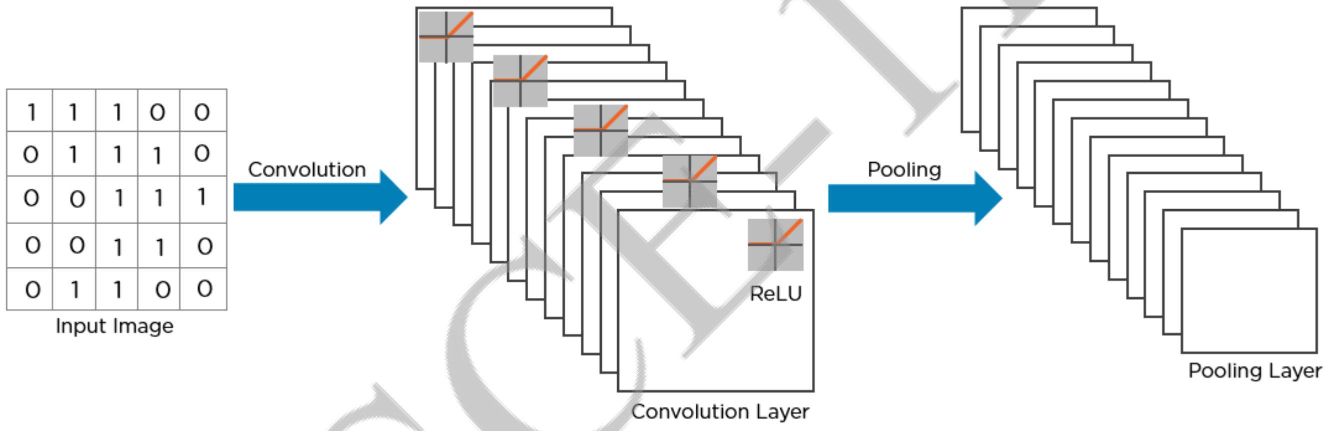
Pooling is a down-sampling operation that reduces the dimensionality of the feature map. The rectified feature map now goes through a pooling layer to generate a pooled feature map.



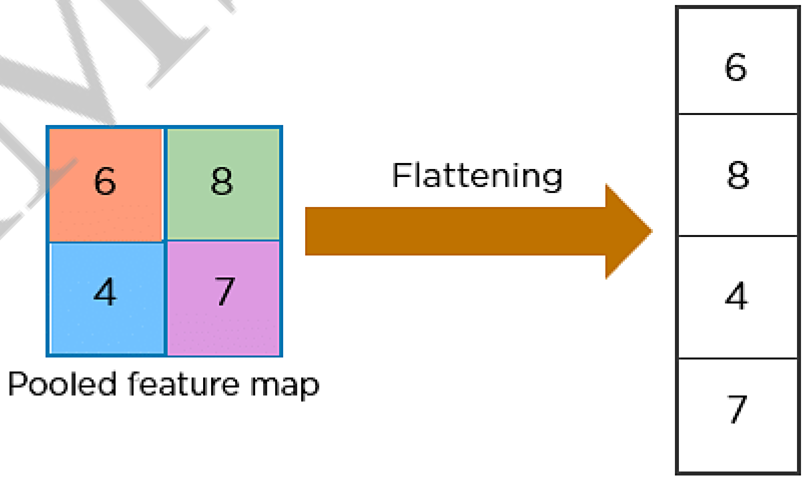
The pooling layer uses various filters to identify different parts of the image like edges, corners, body, feathers, eyes, and beak.



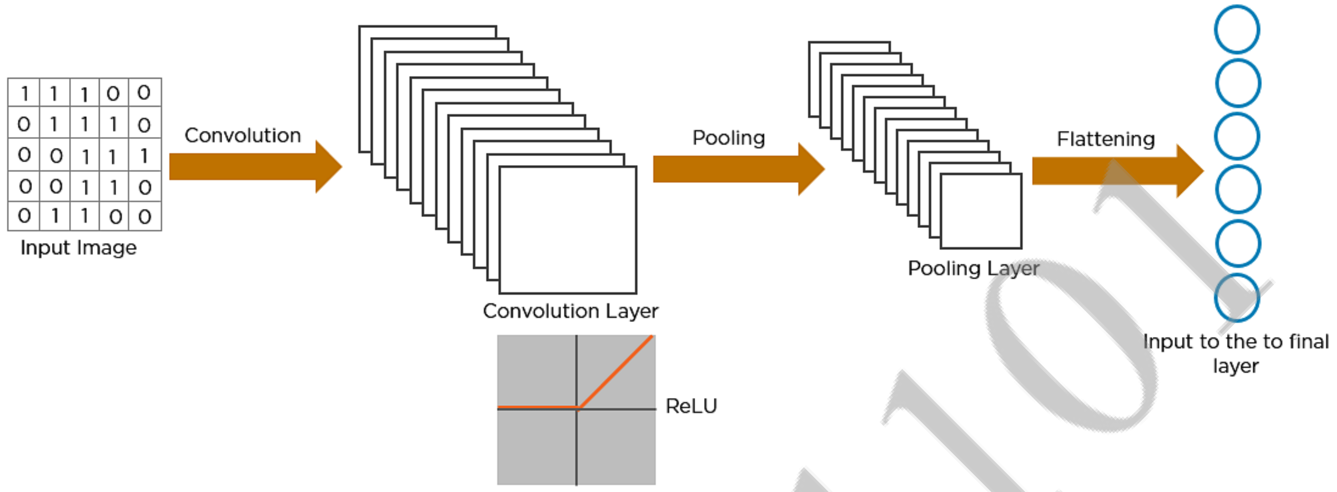
Here's how the structure of the convolution neural network looks so far:

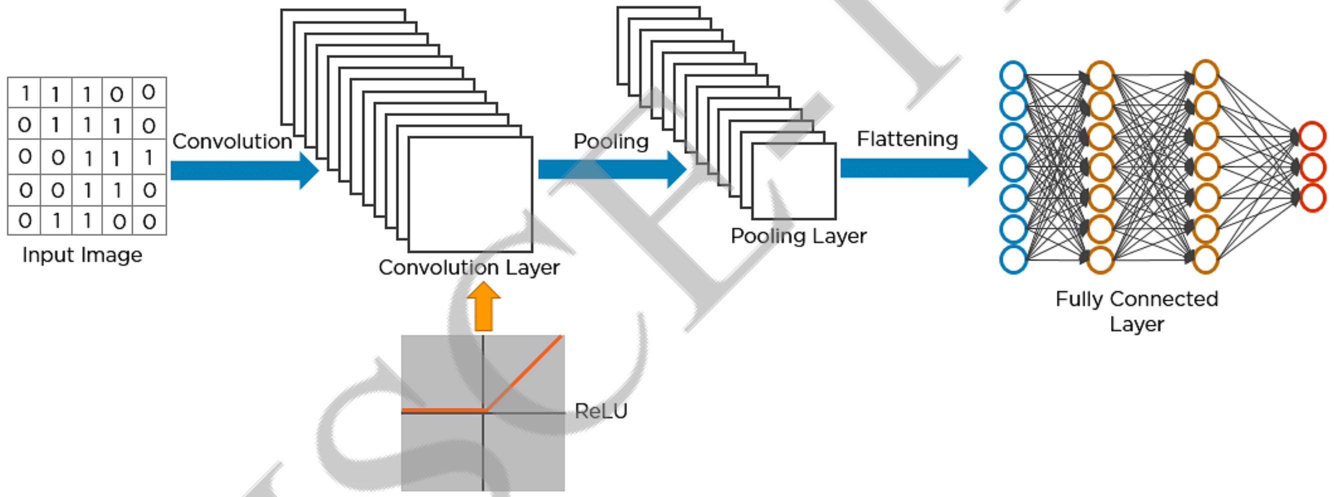


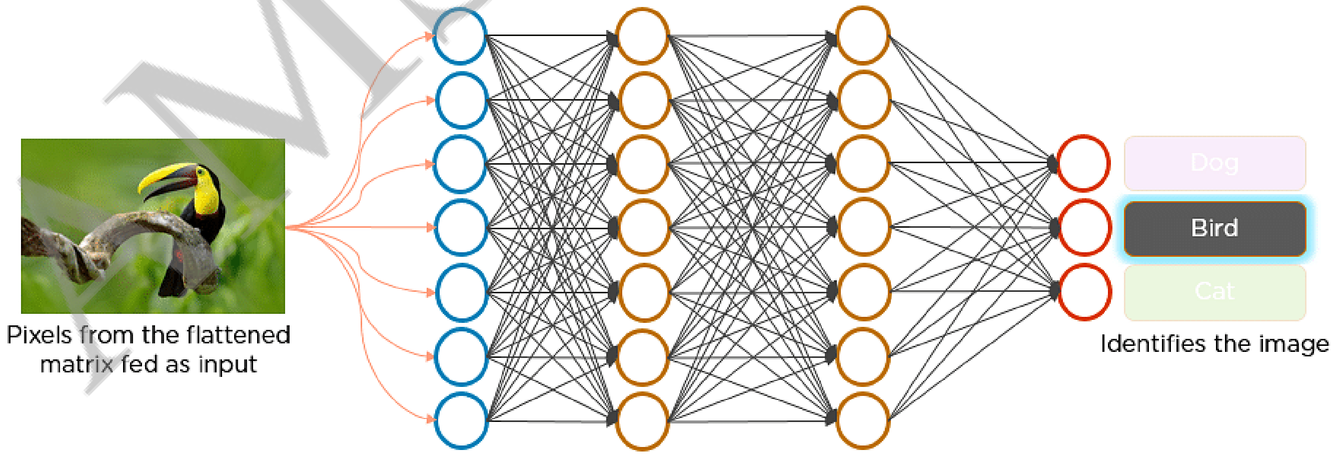
The next step in the process is called flattening. Flattening is used to convert all the resultant 2-Dimensional arrays from pooled feature maps into a single long continuous linear vector.



The flattened matrix is fed as input to the fully connected layer to classify the image.

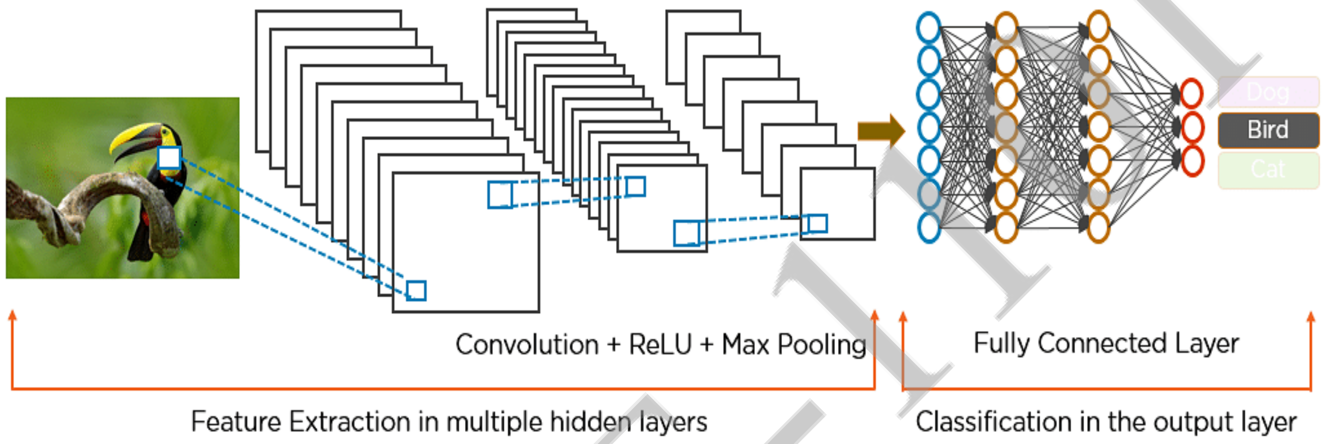






Here’s how exactly CNN recognizes a bird:

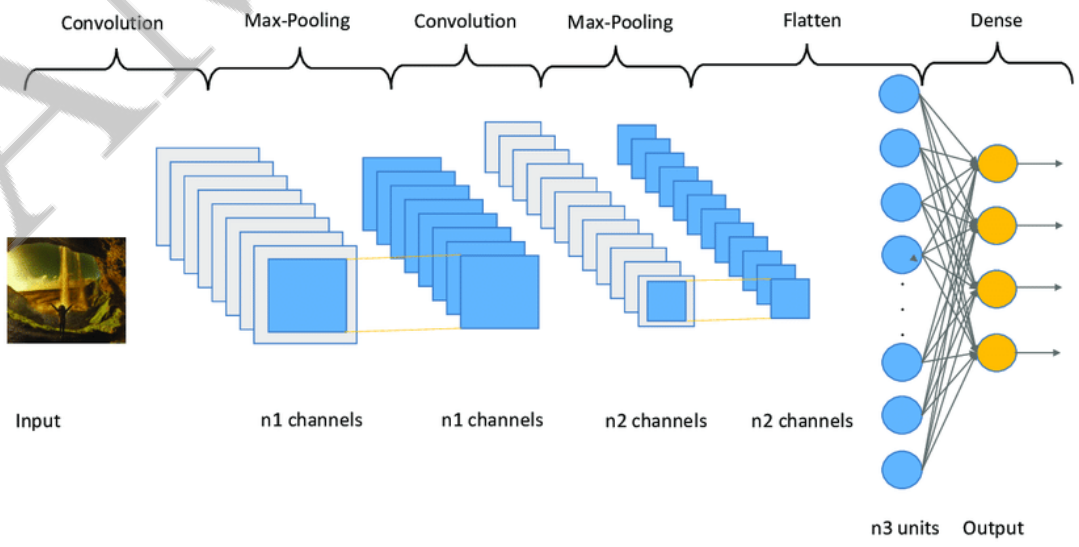
- The pixels from the image are fed to the convolutional layer that performs the convolution operation
- It results in a convolved map
- The convolved map is applied to a ReLU function to generate a rectified feature map
- The image is processed with multiple convolutions and ReLU layers for locating the features
- Different pooling layers with various filters are used to identify specific parts of the image
- The pooled feature map is flattened and fed to a fully connected layer to get the final output


Dog
Bird
Cat
Convolution + ReLU + Max Pooling
Fully Connected Layer
Feature Extraction in multiple hidden layers
Classification in the output layer

Use case implementation using CNN

What a convolutional neural network (CNN) does differently

A convolutional neural network is a specific kind of neural network with multiple layers. It processes data that has a grid-like arrangement then extracts important features. One huge advantage of using CNNs is that you don't need to do a lot of pre-processing on images.



A big difference between a CNN and a regular neural network is that CNNs use convolutions to handle the math behind the scenes. A convolution is used instead of matrix multiplication in at least one layer of the CNN. Convolutions take to two functions and return a function.

CNNs work by applying filters to your input data. What makes them so special is that CNNs are able to tune the filters as training happens. That way the results are fine-tuned in real time, even when you have huge data sets, like with images.

Since the filters can be updated to train the CNN better, this removes the need for hand-created filters. That gives us more flexibility in the number of filters we can apply to a data set and the relevance of those filters. Using this algorithm, we can work on more sophisticated problems like face recognition.

How Convolutional Neural Networks Work

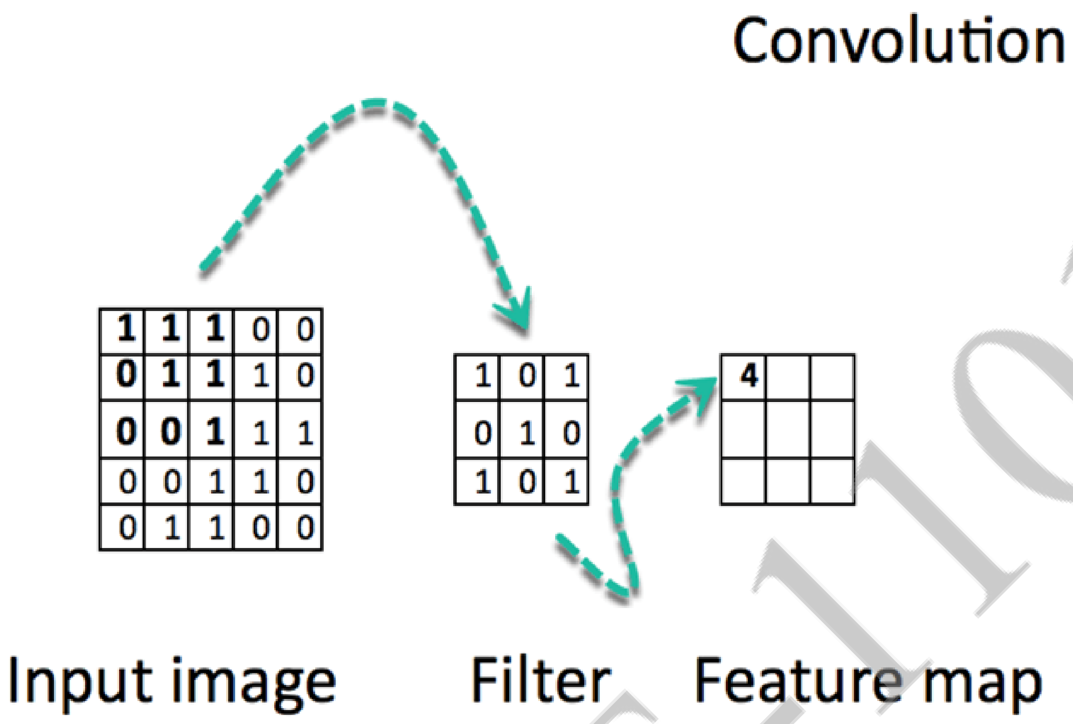
Convolutional neural networks are based on neuroscience findings. They are made of layers of artificial neurons called nodes. These nodes are functions that calculate the weighted sum of the inputs and return an activation map. This is the convolution part of the neural network.

Each node in a layer is defined by its weight values. When you give a layer some data, like an image, it takes the pixel values and picks out some of the visual features.

When you're working with data in a CNN, each layer returns activation maps. These maps point out important features in the data set. If you gave the CNN an image, it'll point out features based on pixel values, like colors, and give you an activation function.

Usually with images, a CNN will initially find the edges of the picture. Then this slight definition of the image will get passed to the next layer. Then that layer will start detecting things like corners and color groups. Then that image definition will get passed to the next layer and the cycle continues until a prediction is made.

As the layers get more defined, this is called max pooling. It only returns the most relevant features from the layer in the activation map. This is what gets passed to each successive layer until you get the final layer.



The last layer of a CNN is the classification layer which determines the predicted value based on the activation map. If you pass a handwriting sample to a CNN, the classification layer will tell you what letter is in the image. This is what autonomous vehicles use to determine whether an object is another car, a person, or some other obstacle.

Training a CNN is similar to training many other machine learning algorithms. You'll start with some training data that is separate from your test data and you'll tune your weights based on the accuracy of the predicted values. Just be careful that you don't overfit your model.

Different types of CNNs

There are multiple kinds of CNNs you can use depending on your problem.

1D CNN: With these, the CNN kernel moves in one direction. 1D CNNs are usually used on time-series data.

2D CNN: These kinds of CNN kernels move in two directions. You'll see these used with image labelling and processing.

3D CNN: This kind of CNN has a kernel that moves in three directions. With this type of CNN, researchers use them on 3D images like CT scans and MRIs.

In most cases, you'll see 2D CNNs because those are commonly associated with image data. Here are some of the applications that you might see CNNs used for.

- Recognize images with little preprocessing
- Recognize different hand-writing
- Computer vision applications
- Used in banking to read digits on checks
- Used in postal services to read zip codes on an envelope

Architecture of CNN

A typical CNN has the following 4 layers (O'Shea and Nash 2015)

1. Input layer
2. Convolution layer
3. Pooling layer
4. Fully connected layer

Please note that we will explain a 2 dimensional (2D) CNN here. But the same concepts apply to a 1 (or 3) dimensional CNN as well.

Input layer

The input layer represents the input to the CNN. An example input, could be a 28 pixel by 28 pixel grayscale image. Unlike FNN, we do not "flatten" the input to a 1D vector, and the input is presented to the network in 2D as a 28 x 28 matrix. This makes capturing spatial relationships easier.

Convolution layer

The convolution layer is composed of multiple filters (also called kernels). Filters for a 2D image are also 2D. Suppose we have a 28 pixel by 28 pixel grayscale image. Each pixel is represented by a number between 0 and 255, where 0 represents the color black, 255 represents the color white, and the values in between represent different shades of gray. Suppose we have a 3 by 3 filter (9 values in total), and the values are randomly set to 0 or 1. Convolution is the process of placing the 3 by 3 filter on the top left corner of the image, multiplying filter values by the pixel values and adding the results, moving the filter to the right one pixel at a time and repeating this process. When we get to the top right corner of the image, we simply move the filter down one pixel and restart from the left. This process ends when we get to the bottom right corner of the image.

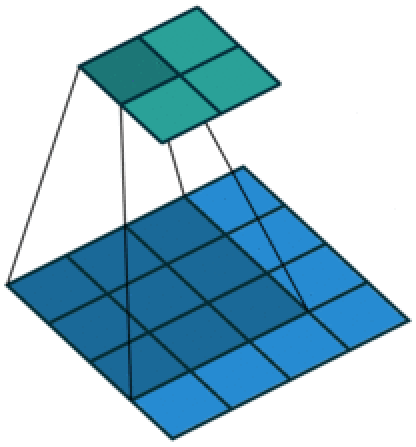


Figure 2: A 3 by 3 filter applied to a 4 by 4 image, resulting in a 2 by 2 image (Dumoulin and Visin 2016)

Covolution operator has the following parameters:

1. Filter size
2. Padding
3. Stride
4. Dilation
5. Activation function

Filter size can be 5 by 5, 3 by 3, and so on. Larger filter sizes should be avoided as the learning algorithm needs to learn filter values (weights), and larger filters increase the number of weights to be learned (more compute capacity, more training time, more chance of overfitting). Also, odd sized filters are preferred to even sized filters, due to the nice geometric property of all the input pixels being around the output pixel.

If you look at Figure 2 you see that after applying a 3 by 3 filter to a 4 by 4 image, we end up with a 2 by 2 image – the size of the image has gone down. If we want to keep the resultant image size the same, we can use *padding*. We pad the input in every direction with 0's before applying the filter. If the padding is 1 by 1, then we add 1 zero in evey direction. If its 2 by 2, then we add 2 zeros in every direction, and so on.

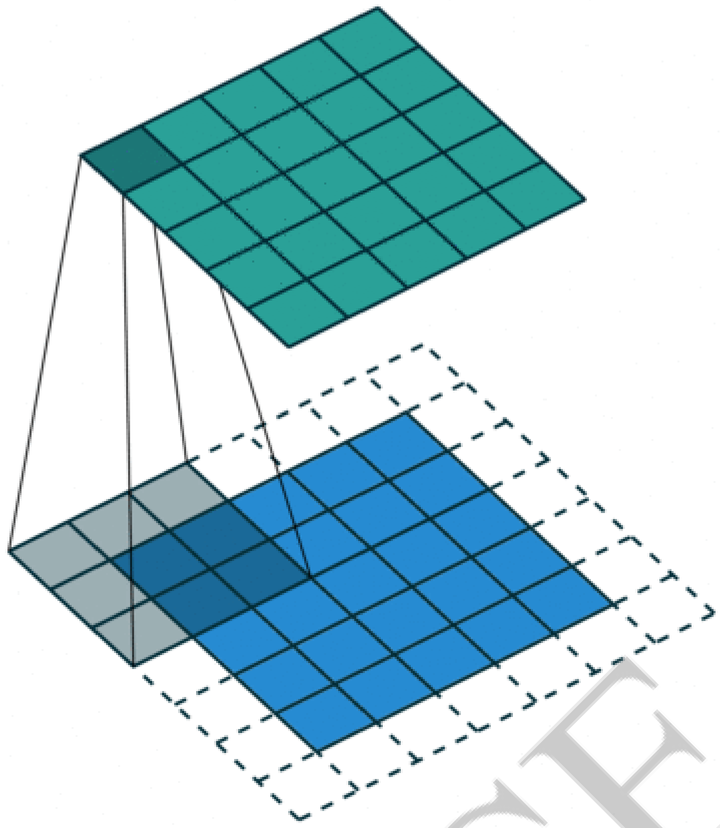


Figure 3: A 3 by 3 filter applied to a 5 by 5 image, with padding of 1, resulting in a 5 by 5 image (Dumoulin and Visin 2016)

As mentioned before, we start the convolution by placing the filter on the top left corner of the image, and after multiplying filter and image values (and adding them), we move the filter to the right and repeat the process. How many pixels we move to the right (or down) is the *stride*. In figure 2 and 3, the stride of the filter is 1. We move the filter one pixel to the right (or down). But we could use a different stride. Figure 4 shows an example of using stride of 2.

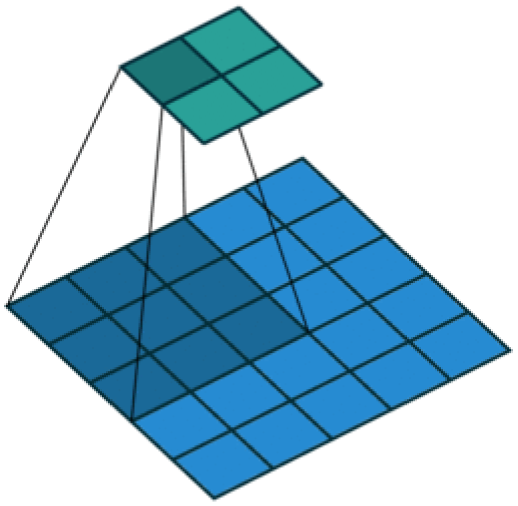


Figure 4: A 3 by 3 filter applied to a 5 by 5 image, with stride of 2, resulting in a 2 by 2 image (Dumoulin and Visin 2016)

When we apply a, say 3 by 3, filter to an image, our filter's output is affected by pixels in a 3 by 3 subset of the image. If we like to have a larger *receptive field* (portion of the image that affect our filter's output), we could use *dilation*. If we set the dilation to 2 (Figure 5), instead of a contiguous 3 by 3 subset of the image, every other pixel of a 5 by 5 subset of the image affects the filter's output.

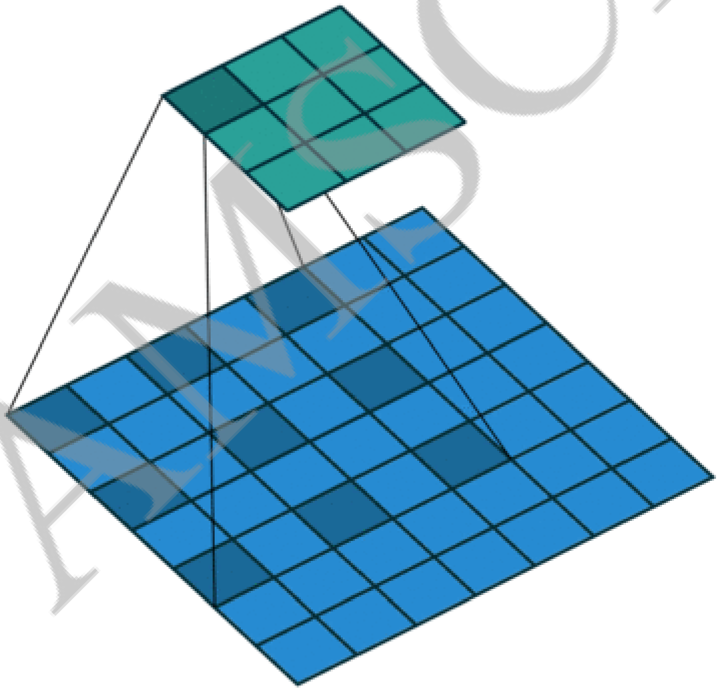
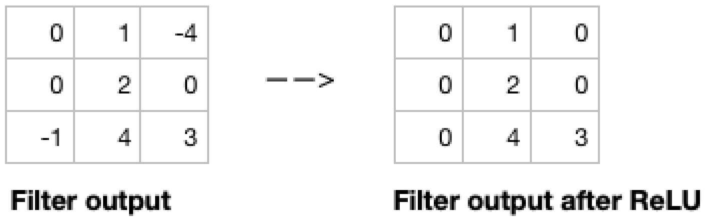


Figure 5: A 3 by 3 filter applied to a 7 by 7 image, with dilation of 2, resulting in a 3 by 3 image (Dumoulin and Visin 2016)

After the filter scans the whole image, we apply an activation function to filter output to introduce non-linearlity. The preferred activation function used in CNN is ReLU or one its variants like Leaky ReLU (Nwankpa *et al.* 2018). ReLU leaves pixels with positive values in filter output as is, and replacing negative values with 0 (or a small number in case of Leaky ReLU). Figure 6 shows the results of applying ReLU activation function to a filter output.



0	1	-4
0	2	0
-1	4	3

Filter output

--->

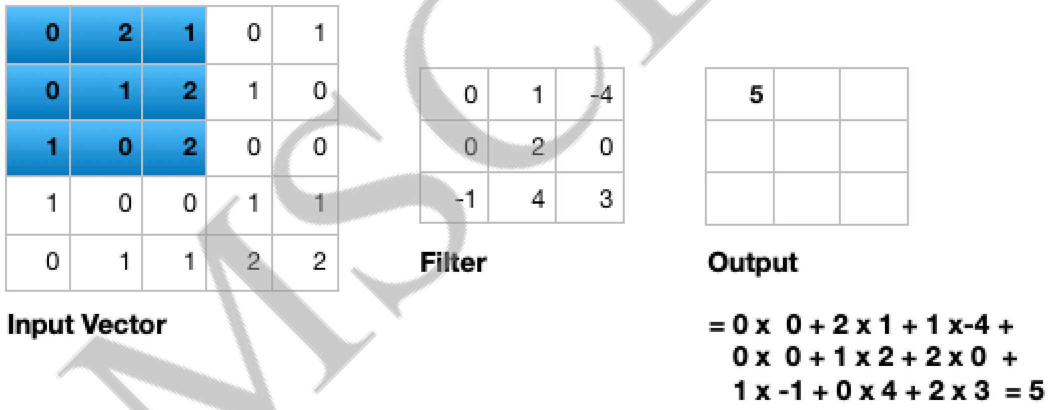
0	1	0
0	2	0
0	4	3

Filter output after ReLU

Figure 6: Applying ReLU activation function to filter output

Given the input size, filter size, padding, stride and dilation you can calculate the output size of the convolution operation as below.

$$(\text{input size} - (\text{filter size} + (\text{filter size} - 1) * (\text{dilation} - 1))) + (2 * padding) stride + 1$$



0	**2**	**1**	0	1
0	**1**	**2**	1	0
1	**0**	**2**	0	0
1	0	0	1	1
0	1	1	2	2

Input Vector

0	1	-4
0	2	0
-1	4	3

Filter

5		

Output

= 0 x 0 + 2 x 1 + 1 x -4 +
0 x 0 + 1 x 2 + 2 x 0 +
1 x -1 + 0 x 4 + 2 x 3 = 5

Figure 7: Illustration of single input channel two dimensional convolution

Figure 7 illustrates the calculations for a convolution operation, via a 3 by 3 filter on a single channel 5 by 5 input vector (5 x 5 x 1). Figure 8 illustrates the calculations when the input vector has 3 channels (5 x 5 x 3). To show this in 2 dimensions, we are displaying each channel in input vector and filter separately. Figure 9 shows a sample multi-channel 2D convolution in 3 dimensions.

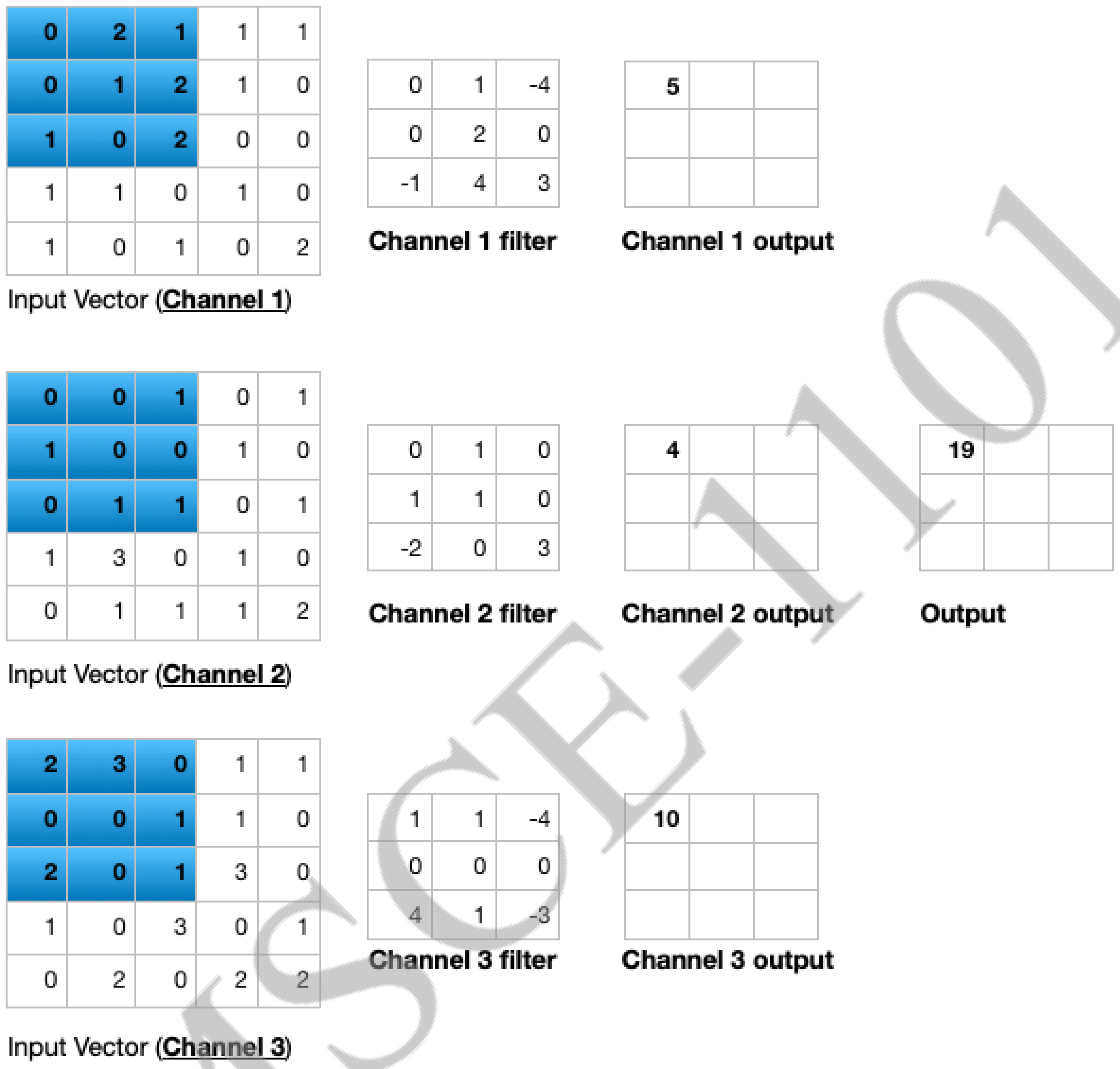


Figure 8: Illustration of multiple input channel two dimensional convolution

As Figures 8 and 9 show the output of a multi-channel 2 dimensional filter is a single channel 2 dimensional image. Applying *multiple* filters to the input image results in a multi-channel 2 dimensional image for the output. For example, if the input image is 28 by 28 by 3 (rows x columns x channels), and we apply a 3 by 3 filter with 1 by 1 padding, we would get a 28 by 28 by 1 image. If we apply 15 filters to the input image, our output would be 28 by 28 by 15. Hence, the number of filters in a convolution layer allows us to increase or decrease the channel size.

Pooling layer

The pooling layer performs down sampling to reduce the spatial dimensionality of the input. This decreases the number of parameters, which in turn reduces the learning time and computation, and the likelihood of overfitting. The most popular type of pooling is *max pooling*. Its usually a 2 by 2 filter with a stride of 2 that returns the maximum value as it slides over the input data (similar to convolution filters).

Fully connected layer

The last layer in a CNN is a fully connected layer. We connect all the nodes from the previous layer to this fully connected layer, which is responsible for classification of the image.

Deep Learning

Deep learning is based on the branch of machine learning, which is a subset of artificial intelligence. Since neural networks imitate the human brain and so deep learning will do. In deep learning, nothing is programmed explicitly. Basically, it is a machine learning class that makes use of numerous nonlinear processing units so as to perform feature extraction as well as transformation. The output from each preceding layer is taken as input by each one of the successive layers.

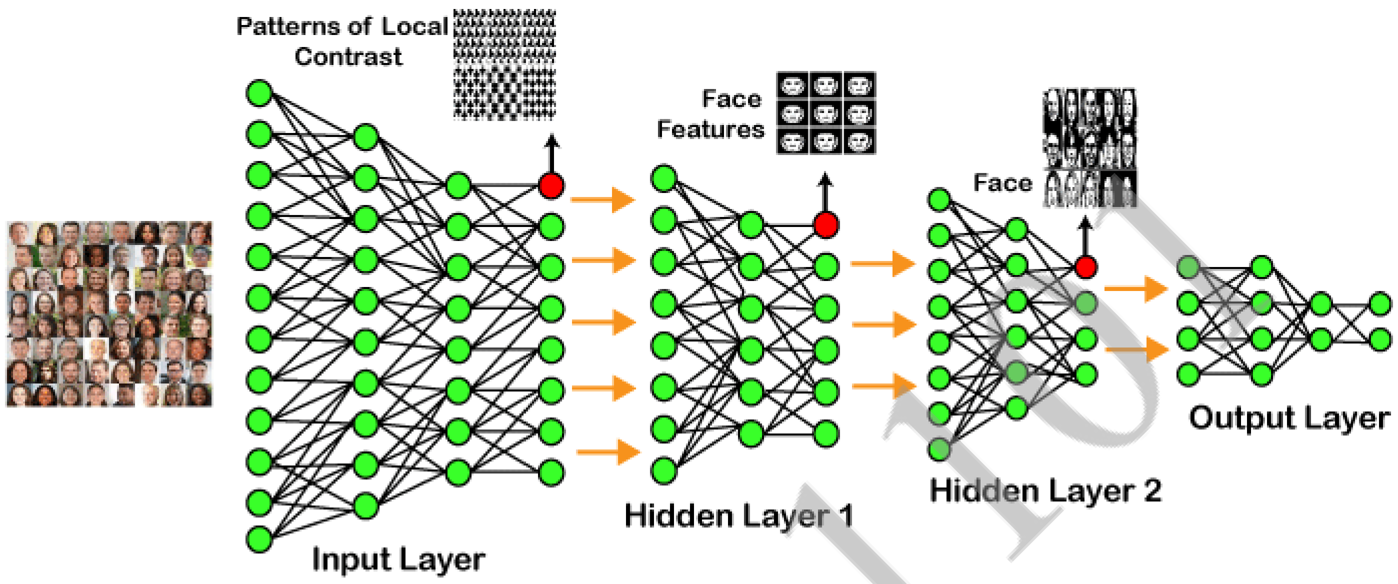
Deep learning models are capable enough to focus on the accurate features themselves by requiring a little guidance from the programmer and are very helpful in solving out the problem of dimensionality. Deep learning algorithms are used, especially when we have a huge no of inputs and outputs.

Since deep learning has been evolved by the machine learning, which itself is a subset of artificial intelligence and as the idea behind the artificial intelligence is to mimic the human behavior, so same is "the idea of deep learning to build such algorithm that can mimic the brain".

Deep learning is implemented with the help of Neural Networks, and the idea behind the motivation of Neural Network is the biological neurons, which is nothing but a brain cell.

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

Example of Deep Learning



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

Architectures

- ***Deep Neural Networks***
  It is a neural network that incorporates the complexity of a certain level, which means several numbers of hidden layers are encompassed in between the input and output layers. They are highly proficient on model and process non-linear associations.
- ***Deep Belief Networks***
  A deep belief network is a class of Deep Neural Network that comprises of multi-layer belief networks.
  Steps to perform DBN:
  1. With the help of the Contrastive Divergence algorithm, a layer of features is learned from perceptible units.
  2. Next, the formerly trained features are treated as visible units, which perform learning of features.
  3. Lastly, when the learning of the final hidden layer is accomplished, then the whole DBN is trained.

- ***Recurrent Neural Networks***
  It permits parallel as well as sequential computation, and it is exactly similar to that of the human brain (large feedback network of connected neurons). Since they are capable enough to reminisce all of the imperative things related to the input they have received, so they are more precise.

Types of Deep Learning Networks

1. Feed Forward Neural Network

A feed-forward neural network is none other than an Artificial Neural Network, which ensures that the nodes do not form a cycle. In this kind of neural network, all the perceptrons are organized within layers, such that the input layer takes the input, and the output layer generates the output. Since the hidden layers do not link with the outside world, it is named as hidden layers. Each of the perceptrons contained in one single layer is associated with each node in the subsequent layer. It can be concluded that all of the nodes are fully connected. It does not contain any visible or invisible connection between the nodes in the same layer. There are no back-loops in the feed-forward network. To minimize the prediction error, the backpropagation algorithm can be used to update the weight values.

Applications:

- Data Compression
- Pattern Recognition
- Computer Vision
- Sonar Target Recognition
- Speech Recognition
- Handwritten Characters Recognition

2. Recurrent Neural Network

Recurrent neural networks are yet another variation of feed-forward networks. Here each of the neurons present in the hidden layers receives an input with a specific delay in time. The Recurrent neural network mainly accesses the preceding info of existing iterations. For example, to guess the succeeding word in any sentence, one must have knowledge about the words that were previously used. It not only processes the inputs but also shares the length as well as weights crossways time. It does not let the size of the model to increase with the increase in the input size. However, the only problem with this recurrent neural network is that it has slow computational speed as well as it does not contemplate any future input for the current state. It has a problem with reminiscing prior information.

Applications:

- Machine Translation
- Robot Control
- Time Series Prediction

- Speech Recognition
- Speech Synthesis
- Time Series Anomaly Detection
- Rhythm Learning
- Music Composition

3. Convolutional Neural Network

Convolutional Neural Networks are a special kind of neural network mainly used for image classification, clustering of images and object recognition. DNNs enable unsupervised construction of hierarchical image representations. To achieve the best accuracy, deep convolutional neural networks are preferred more than any other neural network.

Applications:

- Identify Faces, Street Signs, Tumors.
- Image Recognition.
- Video Analysis.
- NLP.
- Anomaly Detection.
- Drug Discovery.
- Checkers Game.
- Time Series Forecasting.

4. Restricted Boltzmann Machine

RBMs are yet another variant of Boltzmann Machines. Here the neurons present in the input layer and the hidden layer encompasses symmetric connections amid them. However, there is no internal association within the respective layer. But in contrast to RBM, Boltzmann machines do encompass internal connections inside the hidden layer. These restrictions in BMs helps the model to train efficiently.

Applications:

- Filtering.
- Feature Learning.
- Classification.
- Risk Detection.
- Business and Economic analysis.

5. Autoencoders

An autoencoder neural network is another kind of unsupervised machine learning algorithm. Here the number of hidden cells is merely small than that of the input cells. But the number of input cells is equivalent to the number of output cells. An autoencoder network is trained to display the output similar to the fed input to force AEs to find common patterns and generalize the data. The autoencoders are mainly used for the smaller representation of the input. It helps in the reconstruction of the original data from compressed data. This algorithm is comparatively simple as it only necessitates the output identical to the input.

- **Encoder:** Convert input data in lower dimensions.
- **Decoder:** Reconstruct the compressed data.

Applications:

- Classification.
- Clustering.
- Feature Compression.

Deep learning applications

- ***Self-Driving Cars***
  In self-driven cars, it is able to capture the images around it by processing a huge amount of data, and then it will decide which actions should be incorporated to take a left or right or should it stop. So, accordingly, it will decide what actions it should take, which will further reduce the accidents that happen every year.
- ***Voice Controlled Assistance***
  When we talk about voice control assistance, then **Siri** is the one thing that comes into our mind. So, you can tell Siri whatever you want it to do it for you, and it will search it for you and display it for you.
- ***Automatic Image Caption Generation***
  Whatever image that you upload, the algorithm will work in such a way that it will generate caption accordingly. If you say blue colored eye, it will display a blue-colored eye with a caption at the bottom of the image.
- ***Automatic Machine Translation***
  With the help of automatic machine translation, we are able to convert one language into another with the help of deep learning.

Limitations

- It only learns through the observations.
- It comprises of biases issues.

Advantages

- It lessens the need for feature engineering.
- It eradicates all those costs that are needless.
- It easily identifies difficult defects.
- It results in the best-in-class performance on problems.

Disadvantages

- It requires an ample amount of data.

- It is quite expensive to train.
- It does not have strong theoretical groundwork.
-

Extreme Learning Machine

The learning pace of the feed-forward neural networks is considered as much slower than required. Due to this limitation, it has been a major barrier in many applications for decades. One of the major reasons is that sluggish gradient-based learning algorithms are widely employed to train neural networks which iteratively tune all of the network's parameters and makes the learning process slower. Unlike standard learning approaches, there is a learning technique for Single-Hidden Layer Feed-Forward Neural Networks (SLFNs) that is called Extreme Learning Machine (ELM). The ELMs are believed to have the ability to learn thousands of times faster than networks trained using the backpropagation technique. In this article, we will discuss ELM in detail. The major points that we will cover in this article are listed below.

Table of Contents

Let's proceed with understanding Feed-Forward NN.

The Feed-Forward Neural Network

The feedforward neural network was the earliest and most basic type of artificial neural network to be developed. In this network, information flows only in one direction forward from the input nodes to the output nodes, passing via any hidden nodes. The network is devoid of cycles or loops.

A single-layer perceptron network is the simplest type of FeedForward neural network, consisting of a single layer of output nodes with the inputs fed straight to the outputs via a sequence of weights. Each node calculates the total of the weights and inputs, and if the value is greater than a threshold (usually 0), the neuron fires and takes the active value (commonly 1); otherwise, it takes the deactivated value (typically 0 or -1). Artificial neurons or linear threshold units are neurons with this type of activation function. The term perceptron is frequently used in the literature to refer to networks that contain only one of these components.

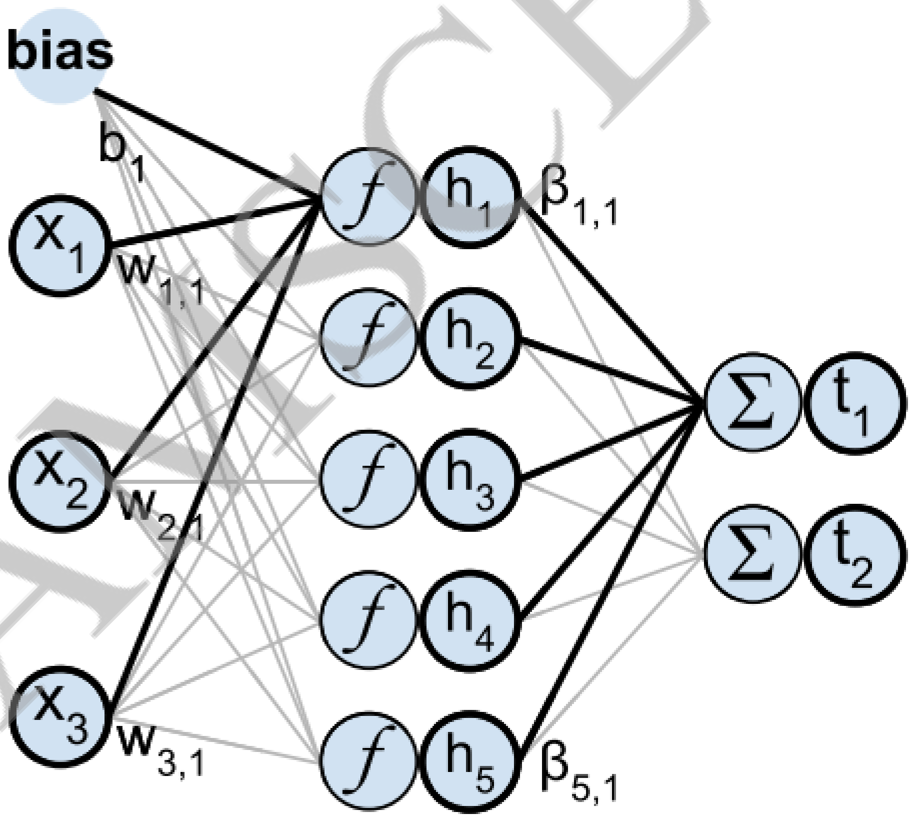
Extreme Learning Machine (ELM)

Extreme learning machines are feed-forward neural networks having a single layer or multiple layers of hidden nodes for classification, regression, clustering, sparse approximation, compression, and feature learning, where the hidden node parameters do not need to be modified. These hidden nodes might be assigned at random and never updated, or they can be inherited from their predecessors and never modified. In most cases, the weights of hidden nodes are usually learned in a single step which essentially results in a fast learning scheme.

These models, according to their inventors, are capable of producing good generalization performance and learning thousands of times quicker than backpropagation networks. These models can also outperform support vector machines in classification and regression applications, according to the research.

Fundamentals of ELM

An ELM is a quick way to train SLFN networks (shown in the below figure). An SLFN comprises three layers of neurons, however, the name Single refers to the model's one layer of non-linear neurons which is the hidden layer. The input layer offers data features but does not do any computations, whereas the output layer is linear with no transformation function and no bias.



Source

The ELM technique sets input layer weights W and biases b at random and never adjusts them. Because the input weights are fixed, the output weights ???? are independent of them (unlike in the Backpropagation training method) and have a straightforward solution that does not require iteration. Such a solution is also linear and very fast to compute for a linear output layer.

Random input layer weights improve the generalization qualities of a linear output layer solution because they provide virtually orthogonal (weakly correlated) hidden layer features. A linear system's solution is always in a range of inputs. If the solution weight range is constrained, orthogonal inputs provide a bigger solution space volume with these constrained weights. Smaller weight norms tend to make the system more stable and noise resistant since input errors are not aggravating in the output of the linear system with smaller coefficients. As a result, the random hidden layer creates weakly correlated hidden layer features, allowing for a solution with a low norm and strong generalization performance.

Variants of ELM

In this section, we will summarize several variants of ELM and will introduce them briefly.

ELM for Online Learning

There are numerous types of data in real-world applications, thus ELM must be changed to effectively learn from these data. For example, because the dataset is increasing, we may not always be able to access the entire dataset. From time to time, new samples are added to the dataset. Every time the set grows, we must retrain the ELM.

However, because the new samples frequently account for only a small portion of the total, re-training the network using the entire dataset again is inefficient. *Huang and Liang* proposed an online sequential ELM to address this issue (OS-ELM). The fundamental idea behind OS-ELM is to avoid re-training over old samples by employing a sequential approach. OS-ELM can update settings over new samples consecutively after startup. As a result, OS-ELM can be trained one at a time or block by block.

Incremental ELM

To build an incremental feedforward network, Huang et al. developed an incremental extreme learning machine (I-ELM). When a new hidden node was introduced, I-ELM randomly added nodes to the hidden layer one by one, freezing the output weights of the existing hidden nodes. I-ELM is effective for SLFNs with piecewise continuous activation functions (including differentiable) as well as SLFNs with continuous activation functions (such as threshold).

Pruning ELM

Rong et al. proposed a pruned-ELM (P-ELM) algorithm as a systematic and automated strategy for building ELM networks in light of the fact that using too few/many hidden nodes could lead to underfitting/overfitting concerns in pattern categorization. P-ELM started with a large number of hidden nodes and subsequently deleted the ones that were irrelevant or lowly relevant during learning by considering their relevance to the class labels.

ELM's architectural design can thus be automated as a result. When compared to the traditional ELM, simulation results indicated that the P-ELM resulted in compact network classifiers that generate fast response and robust prediction accuracy on unseen data.

Error-Minimized ELM

Feng et al. suggested an error-minimization-based method for ELM (EM-ELM) that can automatically identify the number of hidden nodes in generalized SLFNs by growing hidden nodes one by one or group by group. The output weights were changed incrementally as the networks grew, reducing the computational complexity dramatically. The simulation results on sigmoid type hidden nodes demonstrated that this strategy may greatly reduce the computational cost of ELM and offer an ELM implementation that is both efficient and effective.

Evolutionary ELM

When ELM is used, the number of hidden neurons is usually selected at random. Due to the random determination of input weights and hidden biases, ELM may require a greater number of hidden neurons. Zhu et al. introduced a novel learning algorithm called evolutionary extreme learning machine (E-ELM) for optimizing input weights and hidden biases and determining output weights.

To improve the input weights and hidden biases in E-ELM, the modified differential evolutionary algorithm was utilized. The output weights were determined analytically using Moore– Penrose (MP) generalized inverse.

Applications of ELM

Extreme learning machine has been used in many application domains such as medicine, chemistry, transportation, economy, robotics, and so on due to its superiority in training speed, accuracy, and generalization. This section highlights some of the most common ELM applications.

IoT Application

As the Internet of Things (IoT) has gained more attention from academic and industry circles in recent years, a growing number of scientists have developed a variety of IoT approaches or applications based on modern information technologies.

Using ELM in IoT applications can be done in a variety of ways. Rathore and Park developed an ELM-based strategy for detecting cyber-attacks. To identify assaults from ordinary visits, they devised a fog computing-based attack detection system and used an updated ELM as a classifier.

Transportation Application

The application of machine learning in transportation is a popular issue. Scientists, for example, used machine learning techniques to create a driver sleepiness monitoring system to prevent unsafe driving and save lives. It's been a long time since an extreme learning machine was used to solve transportation-related challenges. Sun and Ng suggested a two-stage approach to transportation system optimization that integrated linear programming and extreme learning machines. Two trials showed that combining their approaches might extend the life of a transportation system while also increasing its reliability.

Convolutional Networks

Convolutional networks convolutional (LeCun, 1989), also known as *neural networks* or *CNNs*, are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data,which can be thought of as a 2D grid of pixels. Convolutional

Convolution is a specialized kind of linear operation. **Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.**

The Convolution Operation

convolution is an operation on two functions of a realvalued argument. To motivate the definition of convolution, we start with examples of two functions we might use.

Suppose we are tracking the location of a spaceship with a laser sensor. Our
laser sensor provides a single output x(t), the position of the spaceship at time
t. Both x and t are real-valued, i.e., we can get a different reading from the laser sensor at any instant in time.
Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship's position, we would like to average together several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function w(a), where a is the age of a measurement.If we apply such a weighted average operation at every moment, we obtain a new function providing a smoothed estimate of the position **s** of the spaceship:

$$s(t) = \int x(a)w(t-a)da$$

This operation is called *convolution*. The convolution operation is typically
denoted with an asterisk:

$$s(t) = (x * w)(t)$$

In our example, w needs to be a valid probability density function, or the output is not a weighted average. Also, w needs to be 0 for all negative arguments, or it will look into the future, which is presumably beyond our capabilities.

In convolutional network terminology, the first argument (in this example, the function x) to the convolution is often referred to as the *input* and the second argument (in this example, the function w) as the *kernel*. The output is sometimes referred to as the *feature map*.

In our example, it might be more realistic to assume that our laser provides a measurement once per second. The time index t can then take on only integer values. If we now assume that x and w are defined only on integer t, we can define the discrete convolution:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

In machine learning applications, the input is usually a multidimensional array
of data and the kernel is usually a multidimensional array of parameters that are
adapted by the learning algorithm. We will refer to these multidimensional arrays
as tensors.
Finally, we often use convolutions over more than one axis at a time. For
example, if we use a two-dimensional image I as our input, we probably also want to use a two-dimensional kernel K:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n).$$

Convolution is commutative, meaning we can equivalently write:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n).$$

Usually the latter formula is more straightforward to implement in a machine learning library, because there is less variation in the range of valid values of m and n.The commutative property of convolution arises because we have *flipped* the kernel relative to the input, in the sense that as m increases, the index into the input increases, but the index into the kernel decreases. The only reason to flip the kernel is to obtain the commutative property.

While the commutative property is useful for writing proofs, it is not usually an important property of a neural network implementation. Instead, many neural network libraries implement a related function called the *cross-correlation*, which is the same as convolution but without flipping the kernel:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n).$$

Fig. 9.1 for an example of convolution (without kernel flipping) applied to a 2-D tensor.Discrete convolution can be viewed as multiplication by a matrix. However, the matrix has several entries constrained to be equal to other entries. For example, for univariate discrete convolution, each row of the matrix is constrained to be equal to the row above shifted by one element. This is known as a *Toeplitz matrix*.
In two dimensions, a *doubly block circulant matrix* corresponds to convolution. In addition to these constraints that several elements be equal to each other, convolution usually corresponds to a very sparse matrix (a matrix whose entries are mostly equal to zero). This is because the kernel is usually much smaller than the input image. Any neural network algorithm that works with matrix multiplication and does not depend on specific properties of the matrix structure should work with convolution, without requiring any further changes to the neural network.Typical convolutional neural networks do make use of further specializations in order to deal with large inputs efficiently, but these are not strictly necessary from a theoretical perspective.

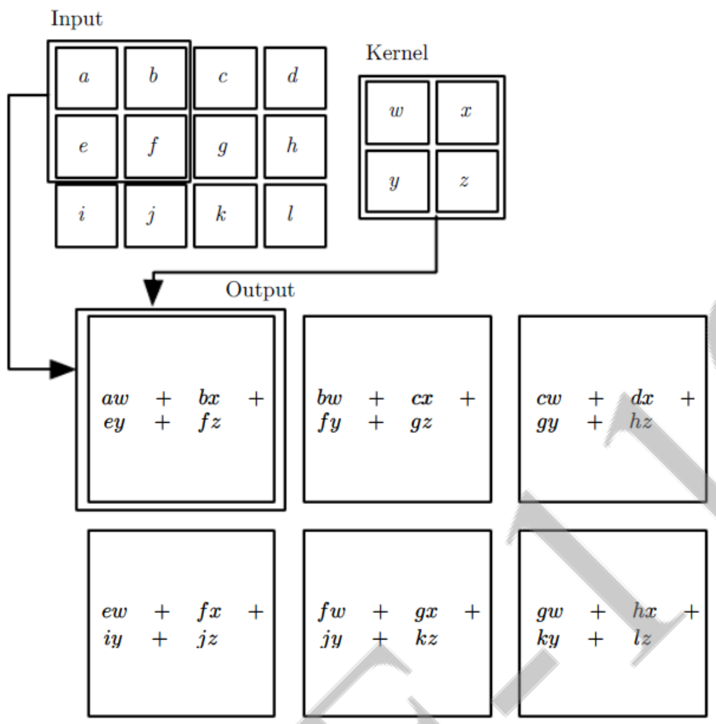


Figure 9.1: An example of 2-D convolution without kernel-flipping. In this case we restrict the output to only positions where the kernel lies entirely within the image, called "valid" convolution in some contexts. We draw boxes with arrows to indicate how the upper-left element of the output tensor is formed by applying the kernel to the corresponding upper-left region of the input tensor.

Motivation

Convolution leverages three important ideas that can help improve a machine learning system: *sparse interactions parameter sharing equivariant* , and *representations*. Moreover, convolution provides a means for working with inputs of variable
size. We now describe each of these ideas in turn.
Traditional neural network layers use matrix multiplication by a matrix of
parameters with a separate parameter describing the interaction between each
input unit and each output unit. This means every output unit interacts with every input unit. Convolutional networks, however, typically have *sparse interactions* (also referred to as *sparse connectivity* or *sparse weights*). This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output requires fewer operations. These improvements in efficiency are usually quite large. If there are m inputs and n outputs, then matrix multiplication requiresm×n parameters and the algorithms used in practice have O(m × n) runtime (per example). If we limit the number of connections

each output may have to k, then the sparsely connected approach requires only k × n parameters and O(k × n) runtime.

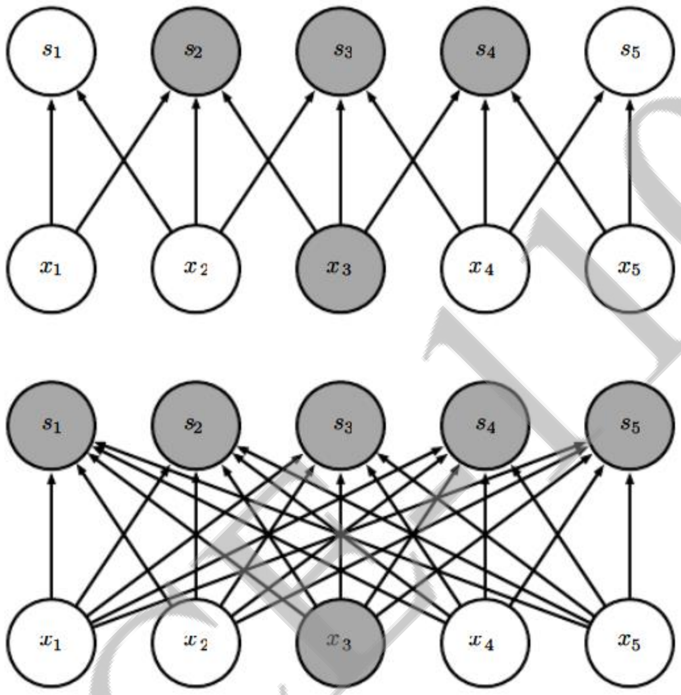


Figure 9.2: *Sparse connectivity, viewed from below:* We highlight one input unit, x_3, and also highlight the output units in $\boldsymbol{s}$ that are affected by this unit. *(Top)* When $\boldsymbol{s}$ is formed by convolution with a kernel of width 3, only three outputs are affected by $\boldsymbol{x}$. *(Bottom)* When $\boldsymbol{s}$ is formed by matrix multiplication, connectivity is no longer sparse, so all of the outputs are affected by x_3.

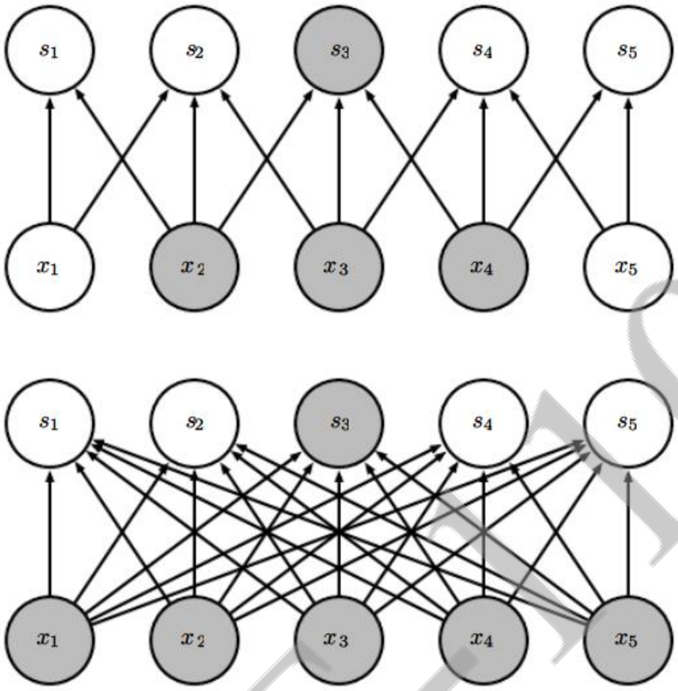


Figure 9.3: *Sparse connectivity, viewed from above:* We highlight one output unit, s_3, and also highlight the input units in $\boldsymbol{x}$ that affect this unit. These units are known as the *receptive field* of s_3. *(Top)* When $\boldsymbol{s}$ is formed by convolution with a kernel of width 3, only three inputs affect s_3. *(Bottom)* When $\boldsymbol{s}$ is formed by matrix multiplication, connectivity is no longer sparse, so all of the inputs affect s_3.

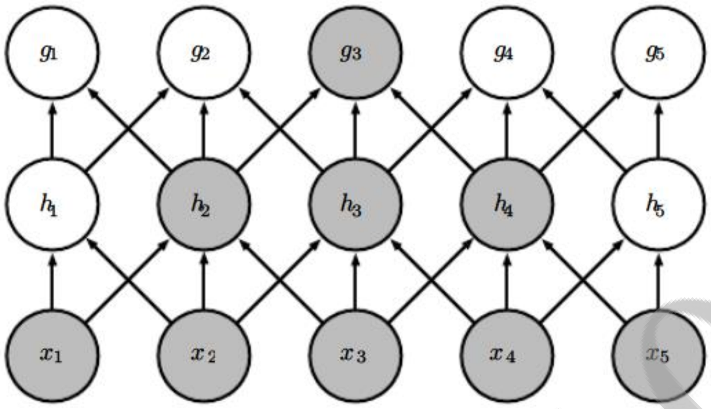


Figure 9.4: The receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers. This effect increases if the network includes architectural features like strided convolution (Fig. 9.12) or pooling (Sec. 9.3). This means that even though *direct* connections in a convolutional net are very sparse, units in the deeper layers can be *indirectly* connected to all or most of the input image.

Parameter sharing refers to using the same parameter for more than one function in a model. In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. As a synonym for parameter sharing, one can say that a network has *tied weights*, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere. In a convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn

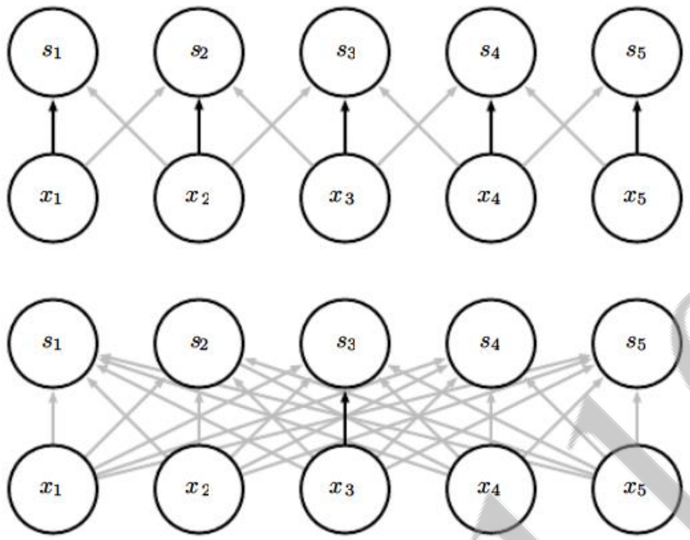


Figure 9.5: *Parameter sharing:* Black arrows indicate the connections that use a particular parameter in two different models. *(Top)* The black arrows indicate uses of the central element of a 3-element kernel in a convolutional model. Due to parameter sharing, this single parameter is used at all input locations. *(Bottom)* The single black arrow indicates the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing so the parameter is used only once.

only one set. This does not affect the runtime of forward propagation—it is still O(k × n)—but it does further reduce the storage requirements of the model to k parameters. Recall that k is usually several orders of magnitude less than m. Since m and n are usually roughly the same size, k is practically insignificant compared to m× n.

In the case of convolution, the particular form of parameter sharing causes the layer to have a property called *equivariance* to translation. To say a function is equivariant means that if the input changes, the output changes in the same way. Specifically, a function f(x) is equivariant to a function g if f (g(x)) = g(f(x)). In the case of convolution, if we let g be any function that translates the input, i.e., shifts it, then the convolution function is equivariant to g. For example, let I be a function giving image brightness at integer coordinates. Let g be a function mapping one image function to another image function, such that I⍰ = g(I) is the image function with I⍰(x, y) = I(x − 1, y). This shifts every pixel of I one unit to the right. If we apply this transformation to I , then apply convolution, the result will be the same as if we applied convolution to I⍰, then applied the transformation g to the output.

Pooling
A typical layer of a convolutional network consists of three stages (see Fig. 9.7). In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. This stage is sometimes called the *detector* stage. In the third stage, we use a *pooling function* to modify the output of the layer further.

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the *max pooling* (Zhou and Chellappa, 1988) operation reports the maximum output within a rectangular

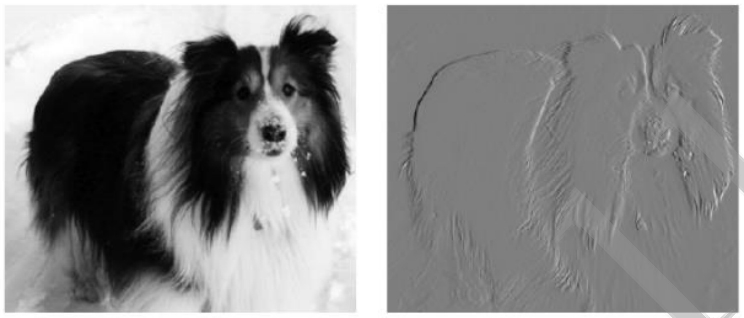
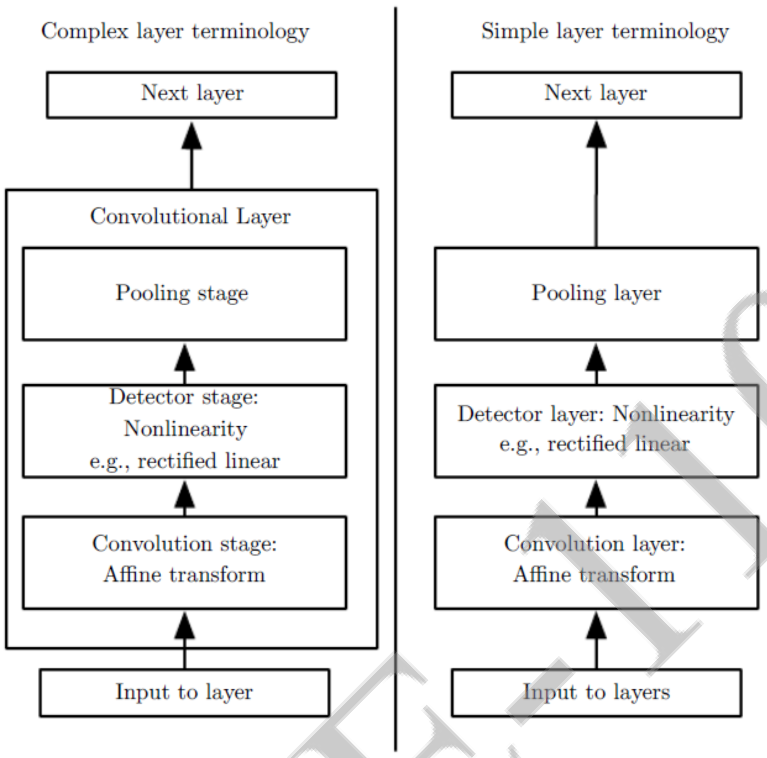


Figure 9.6: *Efficiency of edge detection.* The image on the right was formed by taking each pixel in the original image and subtracting the value of its neighboring pixel on the left. This shows the strength of all of the vertically oriented edges in the input image, which can be a useful operation for object detection. Both images are 280 pixels tall.



Complex layer terminology
Next layer
Convolutional Layer
Pooling stage
Detector stage: Nonlinearity e.g., rectified linear
Convolution stage: Affine transform
Input to layer
Simple layer terminology
Next layer
Pooling layer
Detector layer: Nonlinearity e.g., rectified linear
Convolution layer: Affine transform
Input to layers

Figure 9.7: The components of a typical convolutional neural network layer. There are two commonly used sets of terminology for describing these layers. *(Left)* In this terminology, the convolutional net is viewed as a small number of relatively complex layers, with each layer having many "stages." In this terminology, there is a one-to-one mapping between kernel tensors and network layers. In this book we generally use this terminology. *(Right)* In this terminology, the convolutional net is viewed as a larger number of simple layers; every step of processing is regarded as a layer in its own right. This means that not every "layer" has parameters.

based on the distance from the central pixel. In all cases, pooling helps to make the representation become approximately *invariant* to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. See Fig. for an example 9.8 of how this works. **Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is.**

For example, when determining whether an image contains a face, we need not know the location of the eyes with pixel-perfect accuracy, we just need to know that there is an eye on the left side of the face and an eye on the right side of the face. In other contexts, it is more important to preserve the location of a feature. For example, if we want to find a corner defined by two edges meeting at a specific orientation, we need to preserve the location of the edges well enough to test whether they meet.

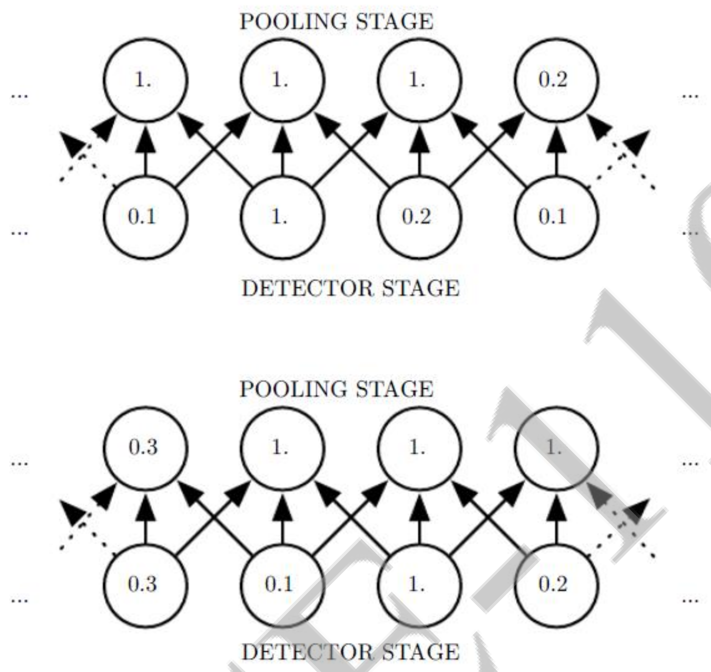


Figure 9.8: Max pooling introduces invariance. *(Top)* A view of the middle of the output of a convolutional layer. The bottom row shows outputs of the nonlinearity. The top row shows the outputs of max pooling, with a stride of one pixel between pooling regions and a pooling region width of three pixels. *(Bottom)* A view of the same network, after the input has been shifted to the right by one pixel. Every value in the bottom row has changed, but only half of the values in the top row have changed, because the max pooling units are only sensitive to the maximum value in the neighborhood, not its exact location.

Pooling over spatial regions produces invariance to translation, but if we pool over the outputs of separately parametrized convolutions, the features can learn which transformations to become invariant to (see Fig. 9.9). Because pooling summarizes the responses over a whole neighborhood, it is possible to use fewer pooling units than detector units, by reporting summary statistics for pooling regions spaced k pixels apart rather than 1 pixel apart. See Fig. 9.10 for an example. This improves the computational efficiency of the network because the next layer has roughly k times fewer inputs to process. When the number of parameters in the next layer is a function of its input size (such as when the next layer is fully connected and based on matrix multiplication) this reduction in the input size can also result in improved statistical efficiency and reduced memory requirements for storing the parameters.

For many tasks, pooling is essential for handling inputs of varying size. For example, if we want to classify images of variable size, the input to the classification layer must have a fixed size. This is usually accomplished by varying the size of an offset between pooling regions so that the classification layer always receives the same number of summary statistics regardless of the input size. For example, the final pooling layer of the network may be defined to output four sets of summary statistics, one for each quadrant of an image, regardless of the image size.

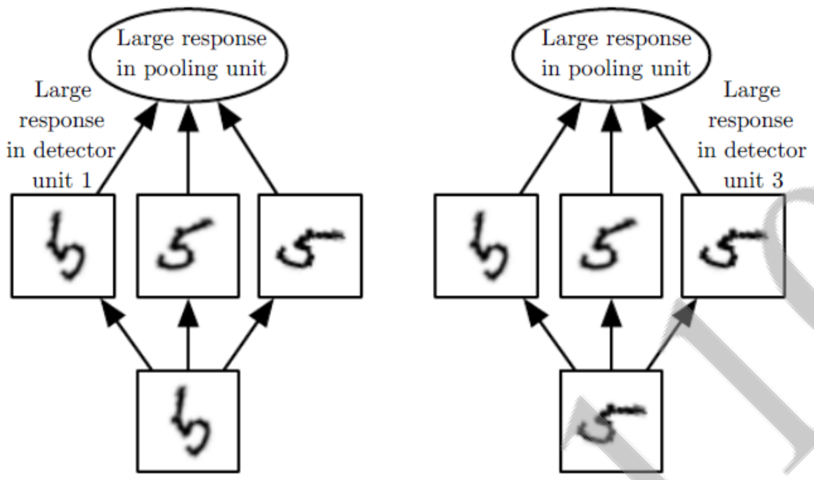

Large response in pooling unit
Large response in detector unit 1
Large response in pooling unit
Large response in detector unit 3

Figure 9.9: *Example of learned invariances:* A pooling unit that pools over multiple features that are learned with separate parameters can learn to be invariant to transformations of the input. Here we show how a set of three learned filters and a max pooling unit can learn to become invariant to rotation. All three filters are intended to detect a hand-written 5. Each filter attempts to match a slightly different orientation of the 5. When a 5 appears in the input, the corresponding filter will match it and cause a large activation in a detector unit. The max pooling unit then has a large activation regardless of which pooling unit was activated. We show here how the network processes two different inputs, resulting in two different detector units being activated. The effect on the pooling unit is roughly the same either way. This principle is leveraged by maxout networks (Goodfellow *et al.*, 2013a) and other convolutional networks. Max pooling over spatial positions is naturally invariant to translation; this multi-channel approach is only necessary for learning other transformations.

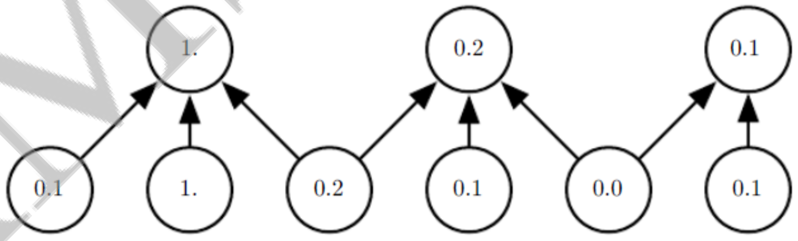


Figure 9.10: *Pooling with downsampling.* Here we use max-pooling with a pool width of three and a stride between pools of two. This reduces the representation size by a factor of two, which reduces the computational and statistical burden on the next layer. Note that the rightmost pooling region has a smaller size, but must be included if we do not want to ignore some of the detector units.

Pooling can complicate some kinds of neural network architectures that use top-down information, such as Boltzmann machines and autoencoders.

Some examples of complete convolutional network architectures for classification using convolution and pooling are shown in Fig. 9.11.

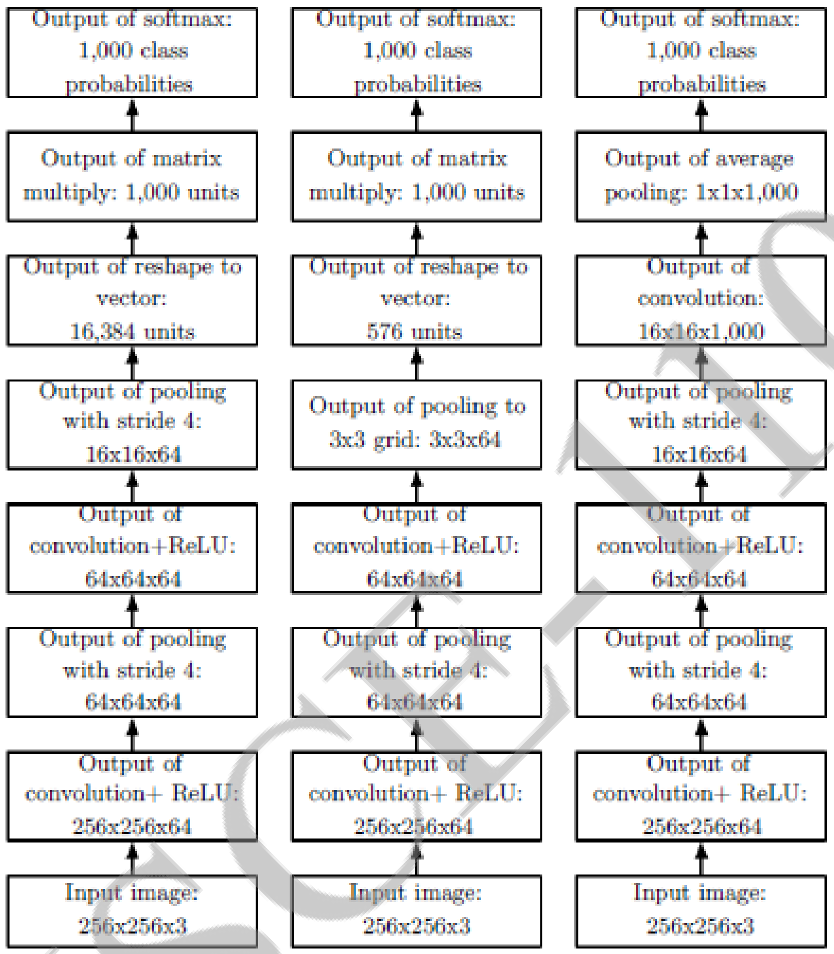

Output of softmax: 1,000 class probabilities
Output of matrix multiply: 1,000 units
Output of reshape to vector: 16,384 units
Output of pooling with stride 4: 16x16x64
Output of convolution+ReLU: 64x64x64
Output of pooling with stride 4: 64x64x64
Output of convolution+ ReLU: 256x256x64
Input image: 256x256x3
Output of softmax: 1,000 class probabilities
Output of matrix multiply: 1,000 units
Output of reshape to vector: 576 units
Output of pooling to 3x3 grid: 3x3x64
Output of convolution+ReLU: 64x64x64
Output of pooling with stride 4: 64x64x64
Output of convolution+ ReLU: 256x256x64
Input image: 256x256x3
Output of softmax: 1,000 class probabilities
Output of average pooling: 1x1x1,000
Output of convolution: 16x16x1,000
Output of pooling with stride 4: 16x16x64
Output of convolution+ReLU: 64x64x64
Output of pooling with stride 4: 64x64x64
Output of convolution+ ReLU: 256x256x64
Input image: 256x256x3

Figure 9.11: Examples of architectures for classification with convolutional networks. The specific strides and depths used in this figure are not advisable for real use; they are designed to be very shallow in order to fit onto the page. Real convolutional networks also often involve significant amounts of branching, unlike the chain structures used here for simplicity. *(Left)* A convolutional network that processes a fixed image size. After alternating between convolution and pooling for a few layers, the tensor for the convolutional feature map is reshaped to flatten out the spatial dimensions. The rest of the network is an ordinary feedforward network classifier, as described in Chapter 6. *(Center)* A convolutional network that processes a variable-sized image, but still maintains a fully connected section. This network uses a pooling operation with variably-sized pools but a fixed number of pools, in order to provide a fixed-size vector of 576 units to the fully connected portion of the network. *(Right)* A convolutional network that does not have any fully connected weight layer. Instead, the last convolutional layer outputs one feature map per class. The model presumably learns a map of how likely each class is to occur at each spatial location. Averaging a feature map down to a single value provides the argument to the softmax classifier at the top.

Variants of the Basic Convolution Function
When discussing convolution in the context of neural networks, we usually do not refer exactly to the standard discrete convolution operation as it is usually understood in the mathematical literature. The functions used in practice differ slightly. Here we describe these differences in detail, and highlight some useful properties of the functions used in neural networks. First, when we refer to convolution in the context of neural networks, we usually actually mean an operation that consists of many applications of convolution in parallel. This is because convolution with a single kernel can only extract one kind of feature, albeit at many spatial locations. Usually we want each layer of our network to extract many kinds of features, at many locations.

Additionally, the input is usually not just a grid of real values. Rather, it is a grid of vector-valued observations. For example, a color image has a red, green and blue intensity at each pixel. In a multilayer convolutional network, the input to the second layer is the output of the first layer, which usually has the output of many different convolutions at each position. When working with images, we usually think of the input and output of the convolution as being 3-D tensors, with one index into the different channels and two indices into the spatial coordinates of each channel. Software implementations usually work in batch mode, so they will actually use 4-D tensors, with the fourth axis indexing different examples in the batch, but we will omit the batch axis in our description here for simplicity. Because convolutional networks usually use multi-channel convolution, the linear operations they are based on are not guaranteed to be commutative, even if kernel-flipping is used. These multi-channel operations are only commutative if each operation has the same number of output channels as input channels.
Assume we have a 4-D kernel tensor **K** with element Ki,j,k,l giving the connection strength between a unit in channel i of the output and a unit in channel j of the input, with an offset of k rows and l columns between the output unit and the
input unit. Assume our input consists of observed data **V** with element Vi,j,k giving the value of the input unit within channel i at row j and column k. Assume our output consists of **Z** with the same format as **V**. If **Z** is produced by convolving **K** across **V** without flipping **K**, then

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n}$$

where the summation over l , m and n is over all values for which the tensor indexing operations inside the summation is valid. In linear algebra notation, we index into arrays using a 1 for the first entry. This necessitates the −1 in the above formula. Programming languages such as C and Python index starting from 0, rendering the above expression even simpler.
We may want to skip over some positions of the kernel in order to reduce the computational cost (at the expense of not extracting our features as finely). We

can think of this as downsampling the output of the full convolution function. If we want to sample only every s pixels in each direction in the output, then we can define a downsampled convolution function c such that

$$Z_{i,j,k} = c(\mathsf{K}, \mathsf{V}, s)_{i,j,k} = \sum_{l,m,n} \left[V_{l,(j-1)\times s+m,(k-1)\times s+n} K_{i,l,m,n} \right].$$

We refer to s as the *stride* of this downsampled convolution. It is also possible to define a separate stride for each direction of motion. See Fig. 9.12 for an illustration.

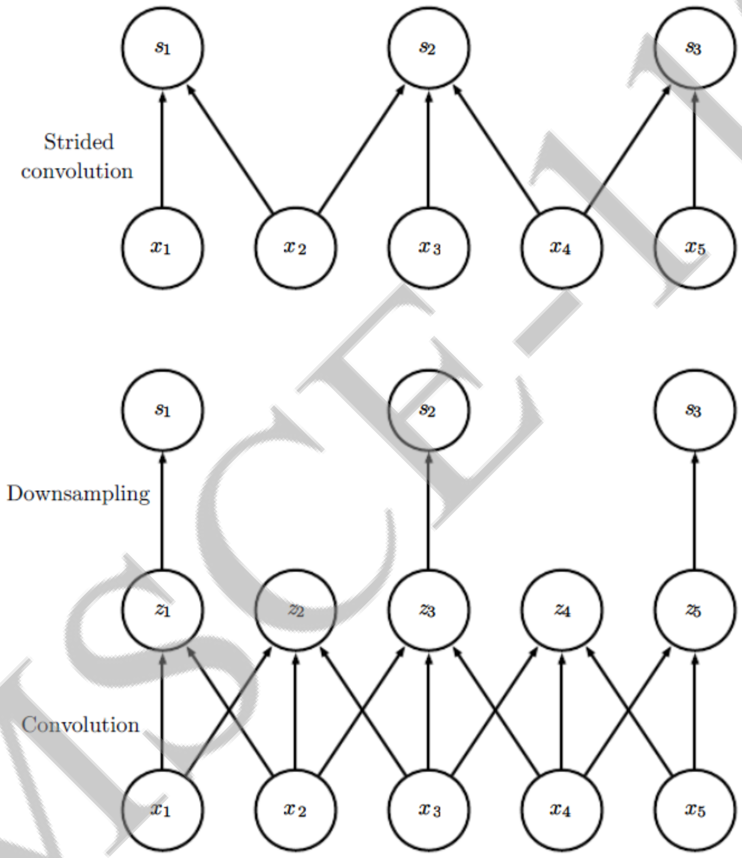


Figure 9.12: Convolution with a stride. In this example, we use a stride of two. *(Top)* Convolution with a stride length of two implemented in a single operation. *(Bottom)* Convolution with a stride greater than one pixel is mathematically equivalent to convolution with unit stride followed by downsampling. Obviously, the two-step approach involving downsampling is computationally wasteful, because it computes many values that are then discarded.

One essential feature of any convolutional network implementation is the ability to implicitly zero-pad the input **V** in order to make it wider. Without this feature, the width of the representation shrinks by one pixel less than the kernel width at each layer. Zero padding the input allows us to control the kernel width and the size of the output independently. Without zero padding, we are forced to choose between shrinking the spatial extent of the network rapidly and using small kernels—both scenarios that significantly limit the expressive power of the network.

Three special cases of the zero-padding setting are worth mentioning. One is the extreme case in which no zero-padding is used whatsoever, and the convolution kernel is only allowed to visit positions where the entire kernel is contained entirely within the image. In MATLAB terminology, this is called *valid* convolution. In this case, all pixels in the output are a function of the same number of pixels in the input, so the behavior of an output pixel is somewhat more regular. However, the size of the output shrinks at each layer. If the input image has width m and the kernel has width k, the output will be of width m− k+ 1. The rate of this shrinkage can be dramatic if the kernels used are large. Since the shrinkage is greater than 0, it limits the number of convolutional layers that can be included in the network. As layers are added, the spatial dimension of the network will eventually drop to 1 × 1, at which point additional layers cannot meaningfully be considered convolutional. Another special case of the zero-padding setting is when just enough zero-padding is added to keep the size of the output equal to the size of the input. MATLAB calls this *same* convolution. In this case, the network can contain as many convolutional layers as the available hardware can support, since the operation of convolution does not modify the architectural possibilities available to the next layer. However, the input pixels near the border influence fewer output pixels than the input pixels near the center. This can make the border pixels somewhat underrepresented in the model. This motivates the other extreme case, which MATLAB refers to as *full convolution*, in which enough zeroes are added for every pixel to be visited k times in each direction, resulting in an output image of width m+ k − 1. In this case, the output pixels near the border are a function of fewer pixels than the output pixels near the center. This can make it difficult to learn a single kernel that performs well at all positions in the convolutional feature map. Usually the optimal amount of zero padding (in terms of test set classification accuracy) lies somewhere between "valid" and "same"convolution.

In some cases, we do not actually want to use convolution, but rather locally connected layers (LeCun, 1986, 1989). In this case, the adjacency matrix in the graph of our MLP is the same, but every connection has its own weight, specified

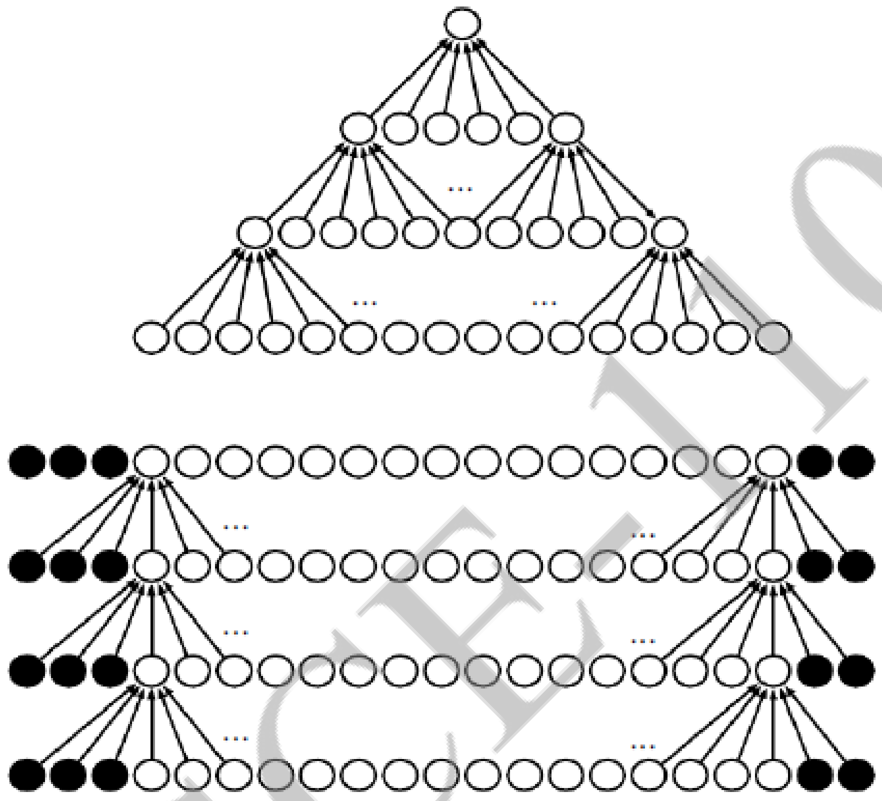


Figure 9.13: *The effect of zero padding on network size*: Consider a convolutional network with a kernel of width six at every layer. In this example, we do not use any pooling, so only the convolution operation itself shrinks the network size. *(Top)* In this convolutional network, we do not use any implicit zero padding. This causes the representation to shrink by five pixels at each layer. Starting from an input of sixteen pixels, we are only able to have three convolutional layers, and the last layer does not ever move the kernel, so arguably only two of the layers are truly convolutional. The rate of shrinking can be mitigated by using smaller kernels, but smaller kernels are less expressive and some shrinking is inevitable in this kind of architecture. *(Bottom)* By adding five implicit zeroes to each layer, we prevent the representation from shrinking with depth. This allows us to make an arbitrarily deep convolutional network.

by a 6-D tensor **W.** The indices into **W** are respectively: i, the output channel, j, the output row, k, the output column, l, the input channel, m, the row offset within the input, and n, the column offset within the input. The linear part of a locally connected layer is then given by

$$Z_{i,j,k} = \sum_{l,m,n} \left[V_{l,j+m-1,k+n-1} w_{i,j,k,l,m,n} \right].$$

This is sometimes also called *unshared convolution*, because it is a similar operation to discrete convolution with a small kernel, but without sharing parameters across locations. Fig. 9.14 compares local connections, convolution, and full connections.

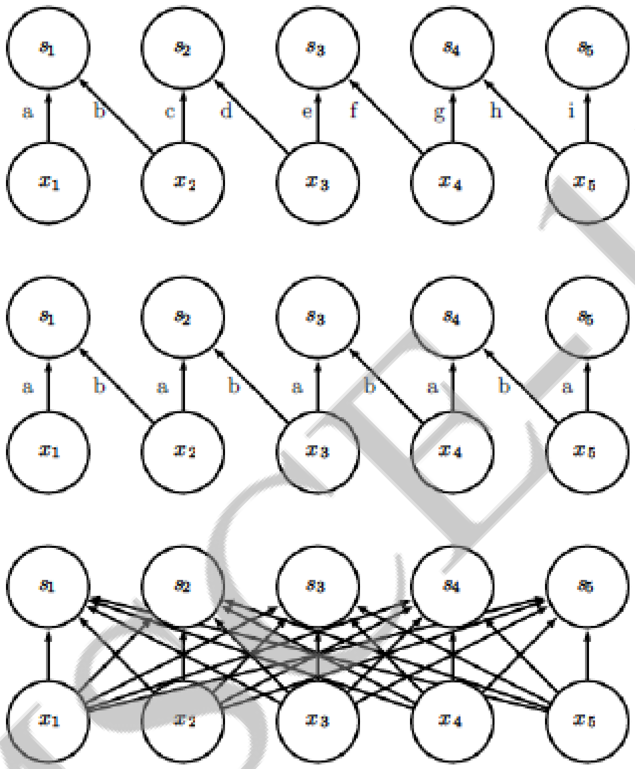


Figure 9.14: Comparison of local connections, convolution, and full connections.
(Top) A locally connected layer with a patch size of two pixels. Each edge is labeled with a unique letter to show that each edge is associated with its own weight parameter.
(Center) A convolutional layer with a kernel width of two pixels. This model has exactly the same connectivity as the locally connected layer. The difference lies not in which units interact with each other, but in how the parameters are shared. The locally connected layer has no parameter sharing. The convolutional layer uses the same two weights repeatedly across the entire input, as indicated by the repetition of the letters labeling each edge.
(Bottom) A fully connected layer resembles a locally connected layer in the sense that each edge has its own parameter (there are too many to label explicitly with letters in this diagram). However, it does not have the restricted connectivity of the locally connected layer.

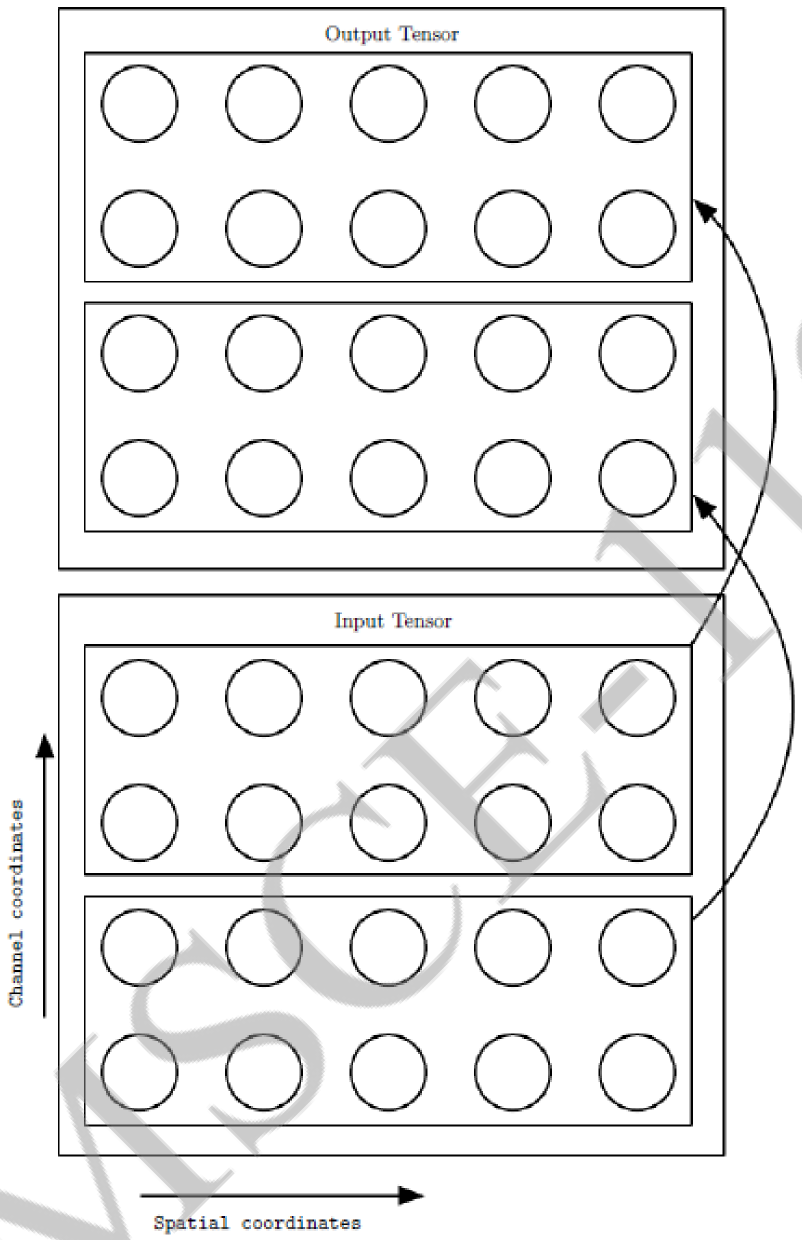


Figure 9.15: A convolutional network with the first two output channels connected to only the first two input channels, and the second two output channels connected to only the second two input channels.

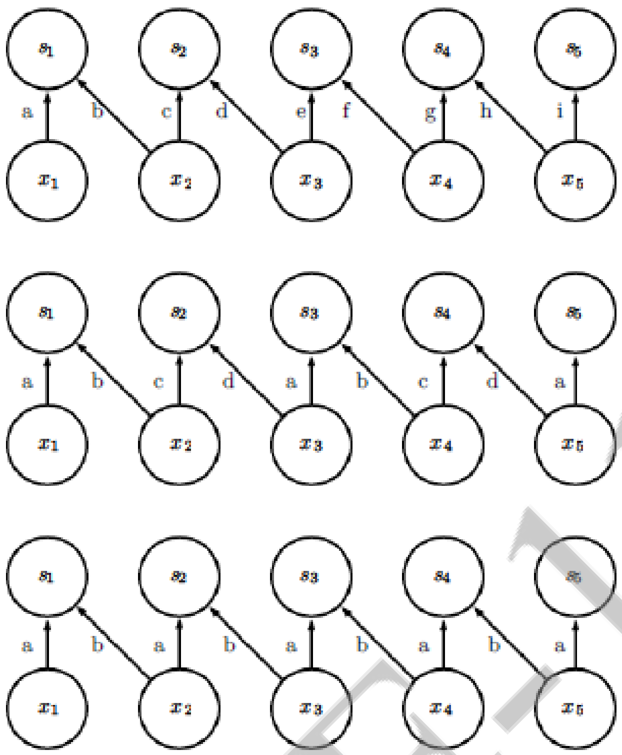


Figure 9.16: A comparison of locally connected layers, tiled convolution, and standard convolution. All three have the same sets of connections between units, when the same size of kernel is used. This diagram illustrates the use of a kernel that is two pixels wide. The differences between the methods lies in how they share parameters. *(Top)* A locally connected layer has no sharing at all. We indicate that each connection has its own weight by labeling each connection with a unique letter. *(Center)* Tiled convolution has a set of t different kernels. Here we illustrate the case of $t = 2$. One of these kernels has edges labeled "a" and "b," while the other has edges labeled "c" and "d." Each time we move one pixel to the right in the output, we move on to using a different kernel. This means that, like the locally connected layer, neighboring units in the output have different parameters. Unlike the locally connected layer, after we have gone through all t available kernels, we cycle back to the first kernel. If two output units are separated by a multiple of t steps, then they share parameters. *(Bottom)* Traditional convolution is equivalent to tiled convolution with $t = 1$. There is only one kernel and it is applied everywhere, as indicated in the diagram by using the kernel with weights labeled "a" and "b" everywhere.

the output width, this is the same as a locally connected layer.

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n,j\%t+1,k\%t+1},$$

where % is the modulo operation, with t%t = 0, (t + 1)%t = 1, etc. It is straightforward to generalize this equation to use a different tiling range for each dimension. Both locally connected layers and tiled convolutional layers have an interesting interaction with max-pooling: the detector units of these layers are driven by different filters. If these filters learn to detect different transformed versions of the same underlying features, then the max-pooled units become invariant to the learned transformation (see Fig. 9.9). Convolutional layers are hard-coded to be invariant specifically to translation.

The matrix involved is a function of the convolution kernel. The matrix is sparse and each element of the kernel is copied to several elements of the matrix. This view helps us to derive some of the other operations needed to implement a convolutional network. Multiplication by the transpose of the matrix defined by convolution is one such operation. This is the operation needed to back-propagate error derivatives through a convolutional layer, so it is needed to train convolutional networks that have more than one hidden layer. This same operation is also needed if we wish to reconstruct the visible units from the hidden units (Simard *et al.*, 1992).

Reconstructing the visible units is an operation commonly used in the models described in Part III of this book, such as autoencoders, RBMs, and sparse coding. Transpose convolution is necessary to construct convolutional versions of those models. Like the kernel gradient operation, this input gradient operation can be implemented using a convolution in some cases, but in the general case requires a third operation to be implemented. Care must be taken to coordinate this transpose operation with the forward propagation. The size of the output that the transpose operation should return depends on the zero padding policy and stride of the forward propagation operation, as well as the size of the forward propagation's output map. In some cases, multiple sizes of input to forward propagation can result in the same size of output map, so the transpose operation must be explicitly told what the size of the original input was.

These three operations—convolution, backprop from output to weights, and backprop from output to inputs—are sufficient to compute all of the gradients needed to train any depth of feedforward convolutional network, as well as to train convolutional networks with reconstruction functions based on the transpose of convolution. See ( ) for a full derivation Goodfellow 2010 of the equations in the fully general multi-dimensional, multi-example case. To give a sense of how these equations work, we present the two dimensional, single example version here.

Suppose we want to train a convolutional network that incorporates strided convolution of kernel stack **K** applied to multi-channel image **V** with stride s as defined by c(**K**,**V**, s) as in Eq. 9.8. Suppose we want to minimize some loss function J(**V**,**K**). During forward propagation, we will need to use c itself to output **Z**, which is then propagated through the rest of the network and used to compute the cost function J. During back-propagation, we will receive a tensor **G** such that

$$G_{i,j,k} = \frac{\partial}{\partial Z_{i,j,k}} J(\mathbf{V}, \mathbf{K}).$$

To train the network, we need to compute the derivatives with respect to the weights in the kernel. To do so, we can use a function

$$g(\mathsf{G}, \mathsf{V}, s)_{i,j,k,l} = \frac{\partial}{\partial K_{i,j,k,l}} J(\mathsf{V}, \mathsf{K}) = \sum_{m,n} G_{i,m,n} V_{j,(m-1)\times s+k,(n-1)\times s+l}. \tag{9.11}$$

If this layer is not the bottom layer of the network, we will need to compute the gradient with respect to V in order to back-propagate the error farther down. To do so, we can use a function

$$h(\mathsf{K}, \mathsf{G}, s)_{i,j,k} = \frac{\partial}{\partial V_{i,j,k}} J(\mathsf{V}, \mathsf{K}) \tag{9.12}$$

$$= \sum_{\substack{l,m \\ \text{s.t.} \\ (l-1)\times s+m=j}} \sum_{\substack{n,p \\ \text{s.t.} \\ (n-1)\times s+p=k}} \sum_{q} K_{q,i,m,p} G_{q,l,n}. \tag{9.13}$$

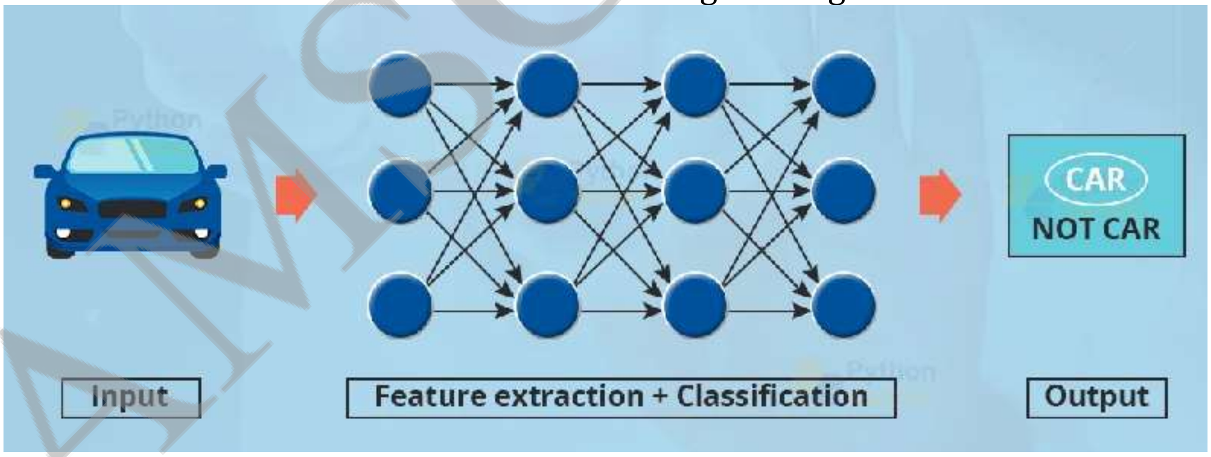
Unit 4 DEEP FEEDFORWARD NETWORKS

Syllabus

History of Deep Learning- A Probabilistic Theory of Deep Learning- Gradient Learning – Chain Rule and Backpropagation - Regularization: Dataset Augmentation – Noise Robustness -Early Stopping, Bagging and Dropout - batch normalization- VC Dimension and Neural Nets.

Deep Learning – An Introduction

- Deep learning is a method in artificial intelligence (AI) that teaches computers to process data in a way that is inspired by the human brain. Deep learning models can recognize complex patterns in pictures, text, sounds, and other data to produce accurate insights and predictions.
- It has become increasingly popular in recent years due to the advances in processing power and the availability of large datasets. Because it is based on artificial neural networks (ANNs) also known as deep neural networks (DNNs).
- These neural networks are inspired by the structure and function of the human brain's biological neurons, and they are designed to learn from large amounts of data.
- The key characteristic of Deep Learning is the use of deep neural networks, which have multiple layers of interconnected nodes. These networks can learn complex representations of data by discovering hierarchical patterns and features in the data.
- Deep Learning algorithms can automatically learn and improve from data without the need for manual feature engineering.



Difference between Machine Learning and Deep Learning:

Machine Learning	Deep Learning
Apply statistical algorithms to learn the hidden patterns and relationships in the dataset.	Uses artificial neural network architecture to learn the hidden patterns and relationships in the dataset.
Can work on the smaller amount of dataset	Requires the larger volume of dataset compared to machine learning
Takes less time to train the model.	Takes more time to train the model.
A model is created by relevant features which are manually extracted from images to detect an object in the image.	Relevant features are automatically extracted from images. It is an end-to-end learning process.
It can work on the CPU or requires less computing power as compared to deep learning.	It requires a high-performance computer with GPU.

History of Deep Learning

Here is a brief history of some key developments in deep learning:

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

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

The 1960s

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

The earliest efforts in developing deep learning algorithms came from Alexey Grigoryevich Ivakhnenko (developed the Group Method of Data Handling) and Valentin Grigor'evich Lapa (author of Cybernetics and Forecasting Techniques) in 1965. They used models with polynomial (complicated equations) activation functions, that were then analyzed statistically. From each layer, the best statistically chosen features were then forwarded on to the next layer (a slow, manual process).

The 1970s

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

The first "convolutional neural networks" were used by Kunihiko Fukushima. Fukushima designed neural networks with multiple pooling and convolutional layers. In 1979, he developed an artificial neural network, called Neocognitron, which used a hierarchical, multilayered design. This design allowed the computer the "learn" to recognize visual patterns. The networks resembled modern versions but were trained with a reinforcement strategy of recurring activation in multiple layers, which gained strength over time. Additionally, Fukushima's design allowed important features to be adjusted manually by increasing the "weight" of certain connections. Many of the concepts of Neocognitron continue to be used.

The use of top-down connections and new learning methods have allowed for a variety of neural networks to be realized. When more than one pattern is presented at the same time, the Selective Attention Model can separate and recognize individual patterns by shifting its attention from one to the other. (The same process many of us use when multitasking). A modern Neocognitron can not only identify patterns with missing information (for example, an incomplete number 5), but can also complete the image by adding the missing information. This could be described as "inference."

Back propagation, the use of errors in training deep learning models, evolved significantly in 1970. This was when Seppo Linnainmaa wrote his master's thesis, including a FORTRAN code for back propagation.

Unfortunately, the concept was not applied to neural networks until 1985. This was when Rumelhart, Williams, and Hinton demonstrated back propagation in a neural network could provide "interesting" distribution representations. Philosophically, this discovery brought to light the question within cognitive psychology of whether human understanding relies on symbolic logic (computationalism) or distributed representations (connectionism).

The 1980s and 90s

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

This time is also when the second AI winter (1985-90s) kicked in, which also effected research for neural networks and deep learning. Various overly-optimistic individuals had exaggerated the "immediate" potential of Artificial Intelligence, breaking expectations and angering investors. The anger was so intense, the phrase Artificial Intelligence reached pseudoscience status. Fortunately, some people continued to work on AI and DL, and some significant advances were made. In 1995, Dana Cortes and Vladimir Vapnik developed the support vector machine (a system for mapping and recognizing similar data).

LSTM (long short-term memory) for recurrent neural networks was developed in 1997, by Sepp Hochreiter and Juergen Schmidhuber.
The next significant evolutionary step for deep learning took place in 1999, when computers started becoming faster at processing data and GPU (graphics processing units) were developed. Faster processing, with GPUs processing pictures, increased computational speeds by 1000 times over a 10 year span. During this time, neural networks began to compete with support vector machines. While a neural network could be slow compared to a support vector machine, neural networks offered better results using the same data. Neural networks also have the advantage of continuing to improve as more training data is added.

2000-2010
Around the year 2000, The Vanishing Gradient Problem appeared. It was discovered "features" (lessons) formed in lower layers were not being learned by the upper layers, because no learning signal reached these layers. This was not a fundamental problem for all neural networks, just the ones with gradient-based learning methods. The source of the problem turned out to be certain activation functions. A number of activation functions condensed their input, in turn reducing the output range in a somewhat chaotic fashion. This produced large areas of input mapped over an extremely small range. In these areas of input, a large change will be reduced to a small change in the output, resulting in a vanishing gradient. Two solutions used to solve this problem were layer-by-layer pre-training and the development of long short-term memory.
In 2001, a research report by META Group (now called Gartner) described he challenges and opportunities of data growth as three-dimensional. The report described the increasing volume of data and the increasing speed of data as increasing the range of data sources and types. This was a call to prepare for the onslaught of Big Data, which was just starting.
In 2009, Fei-Fei Li, an AI professor at Stanford launched ImageNet, assembled a free database of more than 14 million labeled images. The Internet is, and was, full of unlabeled images. Labeled images were needed to "train" neural nets. Professor Li said, "Our vision was that big data would change the way machine learning works. Data drives learning."

2011-2020
By 2011, the speed of GPUs had increased significantly, making it possible to train convolutional neural networks "without" the layer-by-layer pre-training. With the increased computing speed, it became obvious deep learning had significant advantages in terms of efficiency and speed. One example is AlexNet, a convolutional neural network whose architecture won several international competitions during 2011 and 2012. Rectified linear units were used to enhance the speed and dropout.

Also in 2012, Google Brain released the results of an unusual project known as The Cat Experiment. The free-spirited project explored the difficulties of "unsupervised learning." Deep learning uses "supervised learning," meaning the convolutional neural net is trained using labeled data (think images from ImageNet). Using unsupervised learning, a convolutional neural net is given unlabeled data, and is then asked to seek out recurring patterns.
The Cat Experiment used a neural net spread over 1,000 computers. Ten million "unlabeled" images were taken randomly from YouTube, shown to the system, and then the training software was allowed to run. At the end of the training, one neuron in the highest layer was found to respond strongly to the images of cats. Andrew Ng, the project's founder said, "We also found a neuron that responded very strongly to human faces." Unsupervised learning remains a significant goal in the field of deep learning.
The Generative Adversarial Neural Network (GAN) was introduced in 2014. GAN was created by Ian Goodfellow. With GAN, two neural networks play against each other in a game. The goal of the game is for one network to imitate a photo, and trick its opponent into believing it is real. The opponent is, of course, looking for flaws. The game is played until the near perfect photo tricks the opponent. GAN provides a way to perfect a product (and has also begun being used by scammers).

Probabilistic Theory of Deep Learning
The Probabilistic Theory of Deep Learning (PTDL) is a framework aimed at understanding and explaining the behavior of deep neural networks (DNNs) through a probabilistic lens. It seeks to bridge the gap between traditional machine learning and deep learning by integrating probabilistic models with deep learning architectures.
The probabilistic neural networks employs deep neural networks that utilize probabilistic layers which can represent and process uncertainty; the deep probabilistic models uses probabilistic models that incorporate deep neural network components which capture complex non-linear stochastic relationships between the random variables.
The main advantages of probabilistic models are that these can capture the uncertainties in most real-world applications and provide essential information for decision making.
Probabilistic deep learning aims to address this limitation by incorporating uncertainty estimation into deep learning models. This can be achieved through various approaches:

- **Bayesian Neural Networks (BNNs):** BNNs treat model parameters as random variables with prior distributions. By inferring the posterior distribution of these parameters given the data, BNNs can provide not only point estimates but also uncertainty estimates for predictions.

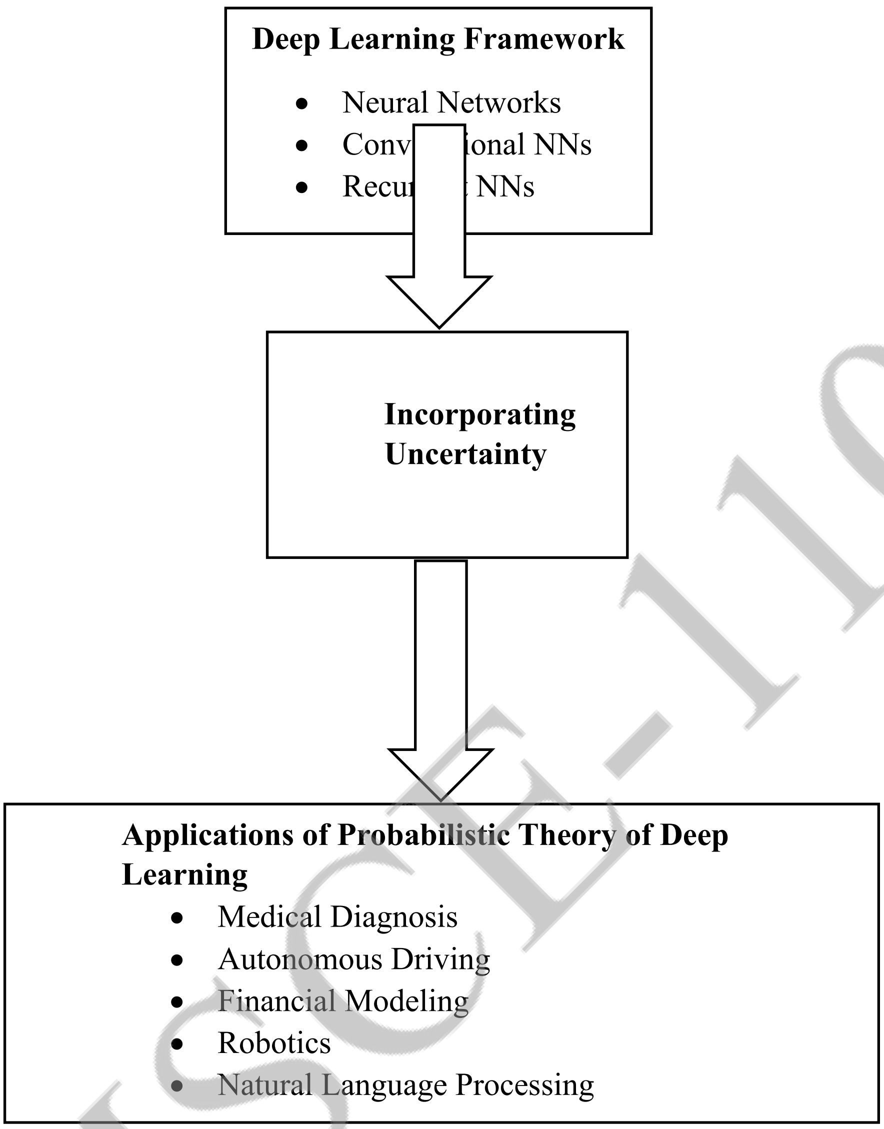
- **Variational Inference:** Variational inference is a technique used to approximate complex posterior distributions with simpler distributions. In the context of deep learning, variational inference can be used to approximate the posterior distribution of neural network weights, enabling uncertainty estimation.

- **Dropout as Bayesian Approximation:** Dropout is a regularization technique commonly used in deep learning to prevent overfitting. Interestingly, dropout can also be interpreted as a form of approximate Bayesian inference, where dropout during training can be seen as sampling from a distribution over possible neural network architecture. This can be leveraged to estimate uncertainty in predictions.

- **Gaussian Processes (GPs):** GPs are a powerful probabilistic modeling tool that can model distributions over functions. By combining GPs with deep neural networks, researchers have developed methods like Deep Gaussian Processes (DGPs), which provide uncertainty estimates while leveraging the representational power of deep learning architectures.

- **Monte Carlo Dropout**: Monte Carlo Dropout extends dropout to the testing phase by performing multiple stochastic forward passes through the network with dropout turned on. This allows for the estimation of predictive uncertainty by observing the variance of predictions across these passes.

- **Ensemble Methods**: Ensemble methods involve training multiple neural networks with different initializations or architectures and averaging their predictions. Ensemble methods naturally provide uncertainty estimates through the variance of predictions across the ensemble members.



Gradient Learning

"Gradient learning" typically refers to the process of updating the parameters of a model, often a neural network, using gradient descent optimization algorithms. Gradient descent is a fundamental optimization technique used to minimize the loss function of a model by iteratively adjusting its parameters in the direction of steepest descent of the loss function.

Gradient learning is essential for training neural networks and is the foundation of many deep learning algorithms.

Variants of gradient descent, such as

1. Stochastic gradient descent (SGD)
2. Mini-batch gradient descent
3. Adaptive learning rate methods like Adam are commonly used in practice to improve convergence speed and stability during training.

Neural networks are usually trained by using iterative, gradient-based optimizers. Gradient- based learning draws on the fact that it is generally much easier to minimize a reasonably smooth, continuous function than a discrete function.

- The **loss function** can be minimized by estimating the impact of small variations of the parameter values on the loss function. Convex optimization converges starting from any initial parameters.
- **Stochastic gradient descent** applied to non-convex loss functions has no such convergence guarantee and is sensitive to the values of the initial parameters.
- For **feedforward neural networks**, it is important to initialize all weights to small random values. The biases may be initialized to zero or to small positive values. The iterative gradient-based optimization algorithms used to train feedforward networks and almost all other deep models.

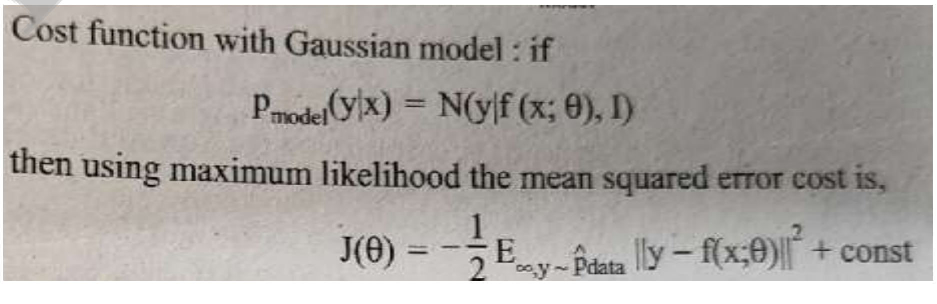
Cost Function

An important aspect of the design of deep neural networks is the cost function. They are similar to those for parametric models such as linear models. In most cases, parametric model defines a distribution ***p(y|x; 0)*** and simply use the principle of maximum likelihood.

The use of cross-entropy between the training data and the model's prediction's function. Most modern neural networks are trained using maximum likelihood.

Cost function is given by

J$(J(\theta) = \sum x, y \sim pdata\ Log\ Pmodel\ (Y|X)$



Cost function with Gaussian model : if

$$p_{model}(y|x) = N(y|f(x;\theta), I)$$

then using maximum likelihood the mean squared error cost is,

$$J(\theta) = -\frac{1}{2} E_{\infty,y\sim\hat{p}data} ||y - f(x;\theta)||^2 + const$$

The advantage of this approach to cost is that deriving cost from maximum likelihood removes the burden of designing cost functions for each model.

Desirable property of gradient:

- Gradient must be large and predictable enough to serve as a good guide to the learning algorithm.

Cross entropy and regularization:

- A property of cross-entropy cost used for MLE is that, it does not have a minimum value. For discrete output variables, they cannot represent probability of zero or one but come arbitrarily close. Logistic regression is an example.
- For real-valued output variables it becomes possible to assign extremely high density to correct training set outputs, e.g, by learning the variance parameter of Gaussian output and the resulting cross-entropy approaches negative infinity.

Learning conditional statistics:

- Instead of learning a full probability distribution, we often want to learn just one conditional statistic of y given x.

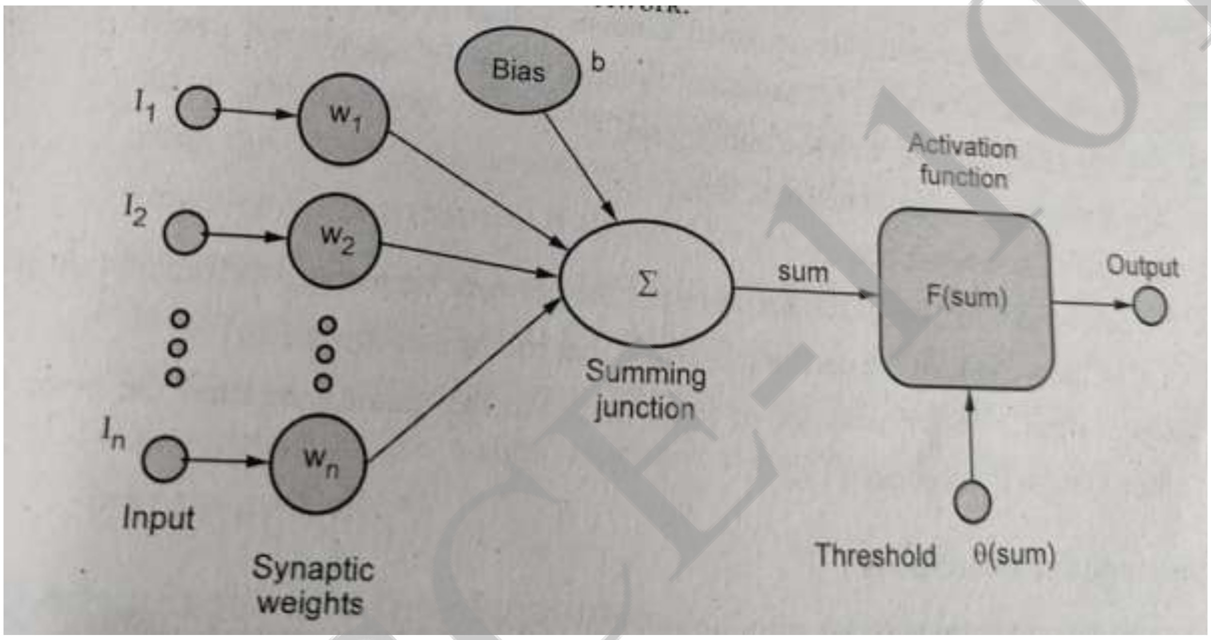
Learning a function:

- If we have a sufficiently powerful neural network, we can think of it as being powerful enough to determine any function "f". This function is limited only by boundedness and continuity.
- From this point of view, cost function is a function rather than a function.
- View cost as a functional, not a function. We can think of learning as a task of choosing a function rather than a set of parameters. We can design our cost function to have its minimum occur at a specific function we desire. For example, design the cost functional to have its minimum lie on the function that maps x to the expected value of y given x.

Chain Rule and Backpropagation

- The chain rule and backpropagation are fundamental concepts in the training of neural networks, especially in the context of gradient-based optimization.
- Backpropagation is a training method used for a multi-layer neural network. It is also called the generalized delta rule. It is a gradient descent method, which minimizes the total squared error of the output computed by the net.

- The backpropagation algorithm looks for the minimum value of the error function in weight space using a technique called the delta rule or gradient descent. The weights that minimize the error function is then considered to be a solution to the learning problem.
- Backpropagation is a systematic method for training multiple layer ANN. It is a generalization of Widrow-Hoff error correction rule. 80 % of ANN applications uses backpropagation.
- The Figure given below shows backpropagation network.



Here's an explanation of each:
Consider a simple neuron:

- Neuron has a summing junction and activation function.
- Any nonlinear function which differentiable everywhere and increases everywhere with sum can be used as activation function.
- **Examples:** Logistic function, arc tangent function, hyperbolic tangent activation function.

These activation function makes the multilayer network to have greater representational power than single layer network only when non-linearity is introduced.

Need of hidden layers:

1. A network with only two layers (input and output) can only represent the input with whatever representation already exists in the input data.
2. If the data is discontinuous or non-linearly separable, the innate representation is inconsistent, and the mapping cannot be learned using two layers (Input and Output).

3. Therefore, hidden layer(s) are used between input and output layers.

- **Weights** connects unit (neuron) in one layer only to those in the next higher layer. The output of the unit is scaled by the value of the connecting weight, and it is fed forward to provide a portion of the activation for the units in the next higher layer.
- **Backpropagation** can be applied to an artificial neural network with any number of hidden layers. The training objective is to adjust the weights so that the application of a set of inputs produces the desired outputs.

Training procedure:

The network is usually trained with a large number of input-output pairs.

Training Algorithm

1. Generate weights randomly to small random values (both positive and negative) ensure that the network is not saturated by large values of weights.
2. Choose a training pair from the training set.
3. Apply the input vector to network input.
4. Calculate the network output.
5. Calculate the error, the difference between the network output and the desired output.
6. Adjust the weights of the network in a way that minimizes this error.
7. Repeat steps 2 - 6 for each pair of input-output in the training set until the error for the entire system is acceptably low.

Forward pass and backward pass:

• Backpropagation neural network training involves two passes.
1. In the forward pass, the input signals moves forward from the network input to the output.
2. In the backward pass, the calculated error signals propagate backward through the network, where they are used to adjust the weights.
3. In the forward pass, the calculation of the output is carried out, layer by layer, in the forward direction. The output of one layer is the input to the next layer.

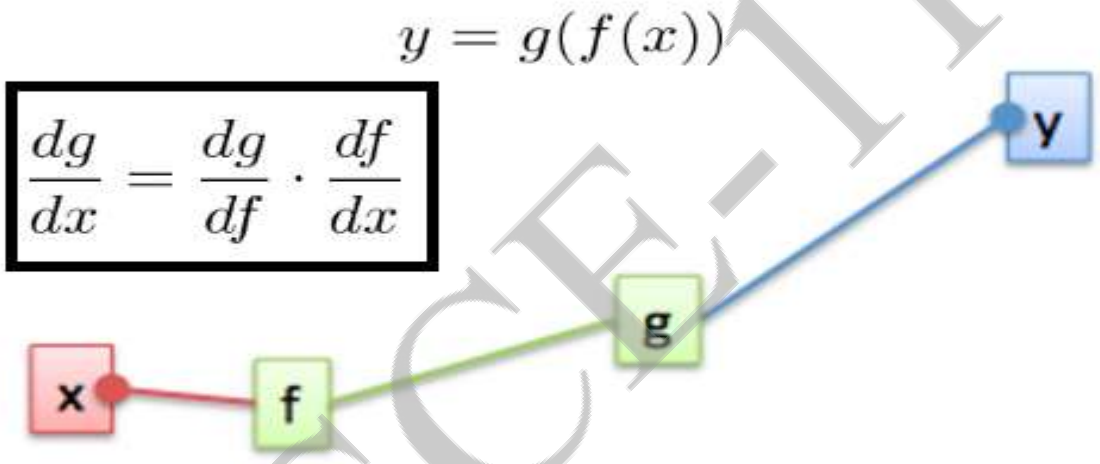
In the reverse pass,
a. The weights of the output neuron layer are adjusted first since the target value of each output neuron is available to guide the adjustment of the associated weights, using the delta rule.
b. Next, we adjust the weights of the middle layers. As the middle layer neurons have no target values, it makes the problem complex.

Chain Rule:

The chain rule is a rule from calculus that allows us to compute the derivative of a composite function. If we have a function $y = f(g(x))$, where f and g are differentiable functions, then the chain rule states that the derivative of y with respect to x can be expressed as:

$$\frac{dy}{dx} = \frac{dy}{dg} \cdot \frac{dg}{dx}$$

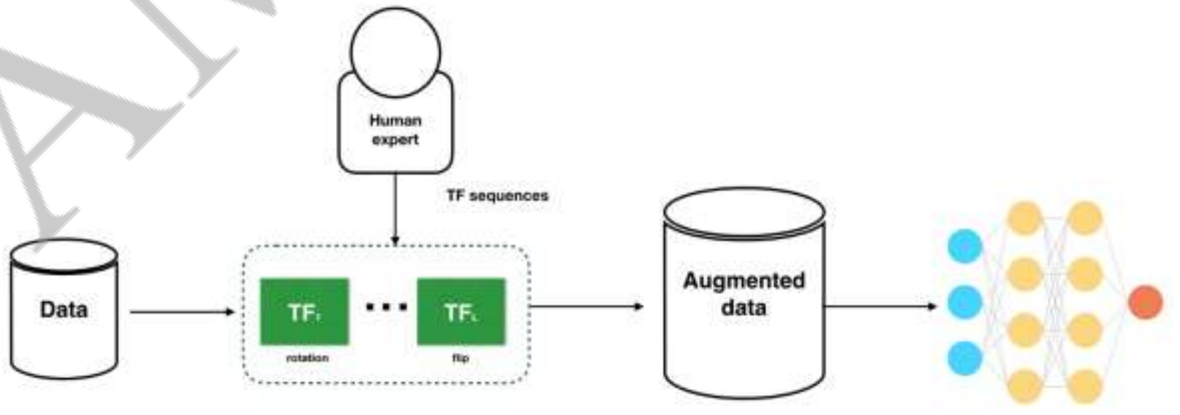
In the context of neural networks, the chain rule allows us to compute the derivative of the loss function with respect to the parameters (weights and biases) of the network by decomposing it into smaller derivatives across the layers of the network.



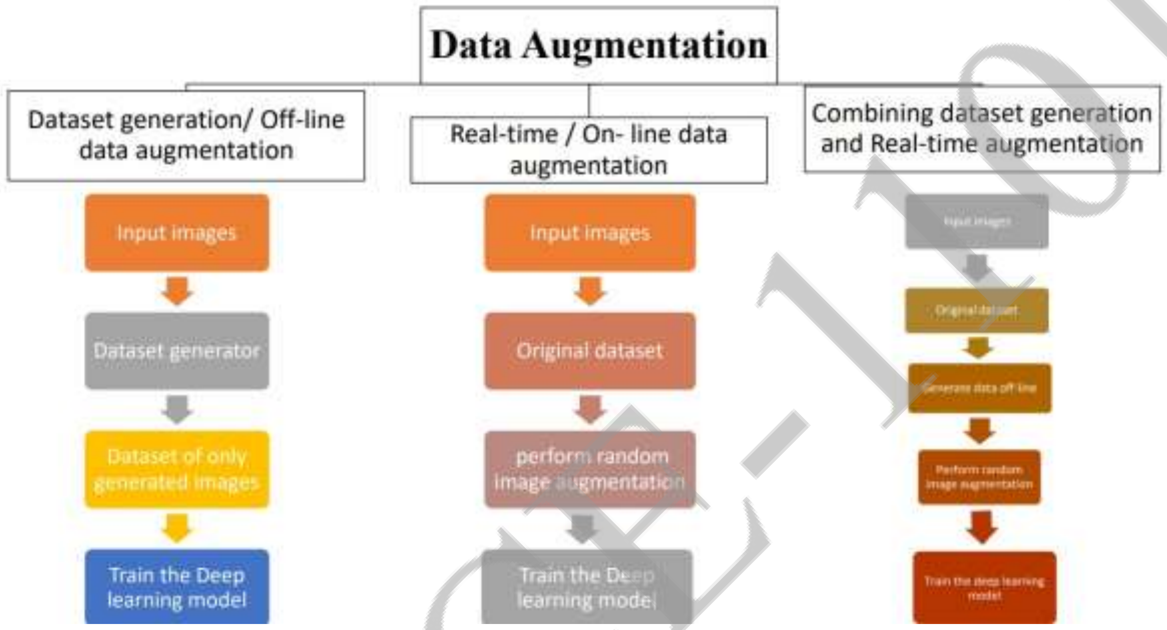
Regularization: Dataset Augmentation

Regularization techniques are essential for preventing overfitting in machine learning models, including neural networks.

Dataset augmentation is one such technique used to enhance the generalization ability of models by artificially increasing the size and diversity of the training dataset.



Heuristic data augmentation schemes often rely on the composition of a set of simple transformation functions (TFs) such as rotations and flips (see Figure). When chosen carefully, data augmentation schemes tuned by human experts can improve model performance. However, such heuristic strategies in practice can cause large variances in end model performance and may not produce augmentations needed for state-of-the-art models.


Data Augmentation
Dataset generation/ Off-line data augmentation
Real-time / On- line data augmentation
Combining dataset generation and Real-time augmentation
Input images
Dataset generator
Dataset of only generated images
Train the Deep learning model
Input images
Original dataset
perform random image augmentation
Train the Deep learning model
Input images
Original dataset
Generate data off line
Perform random image augmentation
Train the deep learning model

Data augmentation can be defined as the technique used to improve the diversity of the data by slightly modifying copies of already existing data or newly create synthetic data from the existing data. It is used to regularize the data and it also helps to reduce overfitting. Some of the techniques used for data augmentation are :
1. Rotation (Range 0-360 degrees)
2. flipping (true or false for horizontal flip and vertical flip)
3. Shear range (image is shifted along x-axis or y-axis)
4. Brightness or Contrast range (image is made lighter or darker)
5. Cropping (resize the image)
6. Scale (image is scaled outward or inward)
7. Saturation (depth or intensity of the image)
Here's how dataset augmentation works within the context of regularization:

Dataset Augmentation:

Dataset augmentation involves applying a variety of transformations to the original training data to create new, slightly modified samples. These transformations typically preserve the semantic content of the data while introducing variability that can help the model learn more robust and invariant features.

Common transformations include:

- **Geometric transformations:** Rotation, translation, scaling, cropping, and flipping of images.
- **Color transformations:** Adjusting brightness, contrast, saturation, and hue of images.
- **Noise injection:** Adding random noise to images or other data samples.
- **Random cropping and padding:** Extracting random crops or adding random padding to images.

By applying these transformations to the training data, the dataset is effectively expanded, providing the model with more diverse examples to learn from. This helps prevent overfitting by exposing the model to a wider range of variations in the data distribution.

Regularization Effect:

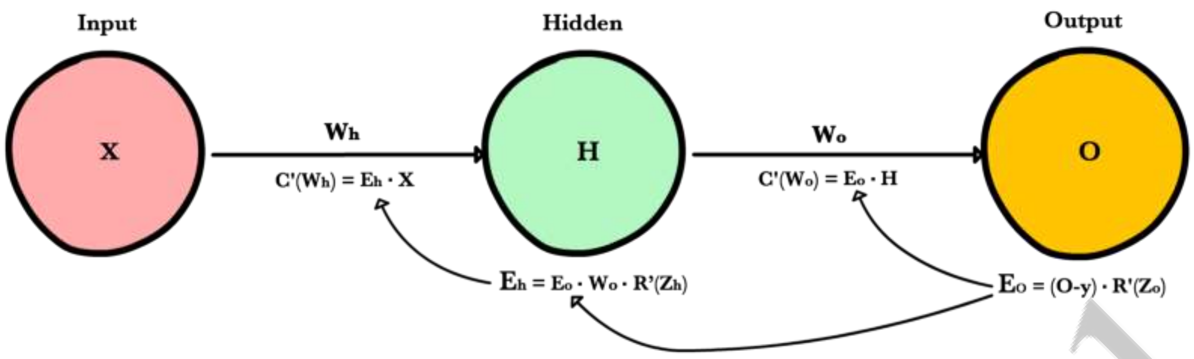
Dataset augmentation acts as a form of regularization by introducing noise and variability into the training process. This helps to prevent the model from memorizing the training examples and encourages it to learn more generalizable features that are invariant to the transformations applied during augmentation.

Additionally, dataset augmentation encourages the model to learn features that are robust to variations commonly encountered in real-world scenarios.
For example, by augmenting images with random rotations and translations, the model learns to recognize objects from different viewpoints and positions, leading to improved generalization performance.

Implementation:

Dataset augmentation is typically applied during the training phase, where each training sample is randomly transformed before being fed into the model for training. The transformed samples are treated as additional training data, effectively enlarging the training dataset.

Modern deep learning frameworks often provide built-in support for dataset augmentation through data preprocessing pipelines or dedicated augmentation modules. These frameworks allow users to easily specify the desired transformations and apply them to the training data on-the-fly during training.



Applying the chain rule

Let’s use the chain rule to calculate the derivative of cost with respect to any weight in the network. The chain rule will help us identify how much each weight contributes to our overall error and the direction to update each weight to reduce our error. Here are the equations we need to make a prediction and calculate total error, or cost:

Function	Formula	Derivative
Weighted input	$Z = XW$	$Z'(X) = W$ $Z'(W) = X$
ReLU activation	$R = max(0, Z)$	$R'(Z) = \begin{Bmatrix} 0 & Z < 0 \\ 1 & Z > 0 \end{Bmatrix}$
Cost function	$C = \frac{1}{2}(\hat{y} - y)^2$	$C'(\hat{y}) = (\hat{y} - y)$

Given a network consisting of a single neuron, total cost could be calculated as:

$$Cost = C(R(Z(XW)))$$

Using the chain rule we can easily find the derivative of Cost with respect to weight W.

$$\begin{aligned} C'(W) &= C'(R) \cdot R'(Z) \cdot Z'(W) \\ &= (\hat{y} - y) \cdot R'(Z) \cdot X \end{aligned}$$

Noise robustness

In the context of machine learning, and particularly deep learning, refers to the ability of a model to maintain its performance and make accurate predictions even when presented with noisy or corrupted input data. Noise in data can arise from various sources, including sensor errors, transmission errors, environmental factors, or imperfections in data collection processes.

Here's how noise robustness is addressed in machine learning, particularly in deep learning:

1. Data Preprocessing:

- **Noise Removal:** In some cases, it's possible to preprocess the data to remove or reduce noise before feeding it into the model. Techniques such as denoising filters, signal processing methods, or data cleaning algorithms can be employed to mitigate noise in the data.

2. Model Architecture:

- **Robust Architectures:** Designing models with architectures that are inherently robust to noise can help improve noise robustness. For example, architectures with skip connections or residual connections (e.g., ResNet) can help propagate information more effectively through the network, making them more resilient to noise.
- **Dropout:** Dropout regularization, which randomly drops units (along with their connections) during training, can act as a form of noise injection. This helps prevent overfitting and encourages the model to learn more robust features that are less sensitive to noise in the input data.

3. Data Augmentation:

- **Augmentation with Noise:** As mentioned earlier, dataset augmentation can help improve noise robustness by exposing the model to a wider range of data variations, including noisy samples. Augmenting the training data with artificially added noise can help the model learn to ignore irrelevant noise while focusing on the relevant signal in the data.

4. Training Strategies:

- **Adversarial Training:** Adversarial training involves training the model on adversarially perturbed examples generated by adding carefully crafted noise to the input data. This helps the model learn to be robust against adversarial attacks, which can be considered as a form of noise.

5. Uncertainty Estimation:

- **Probabilistic Models:** Probabilistic deep learning models, such as Bayesian neural networks or ensemble methods, can provide uncertainty estimates along with predictions. These uncertainty estimates can help the model recognize when it's uncertain about its predictions, which is particularly useful in the presence of noisy or ambiguous input data.

6. Transfer Learning:

- **Pretrained Models**: Transfer learning from pretrained models trained on large datasets can help improve noise robustness. Pretrained models have learned robust features from vast amounts of data, which can generalize well even in the presence of noise in the target domain.

Early Stopping, Bagging and Dropout

Early Stopping:
Early stopping is a regularization technique used to prevent overfitting during the training of machine learning models, including neural networks. The basic idea is to monitor the performance of the model on a separate validation set during training. Training is stopped early (i.e., before the model starts to overfit) when the performance on the validation set starts to degrade.
Specifically, early stopping involves:

- **Monitoring Validation Loss:** During training, the performance of the model is evaluated periodically on a validation set. The validation loss (or other evaluation metric) is calculated to assess the generalization performance of the model.
- **Stopping Criteria**: Training is stopped when the validation loss stops improving or starts to increase for a certain number of epochs. This prevents the model from overfitting to the training data.

Early stopping helps find the optimal point in the training process where the model generalizes best to unseen data, thus improving its ability to make accurate predictions on new samples.

Bagging (Bootstrap Aggregating):
Bagging is an ensemble learning technique that aims to improve the performance and robustness of machine learning models by combining predictions from multiple base models. It involves training multiple instances of the same base model on different subsets of the training data, typically using bootstrapping (sampling with replacement).

The key steps in bagging are:

- **Bootstrap Sampling**: Randomly sample subsets of the training data with replacement to create multiple training sets.
- **Base Model Training**: Train a base model (e.g., decision tree, neural network) on each bootstrap sample independently.
- **Combination of Predictions**: Combine the predictions of the base models by averaging (for regression) or voting (for classification) to make the final prediction.

Bagging helps reduce variance and improve the stability of predictions by leveraging the diversity of base models trained on different subsets of the data.

Pseudocode:

```
1. Given training data $(x_1, y_1)$, .... $(x_m, y_m)$
2. For t = 1, T:
        a. Form bootstrap replicate dataset S, by selecting m random examples
           from the training set with replacement.
        b. Let h, be the result of training base learning algorithm on $S_t$
Output Combined Classifier:
        $H(x) = Majority(h_1(x) \dots\dots h_t(x))$
```

Dropout:

Dropout is a regularization technique specifically designed for training neural networks to prevent overfitting. It involves randomly "dropping out" (i.e., deactivating) a fraction of neurons during training.

The key aspects of dropout are:

- **Random Deactivation:** During each training iteration, a fraction of neurons in the network is randomly set to zero with a probability p, typically chosen between 0.2 and 0.5.
- **Training and Inference:** Dropout is only applied during training. During inference (i.e., making predictions), all neurons are active, but their outputs are scaled by the dropout probability p to maintain the expected output magnitude.
- **Ensemble Effect:** Dropout can be interpreted as training an ensemble of exponentially many subnetworks, which encourages the network to learn more robust and generalizable features.

Dropout effectively prevents the co-adaptation of neurons and encourages the network to learn more distributed representations, leading to improved generalization performance.

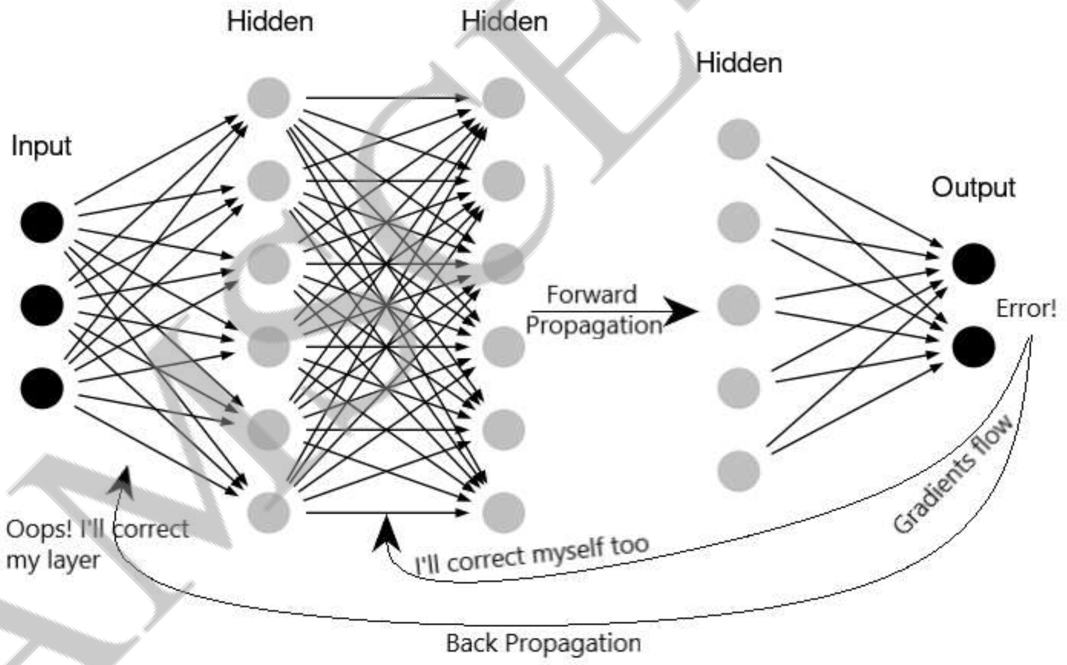
Note: These techniques—early stopping, bagging, and dropout—are powerful tools for preventing overfitting and improving the generalization performance of machine learning models, including neural networks. By incorporating these techniques into the training process, models can become more robust and reliable, making them better suited for real-world applications.

Batch Normalization

Batch normalization is a popular technique used in deep neural networks to stabilize and accelerate the training process. It addresses the problem of internal covariate shift, which refers to the change in the distribution of network activations during training due to changes in the parameters of earlier layers.

Here's how batch normalization works:

1. **Normalization**:
   During training, for each mini-batch of data fed into the network, batch normalization normalizes the activations of each layer to have zero mean and unit variance. This is achieved by subtracting the mean and dividing by the standard deviation computed over the mini-batch:
   $\hat{x} = \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}}$
   where $\hat{x}$ is the normalized output, x is the input, μ is the mean, σ^2 is the variance, and ϵ is a small constant added for numerical stability.
2. **Scaling and Shifting**:
   After normalization, the outputs are scaled and shifted by learnable parameters γ and β:
   $y = \gamma\hat{x} + \beta$
   where y is the final output.
3. **Training and Inference**:
   During training, the mean and variance are computed for each mini-batch and used to normalize the activations. However, during inference (i.e., making predictions), the mean and variance are typically computed over the entire training set or a moving average of the training data.



The normalization step is as follows:
1. Calculate the mean and variance of the activations for each feature in a mini-batch.
2. Normalize the activations of each feature by subtracting the mini-batch mean and dividing by the mini-batch standard deviation.
3. Scale and shift the normalized values using the learnable parameters gamma and beta, which allow the network to undo the normalization if that is what the learned behavior requires.

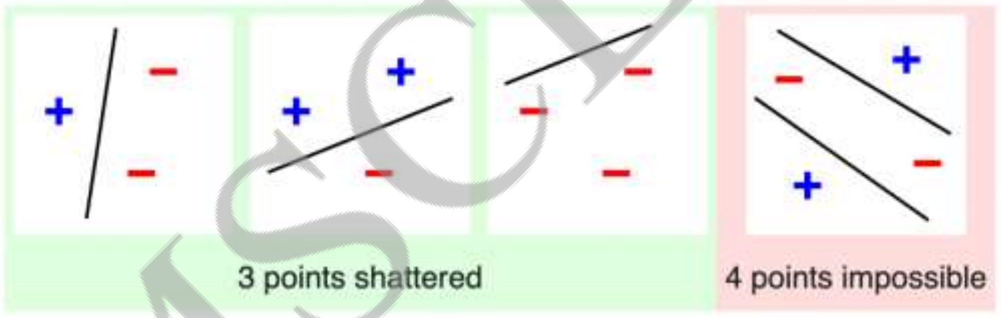
Benefits of Batch Normalization

Batch normalization offers several benefits to the training process of deep neural networks:

- **Improved Optimization**: It allows the use of higher learning rates, speeding up the training process by reducing the careful tuning of parameters.
- **Regularization**: It adds a slight noise to the activations, similar to dropout. This can help to regularize the model and reduce overfitting.
- **Reduced Sensitivity to Initialization**: It makes the network less sensitive to the initial starting weights.
- **Allows Deeper Networks**: By reducing internal covariate shift, batch normalization allows for the training of deeper networks.

VC Dimension and Neural Nets

The Vapnik-Chervonenkis (VC) dimension is a concept from statistical learning theory that provides a measure of the capacity or complexity of a hypothesis space—the set of all possible functions that a learning algorithm can choose from to fit the training data. In the context of neural networks, the VC dimension plays an important role in understanding the expressiveness and generalization ability of different network architectures.



Shattering set of examples:

Assume a binary classification problem with N examples RD and consider the set of $2^{|N|}$ possible dichotomies. For instance, with N = 3 examples, set of all possible dichotomies is {(000), (001), (010), (011), (100), (101), (110), (111)}. A class of functions is said to shatter the dataset if, for every possible dichotomy, there is a function $f(\alpha)$ that models it.

Consider as an example a finite concept class C = {c_1,...,c_4} applied to three instance vectors with the results :

	X_1	X_2	X_3
C_1	1	1	1
C_2	0	1	1
C_3	1	0	0
C_4	0	0	0

Then:

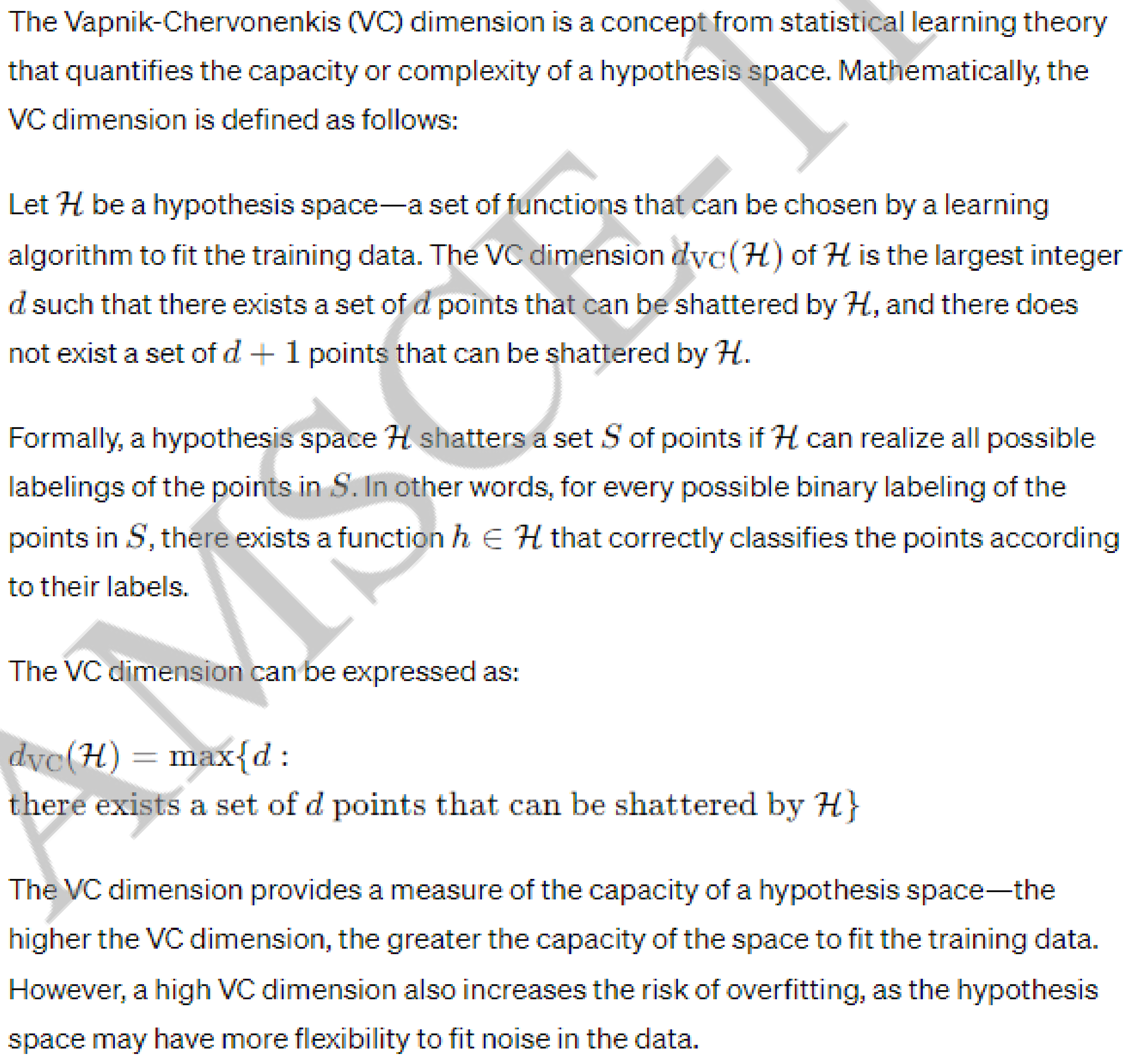
$\pi_c(\{x_1\})$ = {(0), (1)}

π_c ($\{x_1, x_3\}$) = {(0,0), (0, 1), (1,0), (1, 1)}

π_c ($\{x_2, x_3\}$) = {(0,0), (1,1)}

- VC dimension VC(f) is the size of the largest dataset that can be shattered by the set of function $f(\alpha)$.
- If the VC Dimension of (α) is h, then there exists at least one set of h points that can be shattered by (α), but in general it will not be true that every set of h points can be shattered.
- VC dimension cannot be accurately estimated for non-linear models such as neural networks. The VC dimension may be infinite requiring an infinite amount of data.

VC Dimension for Neural Networks



The Vapnik-Chervonenkis (VC) dimension is a concept from statistical learning theory that quantifies the capacity or complexity of a hypothesis space. Mathematically, the VC dimension is defined as follows:

Let $\mathcal{H}$ be a hypothesis space—a set of functions that can be chosen by a learning algorithm to fit the training data. The VC dimension $d_{\mathrm{VC}}(\mathcal{H})$ of $\mathcal{H}$ is the largest integer d such that there exists a set of d points that can be shattered by $\mathcal{H}$, and there does not exist a set of $d+1$ points that can be shattered by $\mathcal{H}$.

Formally, a hypothesis space $\mathcal{H}$ shatters a set S of points if $\mathcal{H}$ can realize all possible labelings of the points in S. In other words, for every possible binary labeling of the points in S, there exists a function $h \in \mathcal{H}$ that correctly classifies the points according to their labels.

The VC dimension can be expressed as:

$$d_{\mathrm{VC}}(\mathcal{H}) = \max\{d : \text{there exists a set of } d \text{ points that can be shattered by } \mathcal{H}\}$$

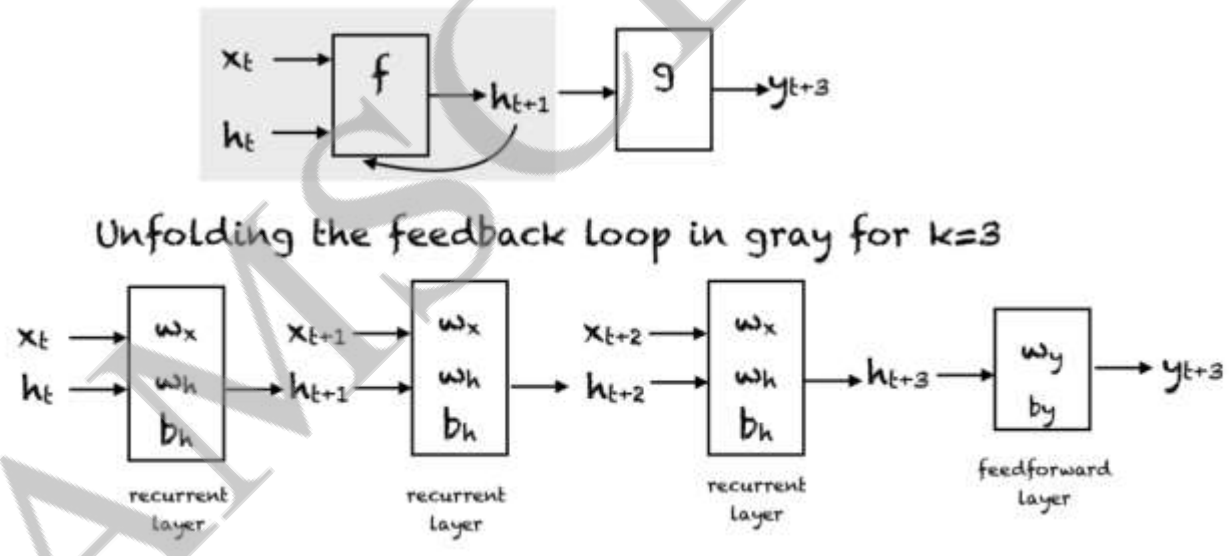
The VC dimension provides a measure of the capacity of a hypothesis space—the higher the VC dimension, the greater the capacity of the space to fit the training data. However, a high VC dimension also increases the risk of overfitting, as the hypothesis space may have more flexibility to fit noise in the data.

Unit 5

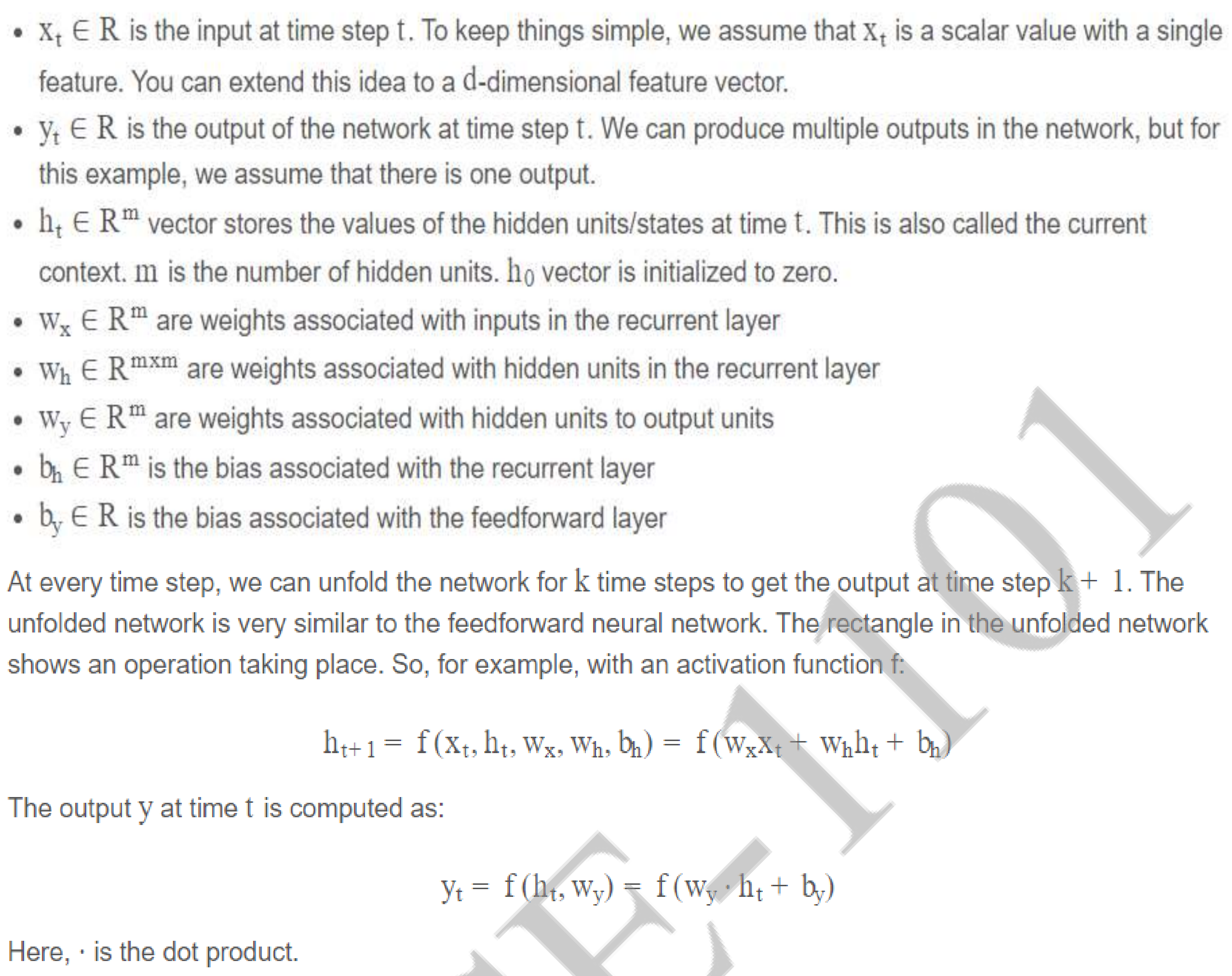
Recurrent Neural Networks: Introduction – Recursive Neural Networks – Bidirectional RNNs – Deep Recurrent Networks – Applications: Image Generation, Image Compression, Natural Language Processing. Complete Auto encoder, Regularized Autoencoder, Stochastic Encoders and Decoders, Contractive Encoders.

Recurrent Neural Networks: Introduction

- Recurrent Neural Networks (RNNs) are a type of artificial neural network designed to effectively deal with sequential data, where the order of elements matters.
- Unlike feedforward neural networks, where the flow of data is strictly forward, RNNs have connections that form directed cycles, allowing them to exhibit dynamic temporal behavior.
- This makes RNNs particularly suitable for tasks such as time series prediction, natural language processing (NLP), speech recognition, and more.
- However, if we have data in a sequence such that one data point depends upon the previous data point, we need to modify the neural network to incorporate the dependencies between these data points.
- RNNs have the concept of "memory" that helps them store the states or information of previous inputs to generate the next output of the sequence.



A simple RNN has a feedback loop, as shown in the first diagram of the above figure.
The feedback loop shown in the gray rectangle can be unrolled in three-time steps to produce the second network of the above figure. Of course, you can vary the architecture so that the network unrolls *k* time steps. In the figure, the following notation is used:



- $x_t \in R$ is the input at time step t. To keep things simple, we assume that x_t is a scalar value with a single feature. You can extend this idea to a d-dimensional feature vector.
- $y_t \in R$ is the output of the network at time step t. We can produce multiple outputs in the network, but for this example, we assume that there is one output.
- $h_t \in R^m$ vector stores the values of the hidden units/states at time t. This is also called the current context. m is the number of hidden units. h_0 vector is initialized to zero.
- $w_x \in R^m$ are weights associated with inputs in the recurrent layer
- $w_h \in R^{mxm}$ are weights associated with hidden units in the recurrent layer
- $w_y \in R^m$ are weights associated with hidden units to output units
- $b_h \in R^m$ is the bias associated with the recurrent layer
- $b_y \in R$ is the bias associated with the feedforward layer

At every time step, we can unfold the network for k time steps to get the output at time step $k + 1$. The unfolded network is very similar to the feedforward neural network. The rectangle in the unfolded network shows an operation taking place. So, for example, with an activation function f:

$$h_{t+1} = f(x_t, h_t, w_x, w_h, b_h) = f(w_x x_t + w_h h_t + b_h)$$

The output y at time t is computed as:

$$y_t = f(h_t, w_y) = f(w_y \cdot h_t + b_y)$$

Here, $\cdot$ is the dot product.

Hence, in the feedforward pass of an RNN, the network computes the values of the hidden units and the output after *k* time steps. The weights associated with the network are shared temporally.
Each recurrent layer has two sets of weights:

- One for the input
- Second for the hidden unit
- The last feedforward layer, which computes the final output for the kth time step, is just like an ordinary layer of a traditional feedforward network.

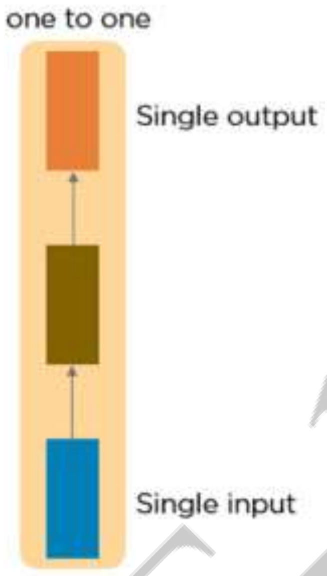
Why Recurrent Neural Networks?

Recurrent Neural Networks have unique capacities as opposed to other kinds of Neural Networks, which open a wide range of possibilities for their users still also bringing some challenges with them. Then's a rundown of the main benefits

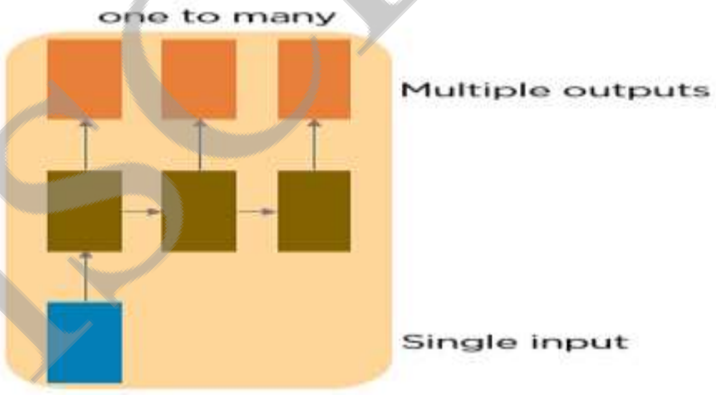
- It's the only neural network with memory and binary data processing.
- It can plan out several inputs and productions. Unlike other algorithms that deliver one product for one input, the benefit of RNN is that it can plot out many to many, one to many, and many to one input and productions.

Types of Recurrent Neural Networks
There are four types of Recurrent Neural Networks:

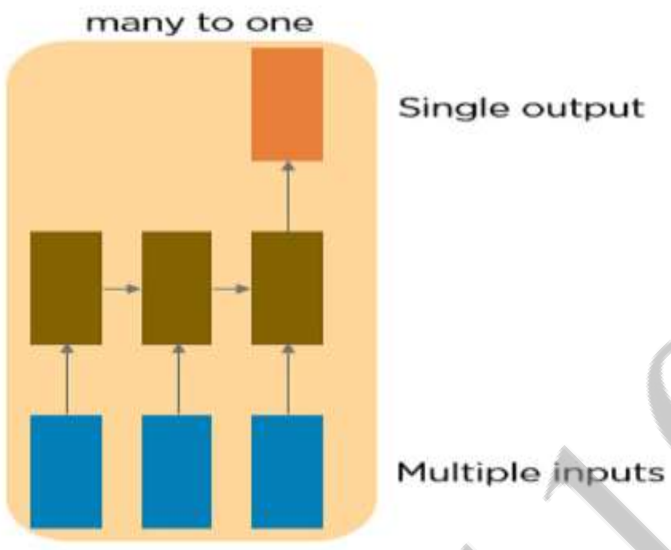
- **One to One**
  This type of neural network is understood because the Vanilla Neural Network. It's used for general machine learning problems, which contains a single input and one output.



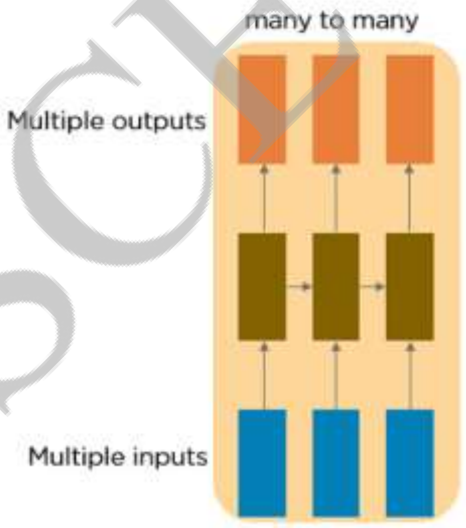
- **One to Many**
  This type of neural network incorporates a single input and multiple outputs. An example of this is often the image caption.



- **Many to One**
  This RNN takes a sequence of inputs and generates one output. Sentiment analysis may be a example of this sort of network where a given sentence are often classified as expressing positive or negative sentiments.



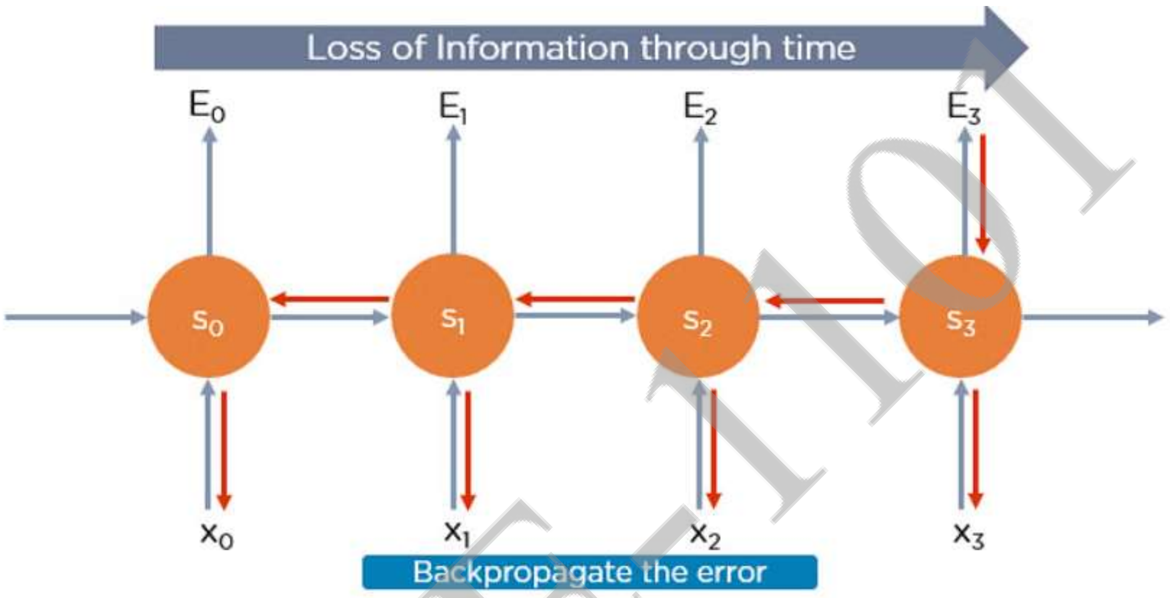
- **Many to Many**
  This RNN takes a sequence of inputs and generates a sequence of outputs. artificial intelligence is one among the examples.



Two Issues of Standard RNNs
1. Vanishing Gradient Problem

- Recurrent Neural Networks enable you to model time-dependent and sequential data problems, like stock exchange prediction, artificial intelligence, and text generation. you'll find, however, RNN is tough to train due to the gradient problem.

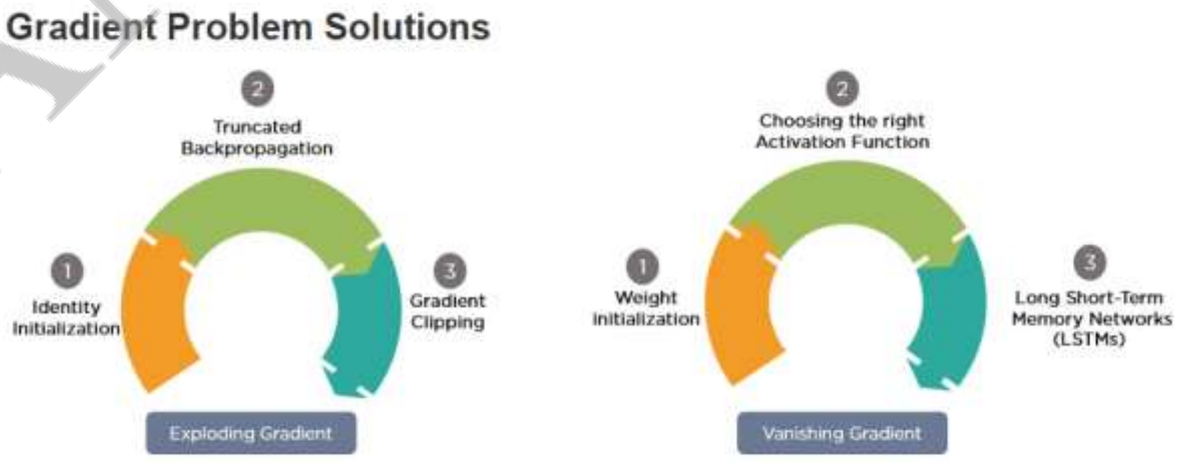
- RNNs suffer from the matter of vanishing gradients. The gradients carry information utilized in the RNN, and when the gradient becomes too small, the parameter updates become insignificant. This makes the training of long data sequences difficult.



2. Exploding Gradient Problem

- While training a neural network, if the slope tends to grow exponentially rather than decaying, this is often called an Exploding Gradient.

- This problem arises when large error gradients accumulate, leading to very large updates to the neural network model weights during the training process.


Gradient Problem Solutions
1 Identity Initialization
2 Truncated Backpropagation
3 Gradient Clipping
Exploding Gradient
1 Weight Initialization
2 Choosing the right Activation Function
3 Long Short-Term Memory Networks (LSTMs)
Vanishing Gradient

Now, let’s discuss the foremost popular and efficient thanks to cope with gradient problems, i.e., Long immediate memory Network (LSTMs).

First, let’s understand Long-Term Dependencies.

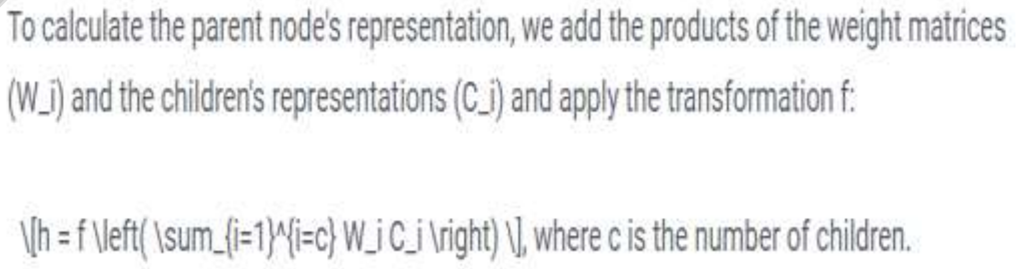
Suppose you wish to predict the last word within the text: “The clouds are within the ______.”
The most obvious answer to the present is that the “sky.” We don’t need from now on context to predict the last word within the above sentence.
Consider this sentence: “I are staying in Spain for the last 10 years…I can speak fluent ______.”
The word you are expecting will rely on the previous couple of words in context. Here, you would like the context of Spain to predict the last word within the text, and also the most fitted answer to the present sentence is “Spanish.” The gap between the relevant information and the point where it’s needed may became very large. LSTMs facilitate to solve this problem.

****

Recursive Neural Networks (ReNNs)

Recursive Neural Networks (ReNNs) are a type of neural network architecture designed to process structured data, such as hierarchical data structures or recursive structures. Unlike traditional feedforward or recurrent neural networks, which operate on fixed-sized input vectors or sequences, ReNNs operate on tree-like or graph-like structures, allowing them to model relationships between elements in a hierarchical manner.

Due to their deep tree-like structure, Recursive Neural Networks can handle hierarchical data. The tree structure means combining child nodes and producing parent nodes. Each child-parent bond has a weight matrix, and similar children have the same weights. The number of children for every node in the tree is fixed to enable it to perform recursive operations and use the same weights. RvNNs are used when there's a need to parse an entire sentence.

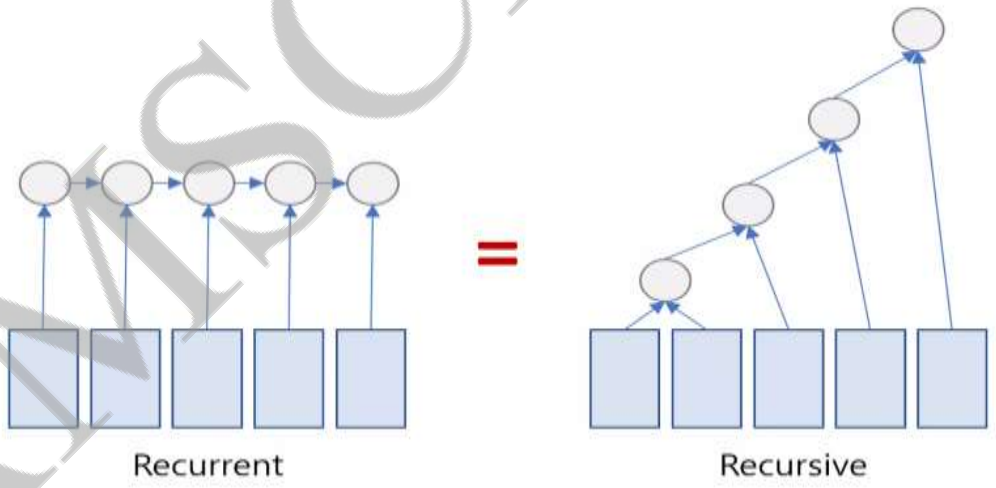


To calculate the parent node's representation, we add the products of the weight matrices (W_i) and the children's representations (C_i) and apply the transformation f:

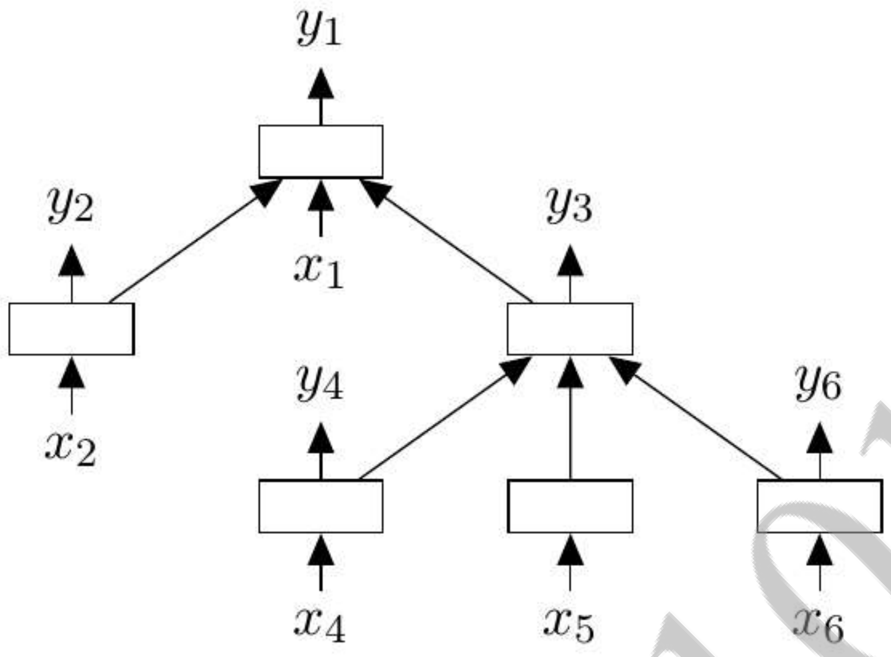
$$h = f \left(\sum_{i=1}^{i=c} W_i C_i \right)$$, where c is the number of children.

Difference between Recurrent neural network and recursive neural networks

Aspect	Recurrent Neural Networks (RNNs)	Recursive Neural Networks (ReNNs)
Architecture	Sequential architecture, nodes connected to previous time steps	Hierarchical or recursive architecture, nodes connected in a tree-like or graph-like structure.
Data Structure	Operates on sequential data where order matters	Handles structured data with hierarchical or recursive relationships
Training	Typically trained using backpropagation through time (BPTT)	May involve specialized algorithms for handling the recursive structure (e.g., BPTS)
Applications	Language modeling, machine translation, sentiment analysis, time series prediction	Parsing syntactic or semantic structures in NLP, analyzing hierarchical structures in images or videos, processing hierarchical data in bioinformatics



A Recursive Neural Networks is more like a hierarchical network where there is really no time aspect to the input sequence but the input has to be processed hierarchically in a tree fashion. Here is an example of how a recursive neural network looks. It shows the way to learn a parse tree of a sentence by recursively taking the output of the operation performed on a smaller chunk of the text.



- The children of each parent node are just a node like that node. RvNNs comprise a class of architectures that can work with structured input. The network looks at a series of inputs, each time at x1, x2… and prints the results of each of these inputs.
- This means that the output depends on the number of neurons in each layer of the network and the number of connections between them. The simplest form of a RvNNs, the vanilla RNG, resembles a regular neural network. Each layer contains a loop that allows the model to transfer the results of previous neurons from another layer.
- Schematically, RvNN layer uses a loop to iterate through a timestamp sequence while maintaining an internal state that encodes all the information about that timestamp it has seen so far.

Features of Recursive Neural Networks

- A recursive neural network is created in such a way that it includes applying same set of weights with different graph like structures.
- The nodes are traversed in topological order.
- This type of network is trained by the reverse mode of automatic differentiation.
- Natural language processing includes a special case of recursive neural networks.
- This recursive neural tensor network includes various composition functional nodes in the tree.

Challenges:

While Recursive Neural Networks offer advantages for modelling structured data, they also come with challenges:

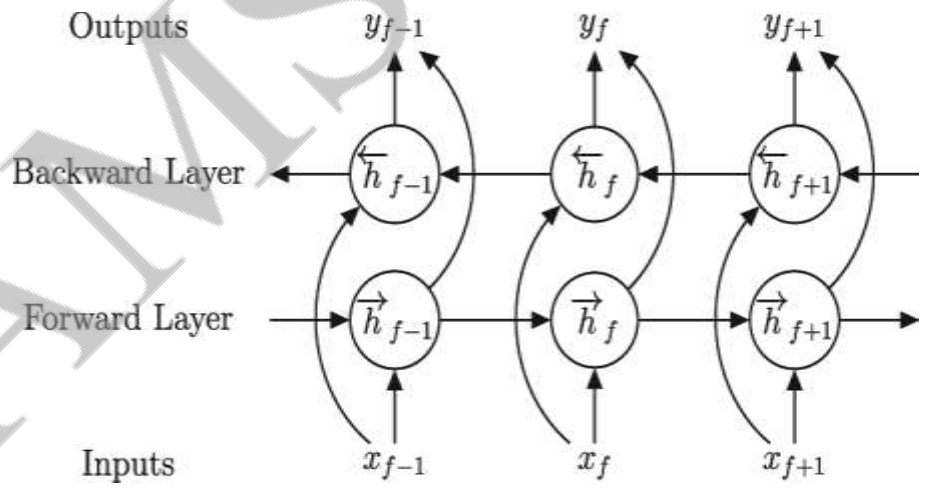
- **Computational Complexity:** Processing recursive structures can be computationally expensive, especially for deep trees or graphs with many nodes.
- **Data Representation:** Representing complex structures in a fixed-dimensional vector space can be challenging, especially for structures with varying sizes or irregularities.
- **Training Difficulty:** Training ReNNs may require specialized algorithms and techniques to handle the recursive nature of the network and mitigate issues such as vanishing gradients.

Bidirectional Recurrent Neural Networks (Bi-RNNs)

Bidirectional Recurrent Neural Networks (Bi-RNNs) are an extension of traditional Recurrent Neural Networks (RNNs) that can capture both past and future information at each time step. In standard RNNs, the prediction at a given time step depends only on the past history of the sequence. However, in many applications, it's beneficial to consider both past and future context to make better predictions.
The architecture of a Bidirectional RNN involves two separate recurrent layers:

1. One processing the input sequence in the forward direction
2. Another processing the sequence in the backward direction.

Each layer computes hidden states at each time step, considering information from both past and future context. The final output at each time step is typically a concatenation of the forward and backward hidden states.



Working of Bidirectional Recurrent Neural Network

Inputting a sequence:
A sequence of data points, each represented as a vector with the same dimensionality, are fed into a BRNN. The sequence might have different lengths.

Dual Processing:

Both the forward and backward directions are used to process the data. On the basis of the input at that step and the hidden state at step t-1, the hidden state at time step t is determined in the forward direction. The input at step t and the hidden state at step t+1 are used to calculate the hidden state at step t in a reverse way.

Computing the hidden state:

A non-linear activation function on the weighted sum of the input and previous hidden state is used to calculate the hidden state at each step. This creates a memory mechanism that enables the network to remember data from earlier steps in the process.

Determining the output:

A non-linear activation function is used to determine the output at each step from the weighted sum of the hidden state and a number of output weights. This output has two options: it can be the final output or input for another layer in the network.

Training:

The network is trained through a supervised learning approach where the goal is to minimize the discrepancy between the predicted output and the actual output. The network adjusts its weights in the input-to-hidden and hidden-to-output connections during training through backpropagation.
To calculate the output from an RNN unit, we use the following formula:

$$H_t \text{ (Forward)} = A(X_t * W_{XH} \text{ (forward)} + H_{t-1} \text{ (Forward)} * W_{HH} \text{ (Forward)} + b_H \text{ (Forward)}$$
$$H_t \text{ (Backward)} = A(X_t * W_{XH} \text{ (Backward)} + H_{t+1} \text{ (Backward)} * W_{HH} \text{ (Backward)} + b_H \text{ (Backward)}$$

where,
A = activation function, W = weight matrix, b = bias

The training of a BRNN is similar to backpropagation through a time algorithm. BPTT algorithm works as follows:

- Roll out the network and calculate errors at each iteration
- Update weights and roll up the network.

However, because forward and backward passes in a BRNN occur simultaneously, updating the weights for the two processes may occur at the same time. This produces inaccurate outcomes. Thus, the following approach is used to train a BRNN to accommodate forward and backward passes individually.

Advantages of Bidirectional RNN

- **Context from both past and future:**
  With the ability to process sequential input both forward and backward, BRNNs provide a thorough grasp of the full context of a sequence. Because of this, BRNNs are effective at tasks like sentiment analysis and speech recognition.

- **Enhanced accuracy:**
  BRNNs frequently yield more precise answers since they take both historical and upcoming data into account.

- **Efficient handling of variable-length sequences:**
  When compared to conventional RNNs, which require padding to have a constant length, BRNNs are better equipped to handle variable-length sequences.

- **Resilience to noise and irrelevant information:**
  BRNNs may be resistant to noise and irrelevant data that are present in the data. This is so because both the forward and backward paths offer useful information that supports the predictions made by the network.

- **Ability to handle sequential dependencies:**
  BRNNs can capture long-term links between sequence pieces, making them extremely adept at handling complicated sequential dependencies.

Applications of Bidirectional Recurrent Neural Network

Bi-RNNs have been applied to various natural language processing (NLP) tasks, including:

- **Sentiment Analysis:**
  By taking into account both the prior and subsequent context, BRNNs can be utilized to categorize the sentiment of a particular sentence.

- **Named Entity Recognition:**
  By considering the context both before and after the stated thing, BRNNs can be utilized to identify those entities in a sentence.

- **Part-of-Speech Tagging:**
  The classification of words in a phrase into their corresponding parts of speech, such as nouns, verbs, adjectives, etc., can be done using BRNNs.

- **Machine Translation:**
  BRNNs can be used in encoder-decoder models for machine translation, where the decoder creates the target sentence and the encoder analyses the source sentence in both directions to capture its context.

- **Speech Recognition:**
  When the input voice signal is processed in both directions to capture the contextual information, BRNNs can be used in automatic speech recognition systems.

Disadvantages of Bidirectional RNN

- **Computational complexity:**
  Given that they analyze data both forward and backward, BRNNs can be computationally expensive due to the increased amount of calculations needed.

- **Long training time:**
  BRNNs can also take a while to train because there are many parameters to optimize, especially when using huge datasets.

- **Difficulty in parallelization:**
  Due to the requirement for sequential processing in both the forward and backward directions, BRNNs can be challenging to parallelize.

- **Overfitting:**
  BRNNs are prone to overfitting since they include many parameters that might result in too complicated models, especially when trained on short datasets.

- **Interpretability:**
  Due to the processing of data in both forward and backward directions, BRNNs can be tricky to interpret since it can be difficult to comprehend what the model is doing and how it is producing predictions.

Deep recurrent networks (DRNs)

- Deep recurrent networks (DRNs) are a class of neural networks that combine the concepts of deep learning and recurrent neural networks (RNNs).

- RNNs are a type of neural network designed to work with sequential data, where the output of each step is dependent on the previous steps.

- This makes them particularly suitable for tasks like natural language processing (NLP), time series prediction, and speech recognition.

- Deep recurrent networks extend the capabilities of traditional RNNs by stacking multiple layers of recurrent units, allowing for the creation of deeper architectures.

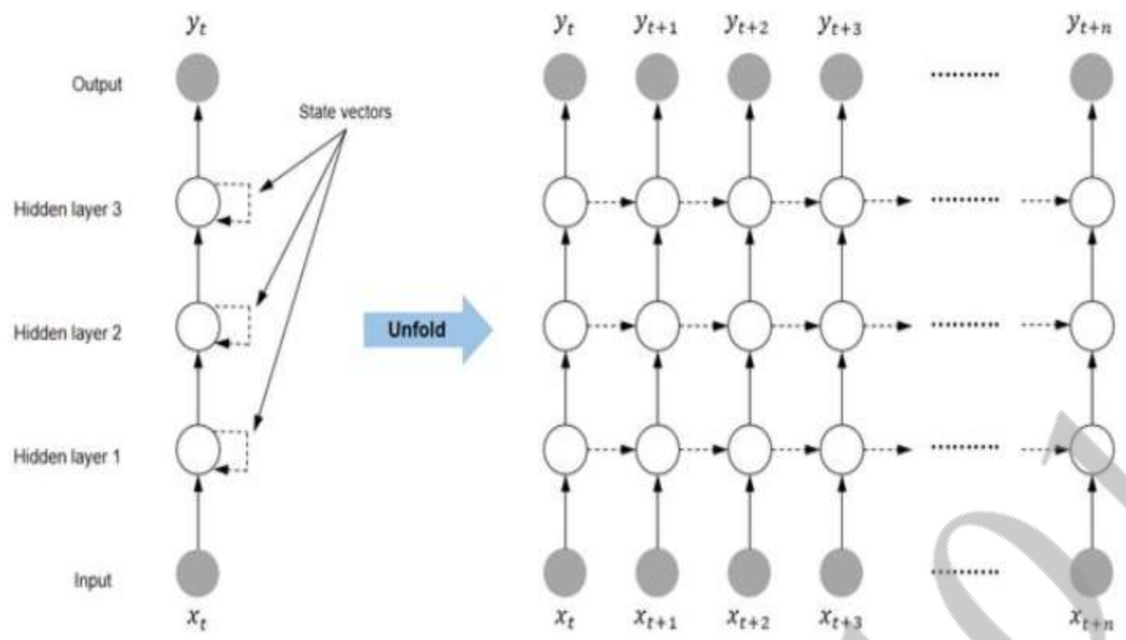
- Each layer in a DRN passes its output as input to the next layer, enabling the network to learn hierarchical representations of sequential data.

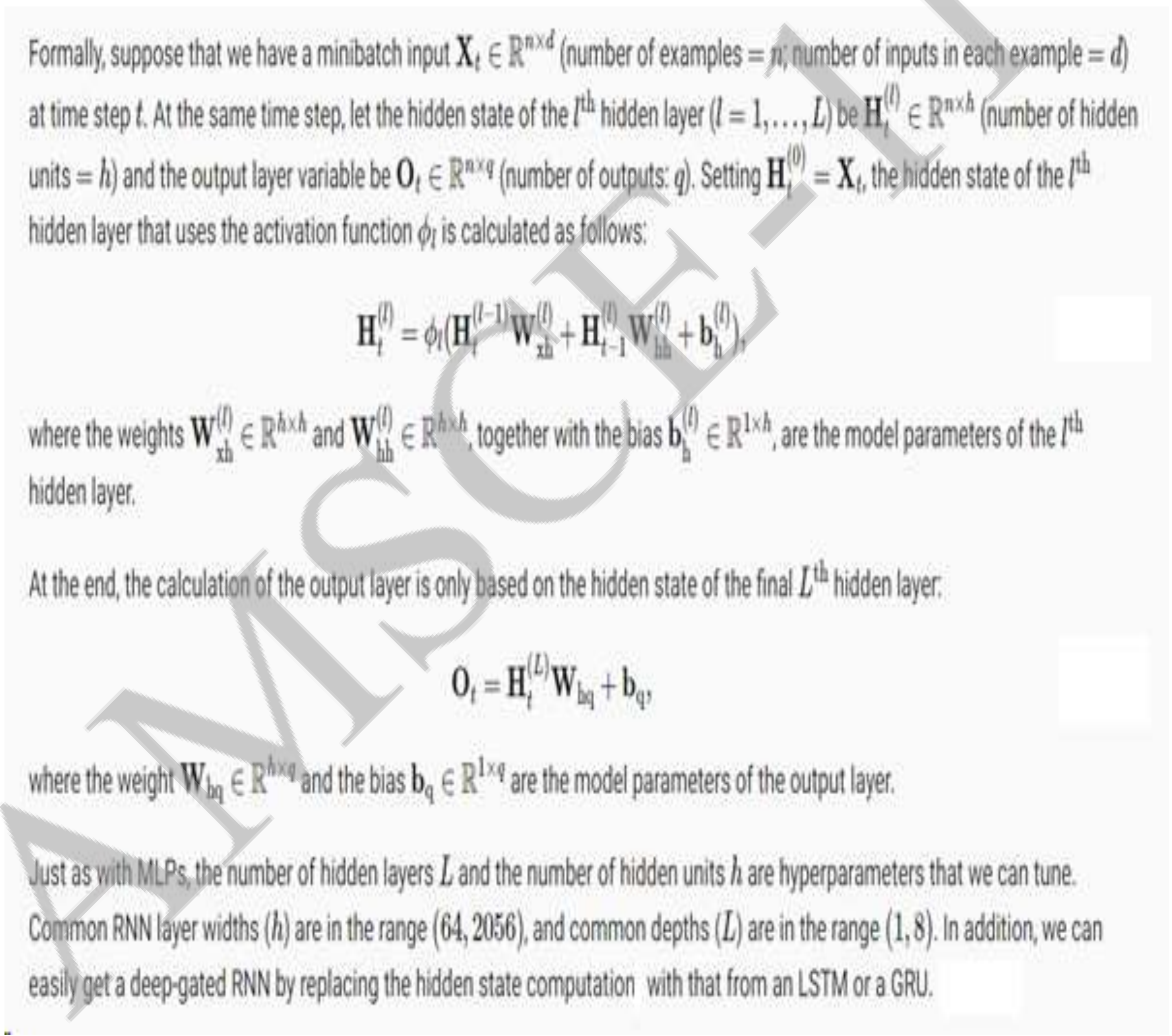
- Deep recurrent networks have been successfully applied to various tasks, including sequence prediction, language modeling, machine translation, and speech recognition.

- They have demonstrated superior performance compared to shallow recurrent networks in many cases, especially when dealing with complex sequential data with long-range dependencies.

There are several types of recurrent units that can be used in deep recurrent networks, such as:

- **Vanilla RNNs:**
  These are the simplest form of recurrent units, where the output is computed based on the current input and the previous hidden state.
- **Long Short-Term Memory (LSTM):**
  LSTMs are a type of recurrent unit that introduces gating mechanisms to control the flow of information within the network, allowing it to learn long-range dependencies more effectively and mitigate the vanishing gradient problem.
- **Gated Recurrent Units (GRUs):**
  GRUs are like LSTMs but have a simpler structure with fewer parameters, making them computationally more efficient.


y_t
Output
State vectors
Hidden layer 3
Hidden layer 2
Hidden layer 1
Input
x_t
Unfold
y_t y_{t+1} y_{t+2} y_{t+3} y_{t+n}
x_t x_{t+1} x_{t+2} x_{t+3} x_{t+n}


Formally, suppose that we have a minibatch input $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ (number of examples = n; number of inputs in each example = d) at time step t. At the same time step, let the hidden state of the l^{th} hidden layer ($l = 1,\dots,L$) be $\mathbf{H}_t^{(l)} \in \mathbb{R}^{n\times h}$ (number of hidden units = h) and the output layer variable be $\mathbf{O}_t \in \mathbb{R}^{n\times q}$ (number of outputs: q). Setting $\mathbf{H}_t^{(0)} = \mathbf{X}_t$, the hidden state of the l^{th} hidden layer that uses the activation function ϕ_l is calculated as follows:

$$\mathbf{H}_t^{(l)} = \phi_l(\mathbf{H}_t^{(l-1)}\mathbf{W}_{\text{xh}}^{(l)} + \mathbf{H}_{t-1}^{(l)}\mathbf{W}_{\text{hh}}^{(l)} + \mathbf{b}_{\text{h}}^{(l)}),$$

where the weights $\mathbf{W}_{\text{xh}}^{(l)} \in \mathbb{R}^{h\times h}$ and $\mathbf{W}_{\text{hh}}^{(l)} \in \mathbb{R}^{h\times h}$, together with the bias $\mathbf{b}_{\text{h}}^{(l)} \in \mathbb{R}^{1\times h}$, are the model parameters of the l^{th} hidden layer.

At the end, the calculation of the output layer is only based on the hidden state of the final L^{th} hidden layer:

$$\mathbf{O}_t = \mathbf{H}_t^{(L)}\mathbf{W}_{\text{hq}} + \mathbf{b}_{\text{q}},$$

where the weight $\mathbf{W}_{\text{hq}} \in \mathbb{R}^{h\times q}$ and the bias $\mathbf{b}_{\text{q}} \in \mathbb{R}^{1\times q}$ are the model parameters of the output layer.

Just as with MLPs, the number of hidden layers L and the number of hidden units h are hyperparameters that we can tune. Common RNN layer widths (h) are in the range $(64, 2056)$, and common depths (L) are in the range $(1, 8)$. In addition, we can easily get a deep-gated RNN by replacing the hidden state computation with that from an LSTM or a GRU.

Steps to develop a deep RNN application

Developing an end-to-end deep RNN application involves several steps, including data preparation, model architecture design, training the model, and deploying it. Here is an example of an end-to-end deep RNN application for sentiment analysis.

Data preparation:
The first step is to gather and preprocess the data. In this case, we'll need a dataset of text reviews labelled with positive or negative sentiment. The text data needs to be cleaned, tokenized, and converted to the numerical format. This can be done using libraries like NLTK or spaCy in Python.

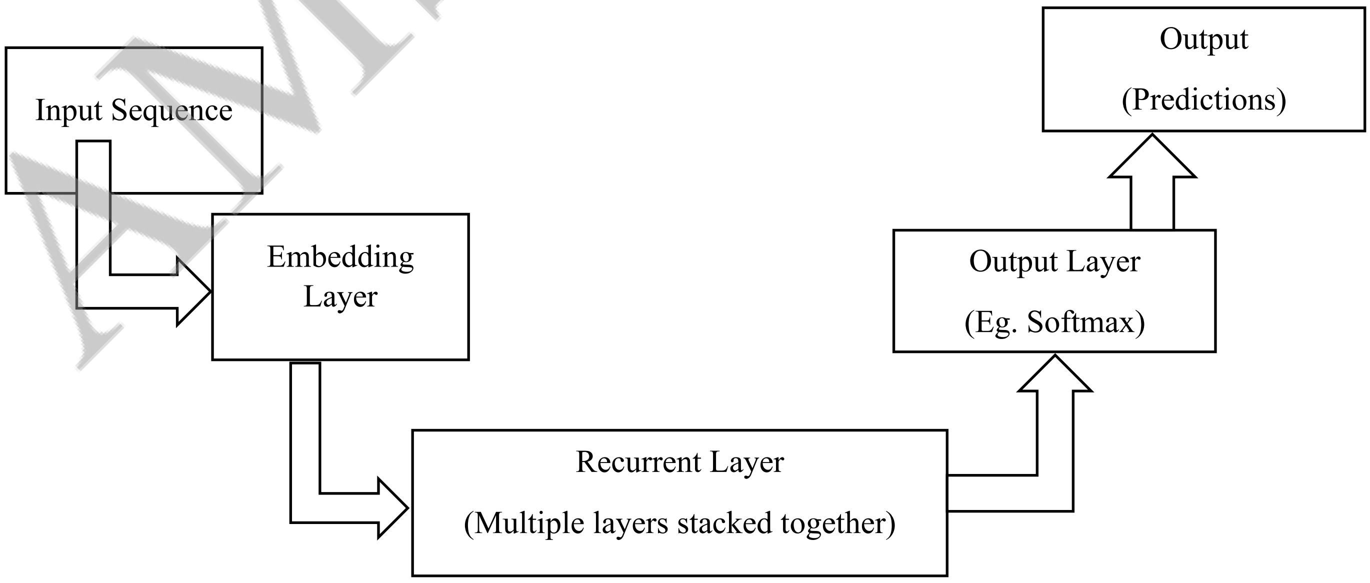
Model architecture design:
The next step is to design the deep RNN architecture. We'll need to decide on the number of layers, number of hidden units, and type of recurrent unit (e.g. LSTM or GRU). We'll also need to decide how to handle the input and output sequences, such as using padding or truncation.

Training the model:
Once the architecture is designed, we'll need to train the model using the preprocessed data. We'll split the data into training and validation sets and train the model using an optimization algorithm like stochastic gradient descent. We'll also need to set hyperparameters like learning rate and batch size.

Evaluating the model:
After training, we'll evaluate the model's performance on a separate test set. We'll use metrics like accuracy, precision, recall, and F1 score to assess the model's performance.

Deploying the model:
Finally, we'll deploy the trained model to a production environment, where it can be used to classify sentiment in real-time. This could involve integrating the model into a web application or API.

Processing Diagram of Deep Recurrent Networks



This block diagram provides a high-level overview of the architecture of a deep recurrent network.

- **Input Sequence:**
  This is the sequential data fed into the network. It could be text, time-series data, audio, etc.

- **Embedding Layer:**
  Converts the input sequence into a dense representation suitable for processing by the recurrent layers. It typically involves mapping each element of the sequence (e.g., word or data point) to a high-dimensional vector space.

- **Recurrent Layers:**
  Consist of multiple recurrent units stacked together. Each layer processes the input sequence sequentially, capturing temporal dependencies. Common types of recurrent units include vanilla RNNs, LSTMs, and GRUs.

- **Output Layer:**
  Takes the output from the recurrent layers and produces the final prediction or output. The structure of this layer depends on the specific task, such as classification (e.g., softmax activation) or regression (e.g., linear activation).

- **Output (Prediction):**
  The final output of the network, which could be a sequence of predictions for each time step or a single prediction for the entire sequence, depending on the task.

Deep recurrent networks (DRNs) offer several advantages:

- **Hierarchical Representation Learning:**
  With multiple layers of recurrent units, DRNs can learn hierarchical representations of sequential data. Each layer can capture different levels of abstraction, allowing the network to extract complex features from the input sequence.
- **Modeling Long-term Dependencies:**
  Deep architectures enable DRNs to capture long-range dependencies in sequential data more effectively. By stacking recurrent layers, the network can maintain and propagate information over longer sequences, which is crucial for tasks involving context or memory over extended periods.
- **Improved Expressiveness:**
  Deeper architectures provide more expressive power, allowing DRNs to learn complex patterns and relationships within sequential data. This increased expressiveness can lead to better performance on tasks that require modeling intricate dependencies or understanding subtle variations in the data.

- **Better Feature Abstraction:**
  Each layer in a DRN learns to abstract features from the input sequence, leading to a hierarchy of representations. This hierarchical feature extraction can facilitate learning informative and discriminative features, which are essential for tasks like sequence classification, language modeling, and machine translation.

- **Transfer Learning:**
  Pre-training deep recurrent networks on large-scale datasets for related tasks (e.g., language modeling) and fine-tuning them for specific tasks often leads to improved performance. The hierarchical representations learned during pre-training capture generic features of the data, which can be beneficial for downstream tasks with limited labeled data.

Disadvantages of Deep recurrent networks (DRNs)

- **Vanishing/Exploding Gradient Problem:**
  Training deep recurrent networks can be challenging due to the vanishing or exploding gradient problem. As gradients are backpropagated through multiple layers during training, they can become either extremely small (vanishing) or extremely large (exploding), which hinders learning and stability. Techniques like gradient clipping and careful initialization of weights are often necessary to mitigate this issue.

- **Computational Complexity:**
  Deep recurrent networks with multiple layers can be computationally expensive to train and deploy, especially when dealing with large-scale datasets or complex architectures. The computational complexity increases with the number of layers, making it challenging to train deep models on resource-constrained devices or in real-time applications.

- **Long Training Time:**
  Training deep recurrent networks requires significant computational resources and time, especially when dealing with large datasets and complex architectures. The training process often involves multiple iterations over the entire dataset, which can take hours, days, or even weeks depending on the size of the data and the complexity of the model.

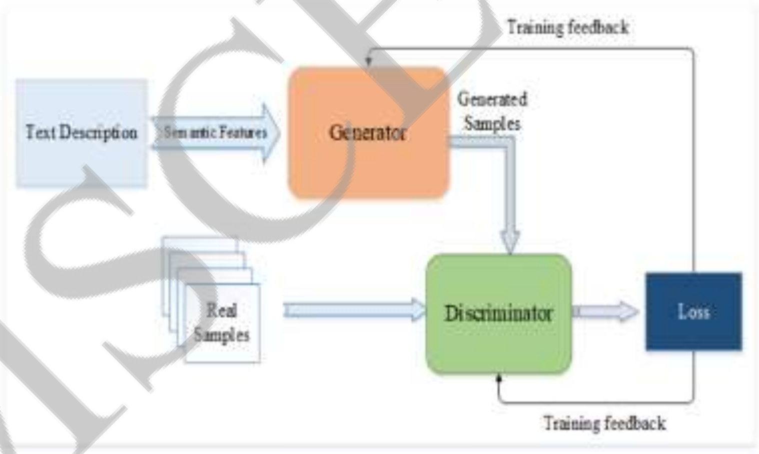
- **Overfitting**:
  Deep recurrent networks are prone to overfitting, especially when dealing with small datasets or overly complex models. With a large number of parameters, deep models have a high capacity to memorize noise or irrelevant patterns in the training data, leading to poor generalization performance on unseen data. Regularization techniques such as dropout and weight decay are commonly used to prevent overfitting.

- **Difficulty in Interpretability:**
  Understanding the internal workings of deep recurrent networks and interpreting their decisions can be challenging. With multiple layers of non-linear transformations, it can be difficult to interpret the learned representations and understand how the network arrives at a particular prediction. This lack of interpretability can be a significant drawback in applications where transparency and interpretability are essential.

Application: Image Generation

- Generating images using recurrent neural networks (RNNs) is an exciting application that leverages the sequential nature of RNNs to produce images pixel by pixel.
- While RNNs are not commonly used for image generation due to their sequential processing nature and the high dimensionality of image data, they can still be applied for certain types of image generation tasks.
- RNN-based approaches can still be useful in scenarios where sequential processing or conditioning on external information is desirable.

Architecture diagram which can generate images from text descriptions:



- Semantic information from the textual description was used as input in the generator model, which converts characteristic information to pixels and generates the images.
- This generated image was used as input in the discriminator along with real/wrong textual descriptions and real sample images from the dataset.
- A sequence of distinct (picture and text) pairings are then provided as input to the model to meet the goals of the discriminator: input pairs of real images and real textual descriptions, wrong images and mismatched textual descriptions, and generated images and real textual descriptions.
- The real photo and real text combinations are provided so that the model can determine if a particular image and text combination align. An incorrect picture and real text description indicates that the image does not match the caption.

- The discriminator is trained to identify real and generated images. At the start of training, the discriminator was good at classification of real/wrong images. Loss was calculated to improve the weight and to provide training feedback to the generator and discriminator model.
- As soon as the training proceeded, the generator produced more realistic images and it fooled the discriminator when distinguishing between real and generated images.

Here's how it can be done:

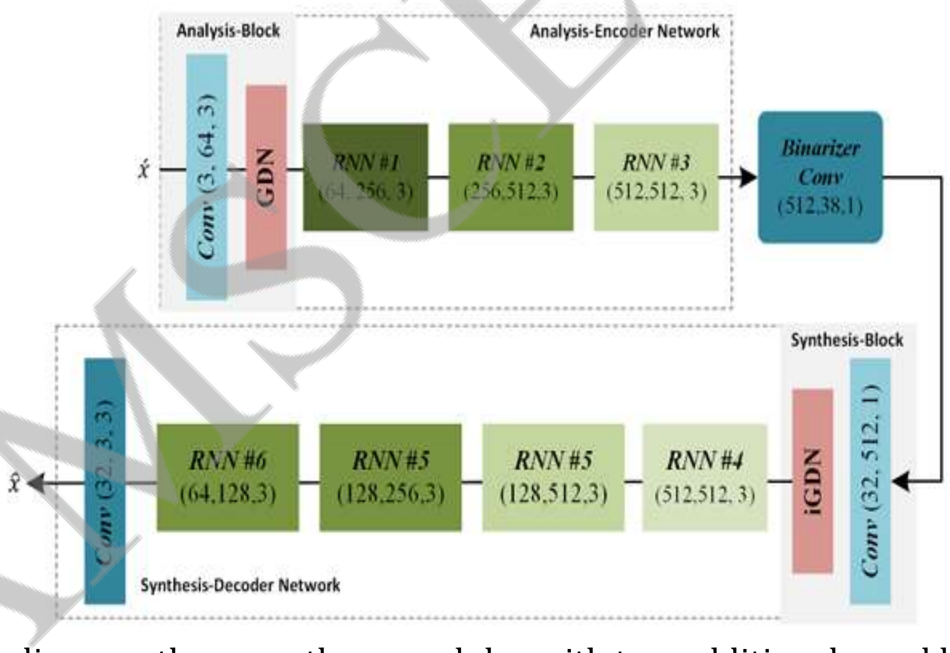
- **Text-to-Image Generation:**
  One common approach to image generation using RNNs is to generate images conditioned on textual descriptions. In this setup, an RNN, such as a Long Short-Term Memory (LSTM) network, is used to process the input text, encoding the semantic information into a fixed-length vector representation. This vector is then used as a conditioning input to another network, typically a Generative Adversarial Network (GAN) or a Variational Autoencoder (VAE), which generates the corresponding image.
- **Sequence-to-Sequence Generation:**
  Another approach is to directly generate images pixel by pixel using autoregressive models. In this setup, an RNN is trained to predict the next pixel in the image sequence given the previous pixels. This process is repeated iteratively until the entire image is generated. Variants of RNNs, such as PixelRNN and PixelCNN, have been proposed for this task, where the model predicts the color value of each pixel conditioned on the previously generated pixels.
- **Conditional Image Generation**:
  RNNs can also be used for conditional image generation, where the generation process is conditioned on some input information. For example, the input could be a low-resolution image, a sketch, or a set of object labels. The RNN processes this input and generates the corresponding high-resolution image or completes the missing parts of the input image.
- **Data Augmentation:**
  RNNs can be used to generate synthetic images for data augmentation purposes. By training an RNN to generate realistic images similar to the training data distribution, additional training samples can be generated to increase the diversity of the dataset and improve the generalization performance of image classification or object detection models.
- **Artistic Style Transfer:**
  RNNs can be used for artistic style transfer, where the style of one image is transferred to the content of another image. In this setup, the RNN is trained to generate an image that matches the content of one image while

incorporating the style features learned from another image. This process typically involves optimizing a loss function that balances content preservation and style transfer.

Application: Image Compression

- Image compression is a method to remove spatial redundancy between adjacent pixels and reconstruct a high-quality image.
- In the past few years, deep learning has gained huge attention from the research community and produced promising image reconstruction results.
- Therefore, recent methods focused on developing deeper and more complex networks, which significantly increased network complexity
- Using recurrent neural networks (RNNs) for image compression is an innovative application that leverages the sequential processing capability of RNNs to effectively encode and compress image data.

Architecture Diagram of image compression framework based on Recurrent Neural Network (RNN)



In above diagram, there are three modules with two additional novel blocks in the end-to-end framework, i.e., encoder network, analysis block, binarizer, decoder network, and synthesis block. Image patches are directly given to the analysis block as an input that generates latent features using the proposed analysis encoder block. The entire framework architecture is presented in architecture diagram.

The single iteration of the end-to-end framework is represented in below Equation.

$$b_t = \mathrm{Bin}\left(\mathrm{Enc}_t\left(r_t - 1\right)\right)$$
$$\hat{x}_t = \mathrm{Dec}_t\left(\mathrm{bin}_t\right) + \gamma \hat{x}_{t-1}$$

$$r_t = x - \hat{x}_t, \quad r_0 = x, \quad \hat{x}_0 = 0$$

The training process of image compression network is optimized by adopting the loss at each iteration based on actual weighted and predicted value.

$$L_1 = \sum_{i=1}^{n} \left| y_{\text{true}} - y_{\text{predicted}} \right|$$

The overall loss at each iteration of the variable-rate framework is:

$$L_1 = \beta \sum_t |r_t|$$

Here's how RNNs can be applied for image compression:

- **Sequence-to-Sequence Compression:**
  In this approach, the input image is divided into a sequence of patches or blocks. Each block is then sequentially processed by an RNN, such as a Long Short-Term Memory (LSTM) network or a Gated Recurrent Unit (GRU). The RNN compresses the information in each block into a fixed-length vector representation, capturing the essential features of the image content.
- **Hierarchical Compression:**
  Another approach involves using a hierarchical RNN architecture for compression. In this setup, multiple layers of RNNs are stacked together, with each layer processing increasingly abstract representations of the image. The lower layers capture fine-grained details, while the higher layers capture more global structures and patterns. This hierarchical representation enables efficient compression of images with varying levels of detail.
- **Conditional Compression:**
  RNNs can be conditioned on contextual information to improve compression performance. For example, the compression process can be conditioned on the image content, image resolution, or specific compression requirements (e.g., target compression ratio). By incorporating additional information into the compression model, RNNs can adapt their encoding strategy to better preserve important features of the input image.

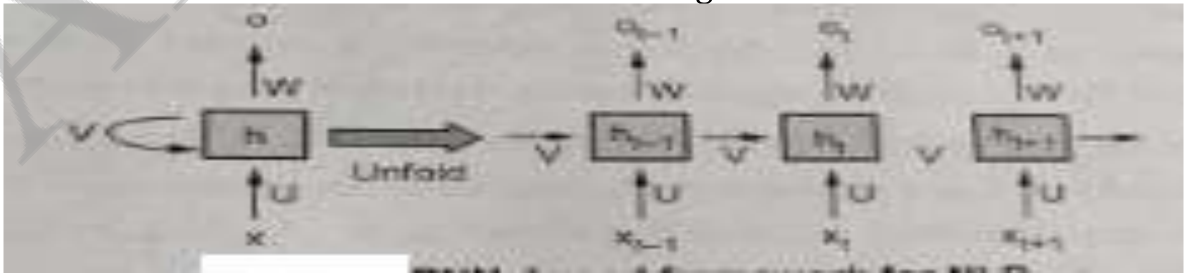
- **Lossy Compression:**
  RNN-based compression models can be trained to perform lossy compression, where some information in the input image is discarded to achieve higher compression ratios. The RNN learns to prioritize important features while discarding less critical information, resulting in compact representations of the input images. Techniques such as quantization and entropy coding can be combined with RNN-based compression to further improve compression efficiency.

- **Learned Compression Algorithms:**
  Instead of handcrafting compression algorithms, RNNs can be trained to learn effective compression strategies directly from data. By optimizing compression performance using techniques such as autoencoders or reinforcement learning, RNN-based compression models can adapt to the statistical properties of different types of images and achieve better compression ratios.

Application: Natural Language Processing

- Natural Language Processing (NLP) using recurrent neural networks (RNNs) is a prominent area of research and application.
- RNNs, with their ability to model sequential data, are well-suited for various NLP tasks that involve understanding and generating natural language.
- RNNs play a vital role in various NLP tasks by effectively modeling the sequential nature of natural language and capturing the contextual dependencies in text data.
- Their versatility and ability to handle sequential data make them a powerful tool for understanding, generating, and processing natural language in a wide range of applications.
- RNN are effective for sequential data processing. In RNN computation is recursively applied to each instance of input sequence from previous computed results. Recurrent unit is sequentially fed with the sequences represented by fixed size vector of tokens.

RNN based framework for NLP is shown in Figure below:



The advantage of RNN is that it can memorize the results of previous computation and utilize that information in current computation.
So, it is possible to model context dependencies in inputs of arbitrary length

with RNN and proper composition of input can be created.
Mainly RNNs are used in different NLP tasks like,

- Natural language generation (e.g. image captioning, machine translation, visual question answering)
- Word - level classification (e.g. Named Entity recognition (NER))
- Language modelling
- Semantic matching
- Sentence-level classification (e.g., sentiment polarity)

Here are some key applications of RNNs in NLP:

✓ **Sequence Modelling:**
RNNs excel at sequence modelling tasks, such as language modelling and text generation. They can be trained to predict the next word in a sentence given the previous words, capturing the sequential dependencies in the language. Language models based on RNNs have been used for tasks like speech recognition, machine translation, and autocomplete suggestions.

✓ **Machine Translation:**
RNNs, particularly the sequence-to-sequence (seq2seq) architecture, have been widely used for machine translation tasks. In this setup, an RNN encoder processes the input sentence in the source language, and another RNN decoder generates the corresponding translation in the target language. This approach has been extended with attention mechanisms to handle longer sentences and improve translation quality.

✓ **Sentiment Analysis:**
RNNs are effective for sentiment analysis tasks, where the goal is to determine the sentiment or opinion expressed in a piece of text. By processing the text sequentially and capturing the contextual information, RNNs can classify text into different sentiment categories (e.g., positive, negative, neutral). They have been used for sentiment analysis in social media posts, customer reviews, and news articles.

✓ **Named Entity Recognition (NER):**
RNNs have been applied to named entity recognition tasks, where the goal is to identify and classify entities (e.g., persons, organizations, locations) mentioned in text. By modelling the sequential context of the text, RNNs can learn to recognize and classify entities based on their surrounding words and phrases. This is useful in applications like information extraction and text summarization.

✓ **Part-of-Speech Tagging:**
RNNs can be used for part-of-speech (POS) tagging, where each word in a sentence is assigned a grammatical category (e.g., noun, verb, adjective). By considering the sequential context of the words, RNNs can

learn to predict the POS tags more accurately, even for ambiguous cases. POS tagging is an essential component in many NLP pipelines and applications.

- ✓ **Text Classification:**
  RNNs are commonly used for text classification tasks, such as document categorization, topic modelling, and spam detection. By processing the text sequentially and capturing the semantic information, RNNs can learn to classify documents or sentences into different categories based on their content. They have been used in various domains, including news categorization, customer support, and email filtering.
- ✓ **Dialogue Systems:**
  RNNs have been employed in dialogue systems, also known as chatbots or conversational agents, to generate responses in natural language. By modelling the sequential interaction between users and the system, RNNs can generate contextually relevant and coherent responses to user queries or prompts. Dialogue systems based on RNNs have been used in virtual assistants, customer service bots, and language learning applications.

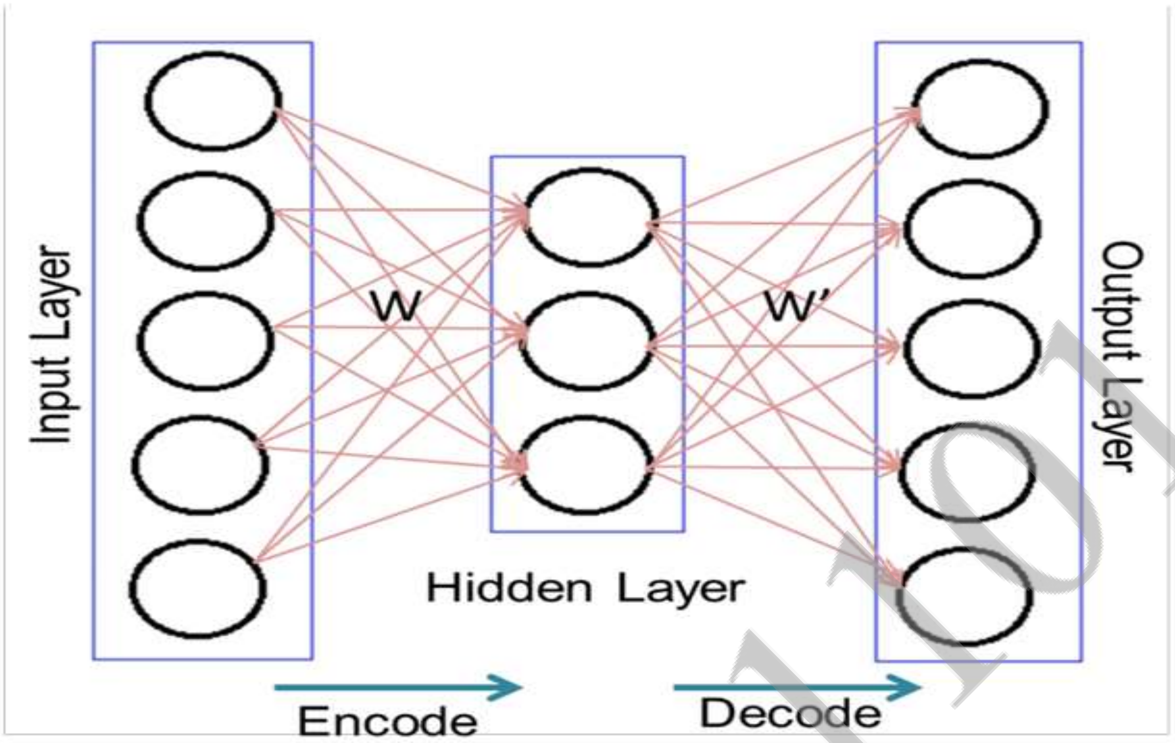
Complete Auto Encoder

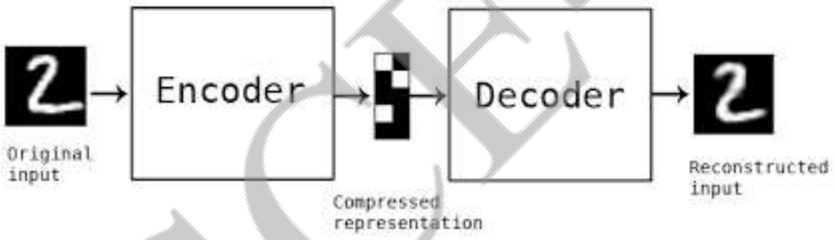
- ✓ An autoencoder is a type of artificial neural network used for unsupervised learning of efficient data representations.
- ✓ Autoencoders emerge as a fascinating subset of neural networks, offering a unique approach to unsupervised learning.
- ✓ Autoencoders are an adaptable and strong class of architectures for the dynamic field of deep learning, where neural networks develop constantly to identify complicated patterns and representations.
- ✓ With their ability to learn effective representations of data, these unsupervised learning models have received considerable attention and are useful in a wide variety of areas, from image processing to anomaly detection.

It consists of two main components:

- ✓ **An encoder:** The encoder compresses the input data into a latent representation.
- ✓ **A decoder:** The decoder reconstructs the original input from the latent representation.

Architecture of Complete Auto Encoder





- 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 can 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.
- Autoencoders can come in various architectures, each serving different purposes and having different properties.

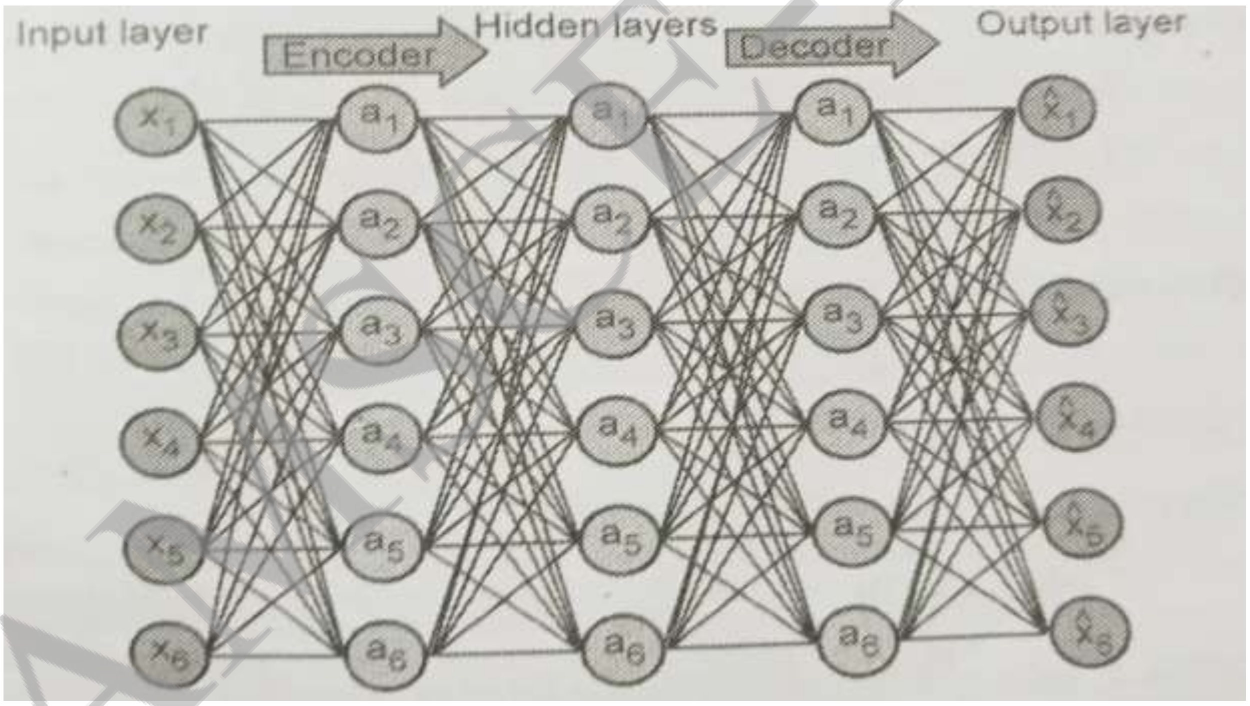
Here are some types of complete autoencoders:

- **Vanilla Autoencoder**:
  A vanilla autoencoder consists of an encoder and a decoder where both are fully connected neural networks. It aims to learn a compressed representation of the input data without any specific constraints on the learned representations.

- **Sparse Autoencoder:**
  In a sparse autoencoder, additional constraints are imposed on the learned representations to encourage sparsity. This can be achieved by adding a sparsity penalty term to the loss function, such as L1 regularization or the Kullback-Leibler (KL) divergence.
- **Denoising Autoencoder:**
  Denoising autoencoders are trained to reconstruct clean data from corrupted inputs. During training, noise is added to the input data, and the model is trained to reconstruct the original, noise-free data. This helps the model learn more robust and informative representations.
- **Variational Autoencoder (VAE):**
  VAEs are probabilistic autoencoders that learn a latent variable model of the data. They aim to capture the underlying probability distribution of the input data in the latent space and generate new samples by sampling from this distribution. VAEs consist of an encoder that outputs the parameters of a probability distribution (e.g., mean and variance) and a decoder that samples from this distribution to generate reconstructions.
- **Contractive Autoencoder:**
  Contractive autoencoders are trained to learn representations that are robust to small perturbations in the input data. They achieve this by adding a penalty term to the loss function that penalizes the Frobenius norm of the Jacobian matrix of the encoder with respect to the input data.
- **Adversarial Autoencoder (AAE):**
  AAEs combine autoencoders with adversarial training techniques. They consist of an encoder-decoder pair trained to reconstruct the input data, along with a discriminator network that tries to distinguish between the latent representations learned by the encoder and samples from a prior distribution.
- **Convolutional Autoencoder:**
  Convolutional autoencoders use convolutional layers instead of fully connected layers in both the encoder and decoder. They are particularly well-suited for image data and can capture spatial dependencies more effectively compared to vanilla autoencoders.
- **Recurrent Autoencoder:**
  Recurrent autoencoders utilize recurrent neural networks (RNNs) in either the encoder, decoder, or both. They are useful for sequential data, such as time series or natural language sequences, and can capture temporal dependencies in the input data.

Regularized autoencoders

- Regularized autoencoders are a type of autoencoder that incorporates regularization techniques to improve the quality of learned representations and prevent overfitting.
- These techniques impose additional constraints on the autoencoder's training process, encouraging it to learn more robust and generalizable representations of the input data.
- Regularization helps prevent the autoencoder from memorizing the training data and capturing noise, resulting in better performance on unseen data.
- Regularized autoencoders are widely used in various applications, including dimensionality reduction, feature learning, data denoising, and anomaly detection.
- By incorporating regularization techniques into the training process, regularized autoencoders can learn more informative and generalizable representations of the input data, leading to better performance on downstream tasks.


Input layer
Encoder
Hidden layers
Decoder
Output layer
x_1 x_2 x_3 x_4 x_5 x_6
a_1 a_2 a_3 a_4 a_5 a_6
$\hat{x}_1$ $\hat{x}_2$ $\hat{x}_3$ $\hat{x}_4$ $\hat{x}_5$ $\hat{x}_6$

Structure of Regularized Autoencoders

Let's dive into the structural nuances that differentiate regularized autoencoders from their traditional counterparts.

Neuronal Arrangement:

The arrangement remains like traditional autoencoders, with an encoder and a decoder. The deviation lies in the incorporation of regularization methods within the layers.

Activation Functions:

Regularized autoencoders may employ specific activation functions tailored for regularization, contributing to a more balanced learning process.

Incorporating Regularization Methods:
Regularization methods, such as dropout or L1/L2 regularization, are integrated into the architecture to curb overfitting.

Some common regularization techniques used in regularized autoencoders include:

- **L1 and L2 Regularization:**
  L1 and L2 regularization penalize the magnitude of the weights in the autoencoder's neural network. By adding a regularization term to the loss function proportional to either the L1 or L2 norm of the weights, these techniques encourage sparsity (in the case of L1 regularization) or small weights (in the case of L2 regularization), helping prevent overfitting.

- **Dropout:**
  Dropout is a regularization technique that randomly sets a fraction of the input units to zero during each training iteration. This helps prevent the autoencoder's neural network from relying too heavily on any individual input features, forcing it to learn more robust representations.

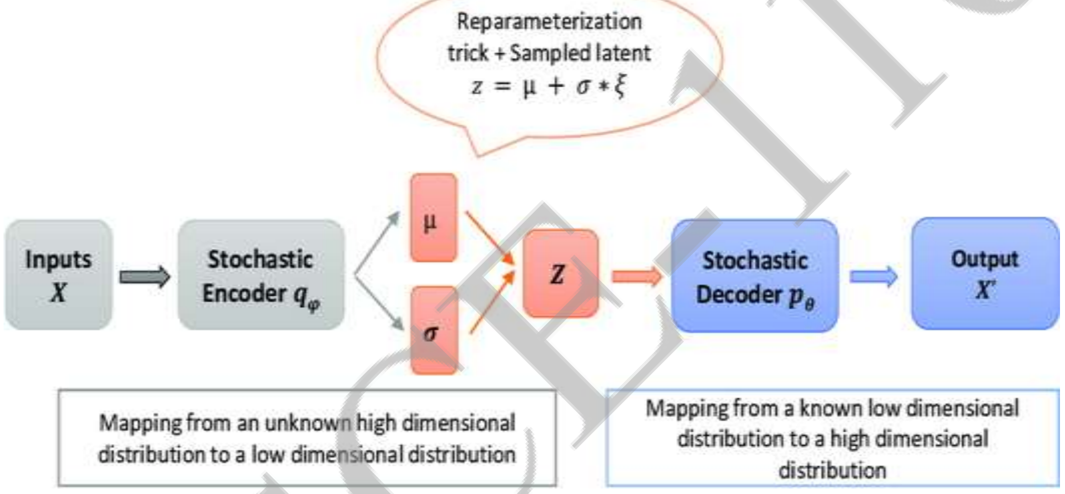
- **Batch Normalization:**
  Batch normalization normalizes the activations of each layer in the autoencoder's neural network, helping stabilize and accelerate the training process. By reducing internal covariate shift, batch normalization acts as a regularizer, making the autoencoder more resistant to overfitting.

- **Noise Injection:**
  Noise injection involves adding noise to the input data or the activations of the autoencoder's hidden layers during training. This helps prevent the autoencoder from memorizing the training data and encourages it to learn more generalizable representations.

- **Contractive Regularization:**
  Contractive regularization penalizes the Frobenius norm of the Jacobian matrix of the encoder with respect to the input data. This encourages the encoder to learn representations that are invariant to small changes in the input data, making the autoencoder more robust to variations in the input.

Stochastic Encoders and Decoders

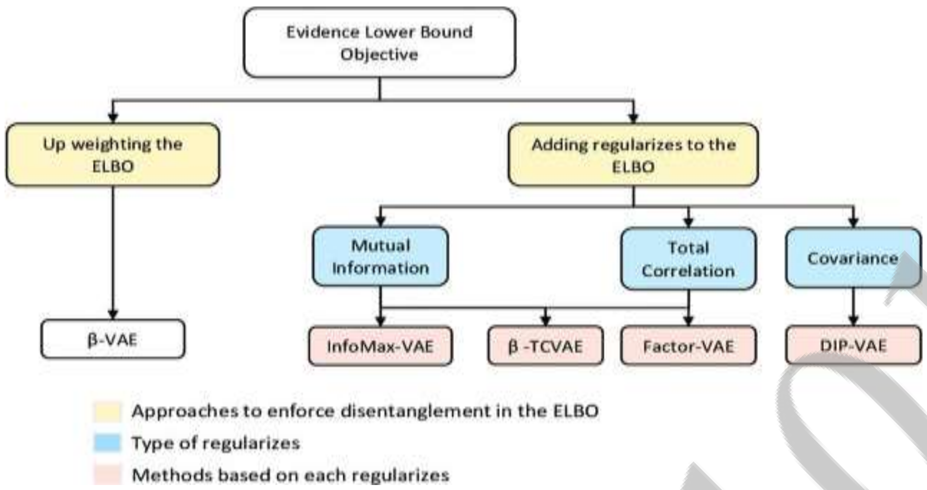
- Stochastic encoders and decoders are components of probabilistic autoencoder models, such as Variational Autoencoders (VAEs).
- These components introduce stochasticity into the encoding and decoding process, enabling the model to learn a probabilistic representation of the input data distribution.
- Stochastic encoders and decoders in VAEs enable various applications, including generative modelling, data synthesis, and unsupervised representation learning.
- They provide a principled framework for learning complex data distributions and generating new samples from these distributions.



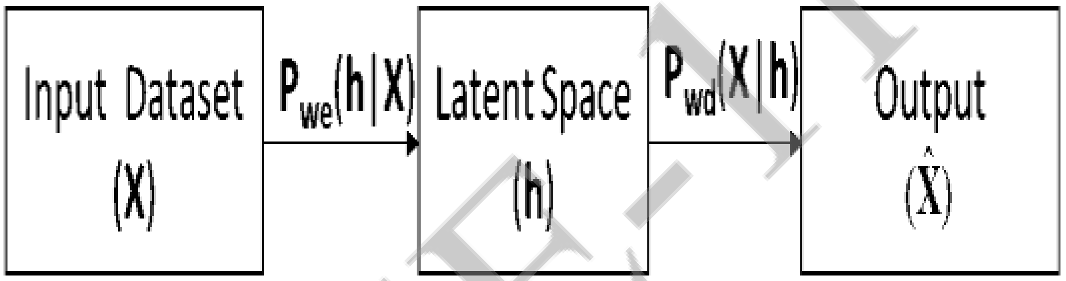
Reparameterization trick + Sampled latent $z = \mu + \sigma * \xi$
Inputs X
Stochastic Encoder q_φ
μ
σ
z
Stochastic Decoder p_θ
Output X'
Mapping from an unknown high dimensional distribution to a low dimensional distribution
Mapping from a known low dimensional distribution to a high dimensional distribution

Stochastic Encoder:
In a VAE, the encoder network outputs the parameters of a probability distribution instead of a deterministic encoding. Instead of directly outputting the latent representation of the input data, the encoder outputs the mean and variance (or other parameters) of a Gaussian distribution that represents the distribution of possible latent variables given the input. The latent variable is then sampled from this distribution to generate a stochastic representation.

Stochastic Decoder:
Similarly, the decoder network in a VAE accepts a sampled latent variable as input instead of a deterministic encoding. This sampled latent variable is generated by sampling from the distribution outputted by the encoder. The decoder then generates the reconstructed output based on this sampled latent variable.


Evidence Lower Bound Objective
Up weighting the ELBO
Adding regularizes to the ELBO
Mutual Information
Total Correlation
Covariance
β-VAE
InfoMax-VAE
β -TCVAE
Factor-VAE
DIP-VAE
Approaches to enforce disentanglement in the ELBO
Type of regularizes
Methods based on each regularizes

Cost Function Calculation



The cost function of VAE is based on log likelihood maximization.
The cost function consists of reconstruction and regularization error terms:

Cost = Reconstruction Error + Regularization Error

Contractive autoencoders

- Contractive autoencoders are a variant of autoencoders that incorporate a regularization term known as contractive regularization.
- The goal of contractive regularization is to encourage the autoencoder's encoder network to learn a more robust and stable representation of the input data by penalizing variations in the input space.
- In a contractive autoencoder, the contractive regularization term is added to the loss function during training. This regularization term penalizes the Frobenius norm of the Jacobian matrix of the encoder's output with respect to the input data.
- Intuitively, this penalizes variations in the input space by encouraging the encoder to learn representations that are insensitive to small changes in the input data.

- Contractive autoencoder simply targets to learn invariant representations to unimportant transformations for the given data.
- CAE surpasses results obtained by regularizing autoencoder using weight decay or by denoising. CAE is a better choice than denoising autoencoder to learn useful feature extraction.

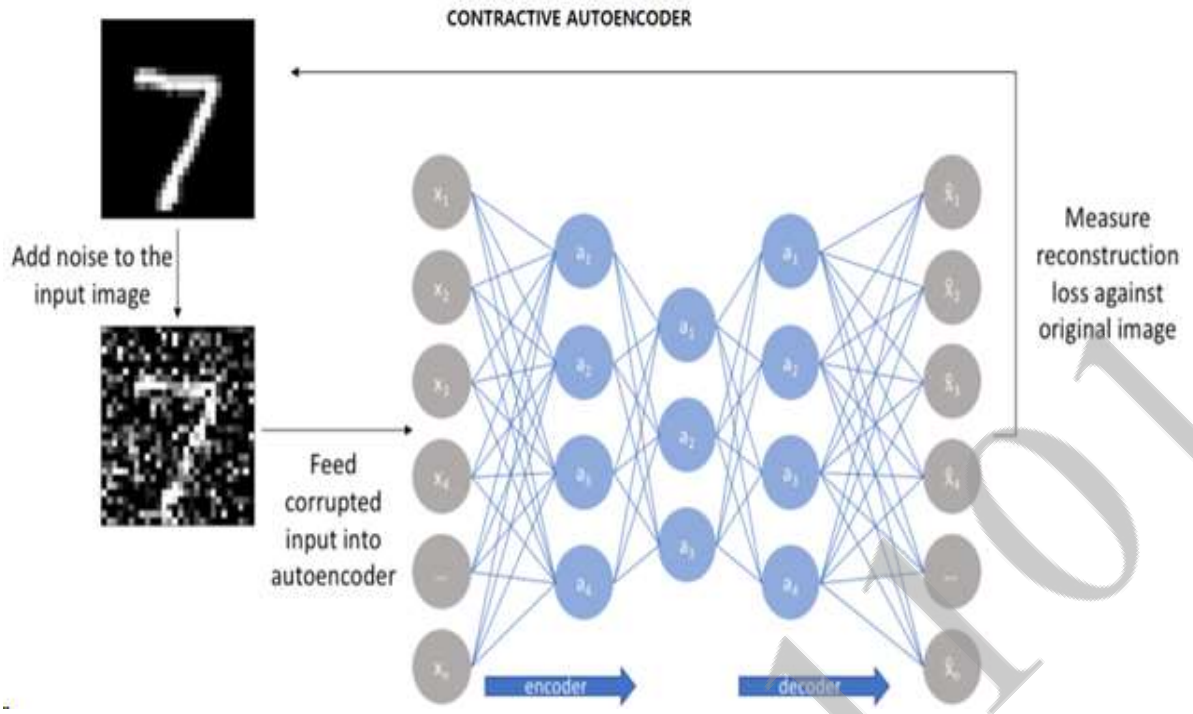
Mathematically, the contractive regularization term can be expressed as follows:

$$\Omega = \lambda \sum_{i=1}^{N} ||\nabla_x h_i(x)||_F^2$$

Where:

- Ω is the contractive regularization term.
- λ is the regularization strength.
- N is the dimensionality of the encoded representation.
- $h_i(x)$ is the ith element of the encoded representation.
- $\nabla_x h_i(x)$ is the gradient of the ith encoded representation with respect to the input x.
- $||\cdot||_F$ denotes the Frobenius norm.

- During training, the contractive autoencoder is optimized to minimize the reconstruction error (e.g., mean squared error) while simultaneously minimizing the contractive regularization term.
- This encourages the encoder to learn representations that capture the underlying structure of the data while being robust to small perturbations in the input space.
- Contractive autoencoders have been applied in various domains, including dimensionality reduction, feature learning, and data denoising.
- They are particularly useful in scenarios where the input data is noisy or contains small variations, as they encourage the autoencoder to learn stable and invariant representations of the data.


CONTRACTIVE AUTOENCODER
Add noise to the input image
Feed corrupted input into autoencoder
Measure reconstruction loss against original image
encoder
decoder

The benefits and applications of contractive autoencoders include:

- **Robustness to Noise:** Contractive regularization encourages the encoder to learn representations that are robust to small variations and noise in the input data. This makes contractive autoencoders suitable for tasks involving noisy or corrupted data, such as denoising autoencoding
- **Improved Generalization:** By penalizing variations in the input space, contractive regularization helps prevent overfitting and improves the generalization performance of the autoencoder. This allows the model to learn more generalizable representations of the data that can be applied to unseen examples.
- **Feature Learning:** Contractive autoencoders can learn informative and discriminative features from the input data by capturing the underlying structure of the data distribution. These learned features can be used for downstream tasks such as classification, clustering, or anomaly detection.
- **Dimensionality Reduction:** The compact and stable representations learned by contractive autoencoders can be used for dimensionality reduction tasks. By projecting high-dimensional data into a lower-dimensional space while preserving important information, contractive autoencoders facilitate visualization, data compression, and efficient storage.
- **Unsupervised Learning:** Contractive autoencoders belong to the class of unsupervised learning algorithms, as they do not require labelled data during training. This makes them suitable for tasks where labelled data is scarce or expensive to obtain, allowing for the extraction of useful information from large amounts of unlabelled data.