studocu

DL Notes 1-5 - deep learning

Computer Science and Engineering (Anna University)

Scan to open on Studocu

Studocu is not sponsored or endorsed by any college or university
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

IF 4071 DEEP LEARNING

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

IF4071	DEEP LEARNING	L T P C 3 0 2 4

COURSE OBJECTIVES:

- Develop and Train Deep Neural Networks.
- Develop a CNN, R-CNN, Fast R-CNN, Faster-R-CNN, Mask-RCNN for detection and recognition.
- Build and train RNNs, work with NLP and Word Embeddings.
- The internal structure of LSTM and GRU and the differences between them.
- The Auto Encoders for Image Processing.

UNIT I DEEP LEARNING CONCEPTS

Fundamentals about Deep Learning. Perception Learning Algorithms. Probabilistic modelling. Early Neural Networks. How Deep Learning different from Machine Learning. Scalars. Vectors. Matrixes, Higher Dimensional Tensors. Manipulating Tensors. Vector Data. Time Series Data. Image Data. Video Data.

UNIT II NEURAL NETWORKS

About Neural Network. Building Blocks of Neural Network. Optimizers. Activation Functions. Loss Functions. Data Pre-processing for neural networks, Feature Engineering. Overfitting and Underfitting. Hyperparameters.

UNIT III CONVOLUTIONAL NEURAL NETWORK

About CNN. Linear Time Invariant. Image Processing Filtering. Building a convolutional neural network. Input Layers, Convolution Layers. Pooling Layers. Dense Layers. Backpropagation Through the Convolutional Layer. Filters and Feature Maps. Backpropagation Through the Pooling Layers. Dropout Layers and Regularization. Batch Normalization. Various Activation Functions. Various Optimizers. LeNet, AlexNet, VGG16, ResNet. Transfer Learning with Image Data. Transfer Learning using Inception Oxford VGG Model, Google Inception Model, Microsoft ResNet Model. R-CNN, Fast R-CNN, Faster R-CNN, Mask-RCNN, YOLO.

UNIT VI NATURAL LANGUAGE PROCESSING USING RNN

About NLP & its Toolkits. Language Modeling . Vector Space Model (VSM). Continuous Bag of Words (CBOW). Skip-Gram Model for Word Embedding. Part of Speech (PoS) Global Co-occurrence Statistics–based Word Vectors. Transfer Learning. Word2Vec. Global Vectors for Word Representation GloVe. Backpropagation Through Time. Bidirectional RNNs (BRNN) . Long Short Term Memory (LSTM). Bi-directional LSTM. Sequence-to-Sequence Models (Seq2Seq). Gated recurrent unit GRU.

UNIT V DEEP REINFORCEMENT & UNSUPERVISED LEARNING

About Deep Reinforcement Learning. Q-Learning. Deep Q-Network (DQN). Policy Gradient Methods. Actor-Critic Algorithm. About Autoencoding. Convolutional Auto Encoding. Variational Auto Encoding. Generative Adversarial Networks. Autoencoders for Feature Extraction. Auto Encoders for Classification. Denoising Autoencoders. Sparse Autoencoders.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

UNIT - I DEEP LEARNING CONCEPTS (Page No. 2-33)

1.Fundamentals about Deep Learning Pg.6

- AI, ML & DL Relation
- AI - Artificial Intelligence
  - What is AI?
  - Definition of AI
  - Subfields
  - History
- ML - Machine Learning
  - Origin of ML
  - ML - New Programming Paradigm
  - ML Process
  - ML vs Statistics
  - Definition & Goal of ML
  - 3 Things to do ML
  - Learning Representations from Data
    - Central Problem in ML & DL
    - What's a representation?
    - Example - Black points & White points
    - Learning in the context of ML
- DL - Deep Learning
  - What is DL?
  - "Deep" in Deep Learning
    - Depth of the model
    - No. of layers
    - How layers learn?
    - Working of Neural networks with Example - Digit Classification
  - How this learning happens?
    - Weights AKA Parameters
    - Loss function AKA Objective function
    - Optimizer
    - Trained Network
  - DL achieved so far?

2.Perception Learning Algorithms Pg.13

- Simple Model of Neural Networks – The Perceptron
- History of Perceptron
- Types of Classification Problems

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

- Linearly Separable
- Non-Linearly Separable

- Types of Perceptron
  - Single Layer Perceptron
  - Multi-Layer Perceptron
    - Forward Stage
    - Backward Stage
- Primary Components of a Perceptron
  - Neurons
  - Synapse
  - Input
  - Weight
  - Bias
  - Activation / Step Function
    i. Sign Function
    ii. Step Function
    iii. Sigmoid Function
  - Weighted Summation
- The Perceptron Learning Algorithm
  - Perceptron Learning Steps
  - Perceptron Learning Rule
    - Error in Perceptron
    - Back Propagation
- Example – Illustrating how a Perceptron works
- Which Applications use Perceptrons?
  - Object Recognition in Images
  - Customer Sentiment Analysis
  - Spam Detection
- Advantages
- Limitations
- Summary

3.Probabilistic Modelling Pg. 21

- What is Probabilistic Modelling?
- Importance of Probabilistic ML Models
- Examples of Probabilistic Models
  - Straight Line Modelling
  - Weather and Traffic
  - Naive Bayes Algorithm

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

- Real-world examples of Data Tensors
- Vector Data

12.Time Series Data (AKA) Sequence Data Pg. 32

13. Image Data Pg. 33

14. Video Data Pg. 33

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

1.Fundamentals about Deep Learning

AI, ML & DL Relation

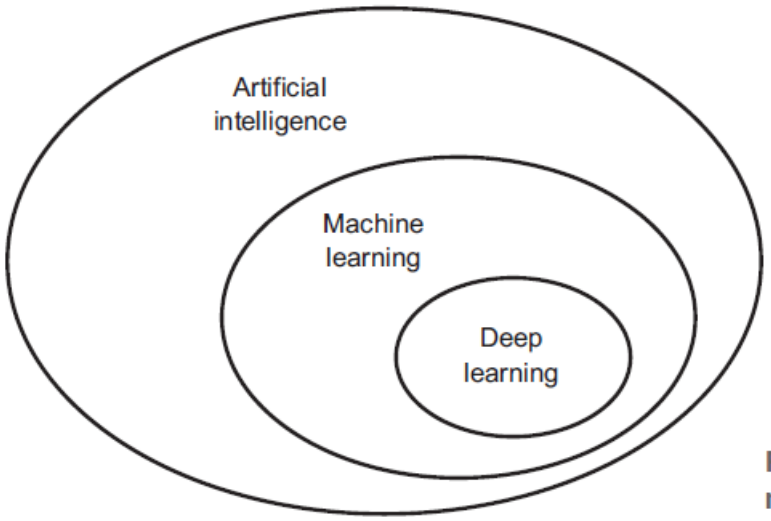


Figure 1.1 Artificial intelligence, machine learning, and deep learning

The field of AI is broad and has been around for a long time. Deep learning is a subset of the field of machine learning, which is a subfield of AI.

AI - Artificial Intelligence

What is AI?

Artificial intelligence was born in the 1950s, when a handful of pioneers from the nascent field of computer science started asking whether computers could be made to "think"—a question whose ramifications we're still exploring today.

Definition of AI

A concise definition of the field would be as follows: the effort to automate intellectual tasks normally performed by humans.

Subfields

AI is a general field that encompasses machine learning and deep learning, but that also includes many more approaches that don't involve any learning.

History

For a fairly long time, many experts believed that human-level artificial intelligence could be achieved by having programmers handcraft a sufficiently large set of explicit rules for manipulating knowledge. This approach is known as symbolic AI, and it was the dominant paradigm in AI from the 1950s to the late 1980s. It reached its peak popularity during the expert systems boom of the 1980s.

Although symbolic AI proved suitable to solve well-defined, logical problems, such as playing chess, it turned out to be intractable to figure out explicit rules for solving more complex, fuzzy

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

problems, such as image classification, speech recognition, and language translation. A new approach arose to take symbolic AI's place: machine learning.

ML - Machine Learning

Origin of ML

Machine learning arises from this question: could a computer go beyond "what we know how to order it to perform" and learn on its own how to perform a specified task? Could a computer surprise us? Rather than programmers crafting data-processing rules by hand, could a computer automatically learn these rules by looking at data?

ML - New Programming Paradigm

In classical programming, the paradigm of symbolic AI, humans input rules (a program) and data to be processed according to these rules, and outcome answers (see figure 1.2). With machine learning, humans input data as well as the answers expected from the data, and outcome the rules. These rules can then be applied to new data to produce original answers.

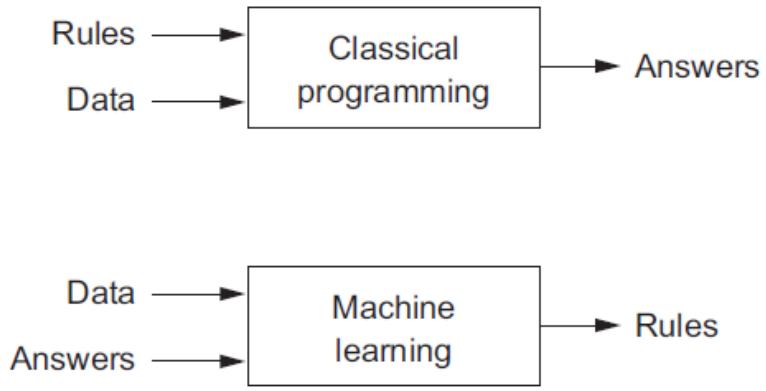


Figure 1.2 Machine learning: a new programming paradigm

ML Process

A machine-learning system is trained rather than explicitly programmed. It's presented with many examples relevant to a task, and it finds statistical structure in these examples that eventually allows the system to come up with rules for automating the task. For instance, if you wished to automate the task of tagging your vacation pictures, you could present a machine-learning system with many examples of pictures already tagged by humans, and the system would learn statistical rules for associating specific pictures to specific tags.

ML vs Statistics

Although machine learning only started to flourish in the 1990s, it has quickly become the most popular and most successful subfield of AI, a trend driven by the availability of faster hardware and larger datasets. Machine learning is tightly related to mathematical statistics, but it differs from statistics in several important ways. Unlike statistics, machine learning tends to deal with large, complex datasets (such as a dataset of millions of images, each consisting of tens of thousands of pixels) for which classical statistical analysis such as Bayesian analysis would be impractical.

Definition & Goal of ML

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Machine learning discovers rules to execute a data-processing task, given examples of what's expected. Machine learning is a subfield of artificial intelligence, which is broadly defined as the capability of a machine to imitate intelligent human behaviour. It focuses on analysing and interpreting patterns and structures in data to enable learning, reasoning, and decision making outside of human interaction.

The goal of machine learning is often — though not always — to train a model on historical, labelled data (i.e., data for which the outcome is known) in order to predict the value of some quantity on the basis of a new data item for which the target value or classification is unknown.

3 Things to do ML

To do machine learning, we need three things:

i. *Input data points*—For instance, if the task is speech recognition, these data points could be sound files of people speaking. If the task is image tagging, they could be pictures.
ii. *Examples of the expected output*—In a speech-recognition task, these could be human-generated transcripts of sound files. In an image task, expected outputs could be tags such as "dog," "cat," and so on.
iii. *A way to measure whether the algorithm is doing a good job*—This is necessary in order to determine the distance between the algorithm's current output and its expected output. The measurement is used as a feedback signal to adjust the way the algorithm works. This adjustment step is what we call *learning*.

Learning Representations from Data

A machine-learning model transforms its input data into meaningful outputs, a process that is "learned" from exposure to known examples of inputs and outputs.

Central Problem in ML & DL

The central problem in machine learning and deep learning is to meaningfully transform data: in other words, to learn useful representations of the input data at hand—representations that get us closer to the expected output.

What's a representation?

It's a different way to look at data—to represent or encode data.

For instance, a colour image can be encoded in the RGB format (red-green-blue) or in the HSV format (hue-saturation-value): these are two different representations of the same data. Some tasks that may be difficult with one representation can become easy with another. For example, the task "select all red pixels in the image" is simpler in the RG format, whereas "make the image less saturated" is simpler in the HSV format.

Machine-learning models are all about finding appropriate representations for their input data—transformations of the data that make it more amenable to the task at hand, such as a classification task.

Example - Black points & White points (Classification Problem)

Consider an x-axis, a y-axis, and some points represented by their coordinates in the (x, y) system, as shown in figure 1.3.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

As you can see, we have a few white points and a few black points. Let's say we want to develop an algorithm that can take the coordinates (x, y) of a point and output whether that point is likely to be black or to be white. In this case,

i. The inputs are the coordinates of our points.

ii. The expected outputs are the colours of our points.

iii. A way to measure whether our algorithm is doing a good job could be, for instance, the percentage of points that are being correctly classified.

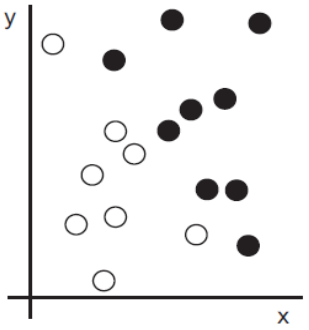


Figure 1.3
Some sample data

What we need here is a new representation of our data that cleanly separates the white points from the black points. One transformation we could use, among many other possibilities, would be a coordinate change, illustrated in figure 1.4.

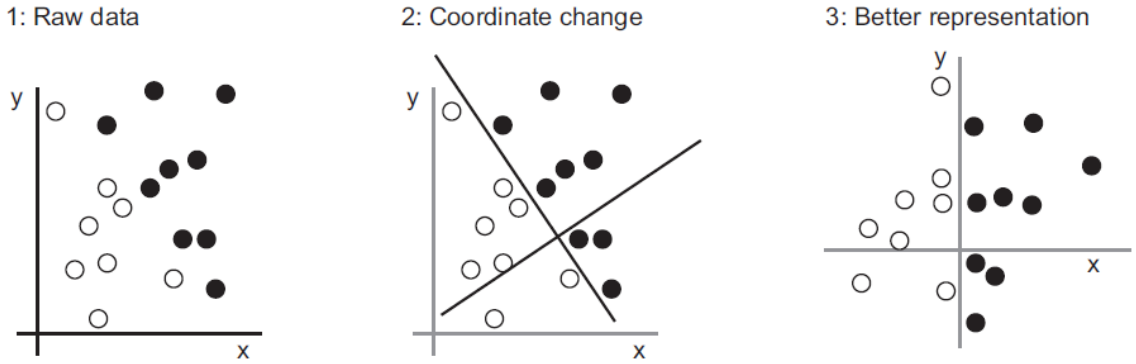


Figure 1.4 Coordinate change

In this new coordinate system, the coordinates of our points can be said to be a new representation of our data. And it's a good one! With this representation, the black/white classification problem can be expressed as a simple rule: "Black points are such that x > 0," or "White points are such that x < 0." This new representation basically solves the classification problem.

In this case, we defined the coordinate change by hand. But if instead we tried systematically searching for different possible coordinate changes, and used as feedback the percentage of points being correctly classified, then we would be doing machine learning.

Learning in the context of ML

Learning, in the context of machine learning, describes an automatic search process for better representations. All machine-learning algorithms consist of automatically finding such transformations that turn data into more-useful representations for a given task. These operations can be coordinate changes, as you just saw, or linear projections (which may destroy information), translations, nonlinear operations (such as "select all points such that x > 0"), and so on.

Machine-learning algorithms aren't usually creative in finding these transformations; they're merely searching through a predefined set of operations, called a *hypothesis space.*

So that's what machine learning is, technically: searching for useful representations of some input data, within a predefined space of possibilities, using guidance from a feedback signal. This simple idea allows for solving a remarkably broad range of intellectual tasks, from speech recognition to autonomous car driving.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

DL - Deep Learning

What is DL?

Deep learning is a specific subfield of machine learning: a new take on learning representations from data that puts an emphasis on learning successive *layers* of increasingly meaningful representations. Other appropriate names for the field could have been layered representations learning and hierarchical representations learning.

"Deep" in Deep Learning

The deep in deep learning isn't a reference to any kind of deeper understanding achieved by the approach; rather, it stands for this idea of successive layers of representations.

Depth of the model - How many layers contribute to a model of the data is called the depth of the model.

No. of layers

Modern deep learning often involves tens or even hundreds of successive layers of representations— and they're all learned automatically from exposure to training data. Meanwhile, other approaches to machine learning tend to focus on learning only one or two layers of representations of the data; hence, they're sometimes called *shallow learning*.

How layers learn?

In deep learning, these layered representations are (almost always) learned via models called *neural networks*, structured in literal layers stacked on top of each other. The term neural network is a reference to neurobiology, but although some of the central concepts in deep learning were developed in part by drawing inspiration from our understanding of the brain, deep-learning models are not models of the brain. There's no evidence that the brain implements anything like the learning mechanisms used in modern deep-learning models. For our purposes, deep learning is a mathematical framework for learning representations from data.

Working of Neural networks with Example - Digit Classification

What do the representations learned by a deep-learning algorithm look like? Let's examine how a network several layers deep (see figure 1.5) transform an image of a digit in order to recognize what digit it is.

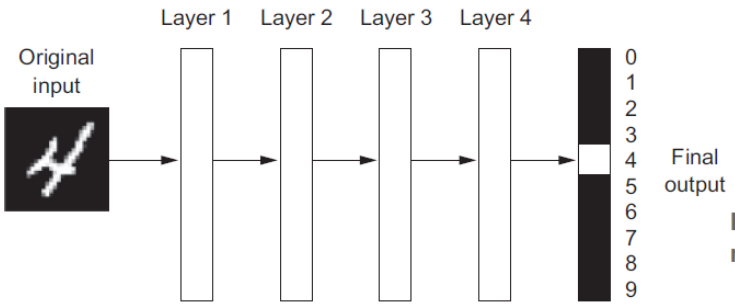


Figure 1.5 A deep neural network for digit classification

As you can see in figure 1.6, the network transforms the digit image into representations that are increasingly different from the original image and increasingly informative about the final result.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

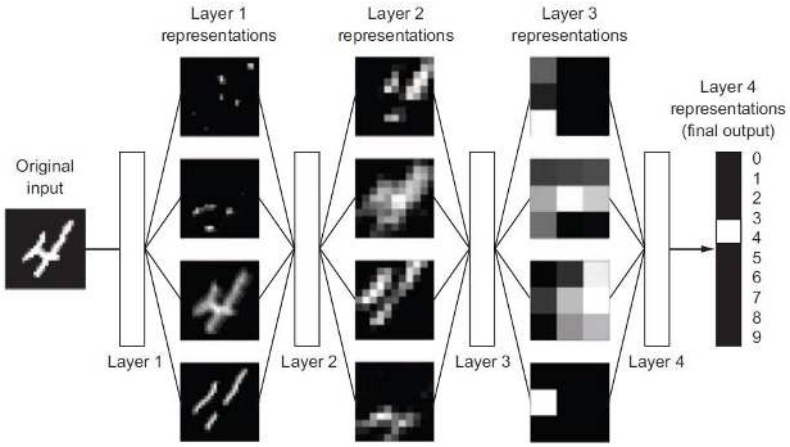


Figure 1.6 Deep representations learned by a digit-classification model

A deep network as a multistage information-distillation operation, where information goes through successive filters and comes out increasingly purified (that is, useful with regard to some task). So that's what deep learning is, technically: *a multistage way to learn data representations*.

How this learning happens?

Machine learning is about mapping inputs (such as images) to targets (such as the label "cat"), which is done by observing many examples of input and targets. Deep neural networks do this input-to-target mapping via a deep sequence of simple data transformations (layers) and that these data transformations are learned by exposure to examples.

Weights AKA Parameters

The specification of what a layer does to its input data is stored in the layer's *weights*, which in essence are a bunch of numbers. In technical terms, we'd say that the transformation implemented by a layer is *parameterized* by its weights (see figure 1.7). (Weights are also sometimes called the *parameters* of a layer.)

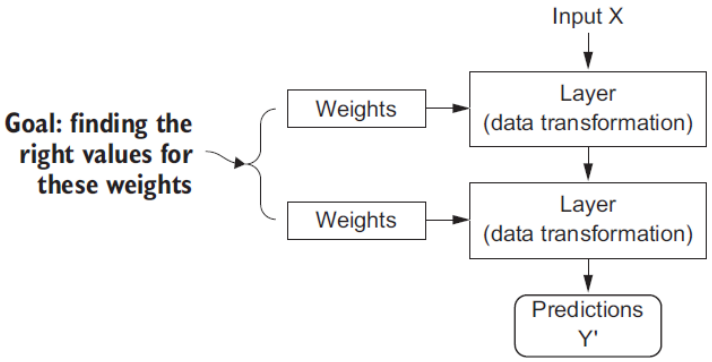


Figure 1.7 A neural network is parameterized by its weights.

In this context, learning means finding a set of values for the weights of all layers in a network, such that the network will correctly map example inputs to their associated targets. But here's the thing: a deep neural network can contain tens of millions of parameters. Finding the correct value for all of them may seem like a daunting task, especially given that modifying the value of one parameter will affect the behaviour of all the others!

Loss function AKA Objective function

To control something, first you need to be able to observe it. To control the output of

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

a neural network, you need to be able to measure how far this output is from what you expected. This is the *job of the loss function* of the network, also called the *objective function*. The loss function takes the predictions of the network and the true target (what you wanted the network to output) and computes a distance score, capturing how well the network has done on this specific example (see figure 1.8).

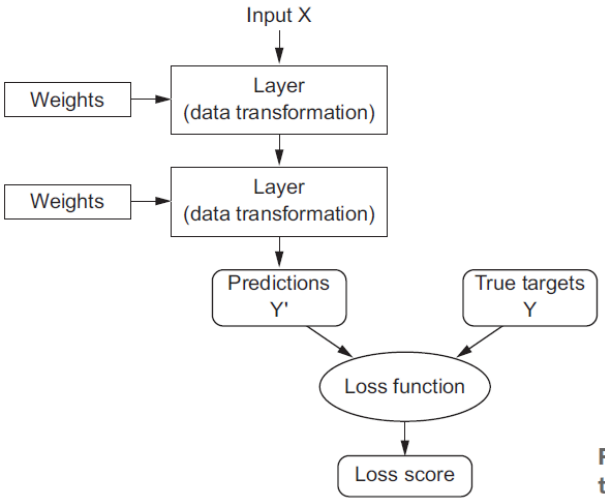


Figure 1.8 A loss function measures the quality of the network's output.

Optimizer

The fundamental trick in deep learning is to use this score as a feedback signal to adjust the value of the weights a little, in a direction that will lower the loss score for the current example (see figure 1.9). This adjustment is the job of the *optimizer*, which implements what's called the Backpropagation algorithm: the central algorithm in deep learning.

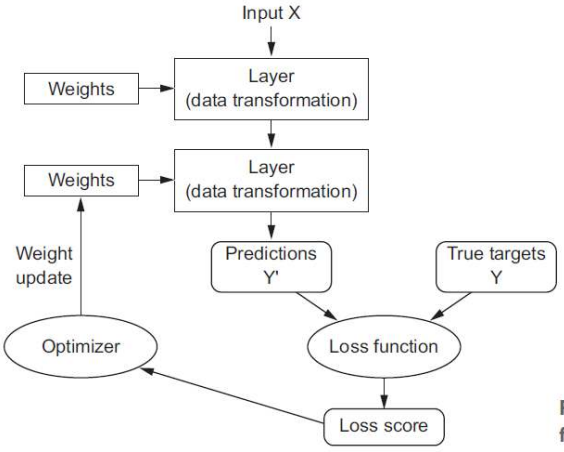


Figure 1.9 The loss score is used as a feedback signal to adjust the weights.

Trained Network

Initially, the weights of the network are assigned random values, so the network merely implements a series of random transformations. Naturally, its output is far from what it should ideally be, and the loss score is accordingly very high. But with every example the network processes, the weights are adjusted a little in the correct direction, and the loss score decreases. This is the *training loop*, which, repeated a sufficient number of times (typically tens of iterations over thousands of examples), yields weight values that minimize the loss function. A network with a minimal loss is one for which the outputs are as close as they can be to the targets: *a trained network*.

DL achieved so far?

Although deep learning is a fairly old subfield of machine learning, it only rose to prominence in the early 2010s. In the few years since, it has achieved nothing short of a revolution in the field, with remarkable results on perceptual problems such as seeing and hearing—problems involving skills that seem natural and intuitive to humans but have long been elusive for machines.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

In particular, deep learning has achieved the following breakthroughs, all in historically difficult areas of machine learning:

- ✓ Near-human-level image classification
- ✓ Near-human-level speech recognition
- ✓ Near-human-level handwriting transcription
- ✓ Improved machine translation
- ✓ Improved text-to-speech conversion
- ✓ Digital assistants such as Google Now and Amazon Alexa
- ✓ Near-human-level autonomous driving
- ✓ Improved ad targeting, as used by Google, Baidu, and Bing
- ✓ Improved search results on the web
- ✓ Ability to answer natural-language questions
- ✓ Superhuman Go playing

2.Perceptron Learning Algorithms

The Perceptron - Simple Model of Neural Networks

The Perceptron is a linear model used for binary classification. The perceptron is more widely known as a "single-layer perceptron" in neural network research to distinguish it from its successor the "multilayer perceptron." As a basic linear classifier, we consider the single-layer perceptron to be the simplest form of the family of feed-forward neural networks.

Definition of the Perceptron

The perceptron is a linear-model binary classifier with a simple input–output relationship as depicted in Figure 2-3, which shows we're summing n number of inputs times their associated weights and then sending this "net input" to a step function with a defined threshold. Typically, with perceptrons, this is a Heaviside step function with a threshold value of 0.5. This function will output a real valued single binary value (0 or a 1), depending on the input.

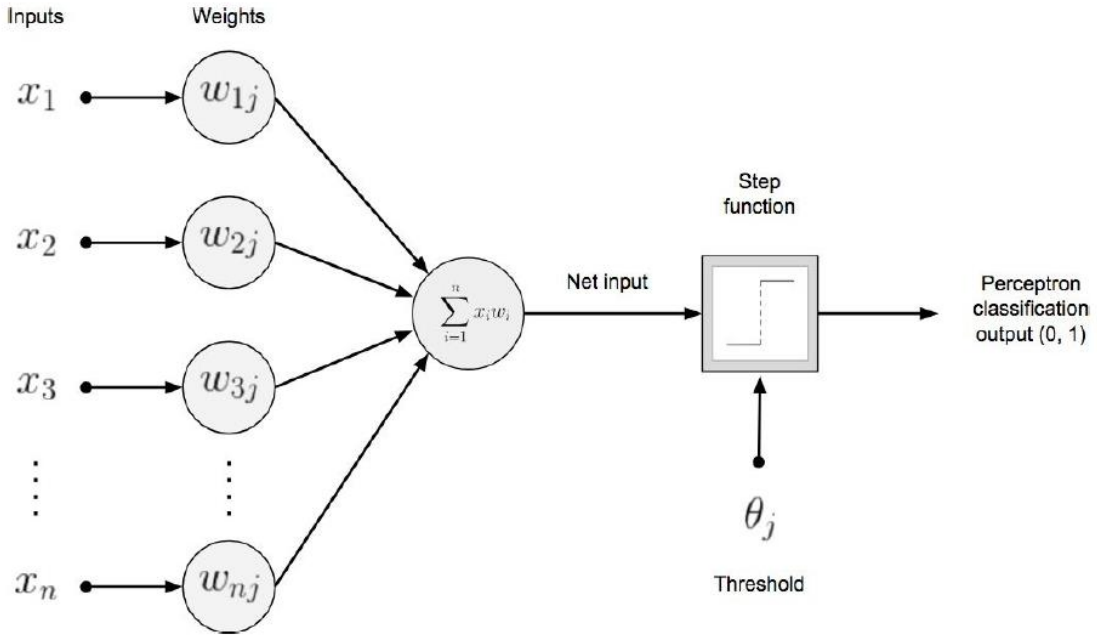


Figure 2-3. Single-layer perceptron

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

We can model the decision boundary and the classification output in the Heaviside step function equation, as follows:

$$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x >= 0 \end{cases}$$

To produce the net input to the activation function (here, the Heaviside step function) we take the dot product of the input and the connection weights. We see this summation in the left half of Figure 2-3 as the input to the summation function.

Table 2-1. The summation function parameters

Function parameter	Description
w	Vector of real-valued weights on the connections
w · **x**	Dot product ($\sum_{i=1}^{n} w_i x_i$)
n	Number of inputs to the perceptron
b	The bias term (input value does not affect its value; shifts decision boundary away from origin)

Table 2-1 provides an explanation of how the summation function is performed as well as notes about the parameters involved in the summation function. The output of the step function (activation function) is the output for the perceptron and gives us a classification of the input values.

If the bias value is negative, it forces the learned weights sum to be a much greater value to get a 1 classification output. The bias term in this capacity moves the decision boundary around for the model. Input values do not affect the bias term, but the bias term is learned through the perceptron learning algorithm.

History of the Perceptron

The perceptron was invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt. Early versions were intended to be implemented as a physical machine rather than a software program. The first software implementation was for the IBM 704, and then later it was implemented in the Mark I Perceptron machine. It also should be noted that McCulloch and Pitts introduced the basic concept of analysing neural activity in 1943 based on thresholds and weighted sums. These concepts were key in developing a model for later variations like the perceptron.

THE MARK I PERCEPTRON

The Mark I Perceptron was designed for *image recognition for military purposes* by the *US Navy*. The Mark I Perceptron had 400 photocells connected to artificial neurons in the machine, and the weights were implemented by potentiometers. Weight updates were *physically performed by electric motors*.

Types of Classification Problems

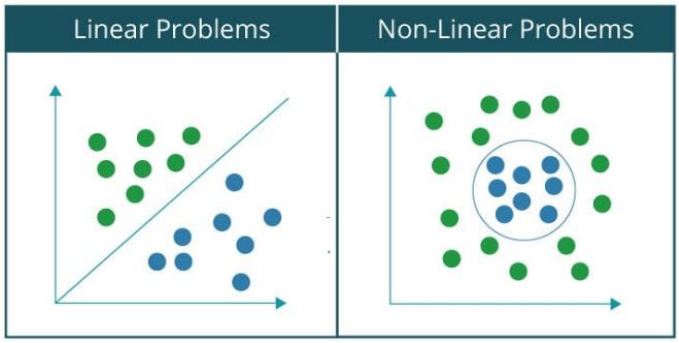
One can categorize all kinds of classification problems that can be solved using neural networks into two broad categories:

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

1) Linearly Separable Problems
2) Non-Linearly Separable Problems

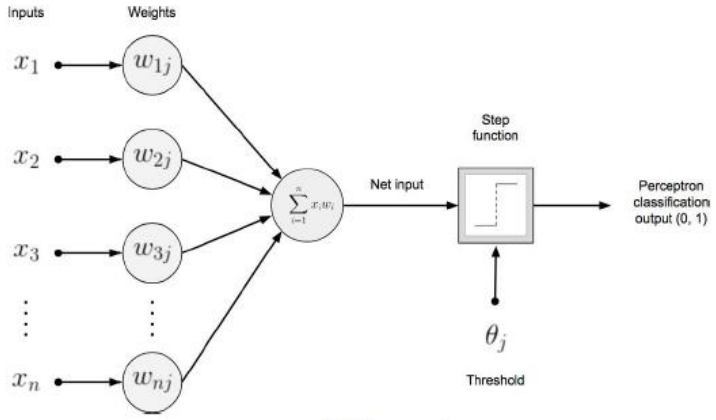
Let us visualize the difference between the two by plotting the graph of a linearly separable problem and non-linearly problem data set.



Basically, a problem is said to be linearly separable if you can classify the data set into two categories or classes using a single line. For example, separating cats from a group of cats and dogs. On the contrary, in case of a non-linearly separable problems, the data set contains multiple classes and requires non-linear line for separating them into their respective classes. For example, classification of handwritten digits.

Types of a Perceptron

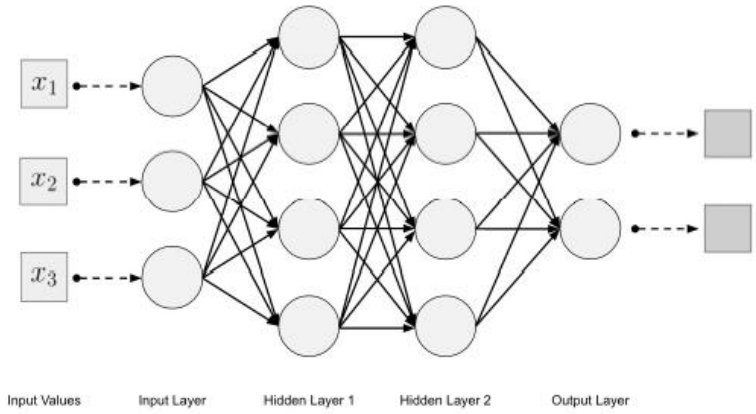
1) **Single Layer Perceptron:** One of the easiest ANN (Artificial Neural Networks) types consists of a feed-forward network and includes a threshold transfer inside the model. In single-layer perceptron's neurons are organized in one layer. The main objective of the single-layer perceptron model is to analyse the linearly separable objects with binary outcomes. A Single-layer perceptron can learn only linearly separable patterns.



Single-layer perceptron

2) **Multi-Layer Perceptron:** It is mainly similar to a single-layer perceptron model but has more hidden layers. In a multilayer perceptron's a group of neurons will be organized in multiple layers. Every single neuron present in the first layer will take the input signal and send a response to the neurons in the second layer and so on. A multilayer perceptron model has a greater processing power and can process linear and non-linear patterns.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

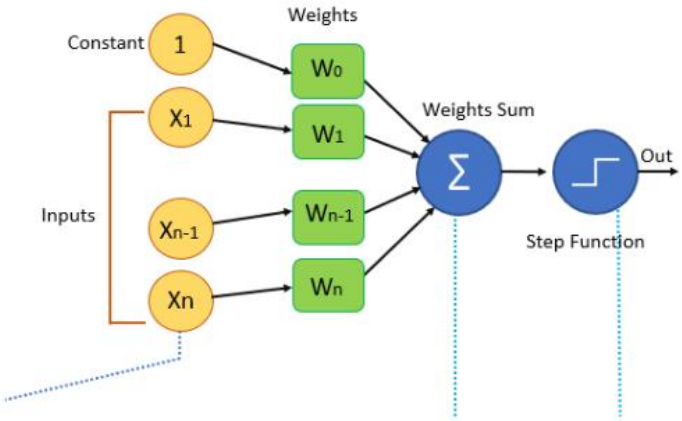


Multilayer neural network topology

- Forward Stage: From the input layer in the on stage, activation functions begin and terminate on the output layer.

- Backward Stage: In the backward stage, weight and bias values are modified per the model's requirement. The backstage removed the error between the actual output and demands originating backward on the output layer.

Primary Components of a Perceptron



1. **Neurons** - A neural network is made up of a collection of units or nodes which are called neurons.

2. **Synapse** - The getting neuron can obtain the sign, process the same, and sign the subsequent one. A neuron can send information or signals through the synapse to another adjacent neuron. It processes it and signals the subsequent one. This process in the perceptron algorithm continues until an output signal is generated.

3. **Input Nodes or Input Layer** - This is the *primary component* of Perceptron which accepts the initial data into the system for further processing. Each input node contains a real numerical value. All the *Features* of the model we want to train the neural network are taken as inputs in the perceptron algorithm. Inputs are denoted as x1, x2, x3, x4, … xn – 'x' in these inputs indicates the *feature value* and 'n' the *total occurrences* of these features.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

4. **Weight** - Weight parameter represents the strength of the connection between units. This is another most important parameter of Perceptron components. These are values that are calculated during the training of the model. Initially, we have to pass some random values as values to the weights and these values get automatically updated after each training error that is the values are generated during the training of the model. In some cases, weights can also be called as weight coefficients that occurs during hidden layers which is donated w1, w2, w3, w4, ... wn. Weight is directly proportional to the strength of the associated input neuron in deciding the output.

5. **Bias** – Bias is a special input type which allows the classifier to move the decision boundary around from its original position. The objective of the bias is to shift each point in a particular direction for a specified distance. Bias allows for higher quality and faster model training.

   If you notice, we have passed value one as input in the starting and W0 in the weights section. Bias is an element that adjusts the boundary away from origin to move the activation function left, right, up or down. Since we want this to be independent of the input features, we add constant one in the statement so that the features will not get affected by this and this value is known as Bias.

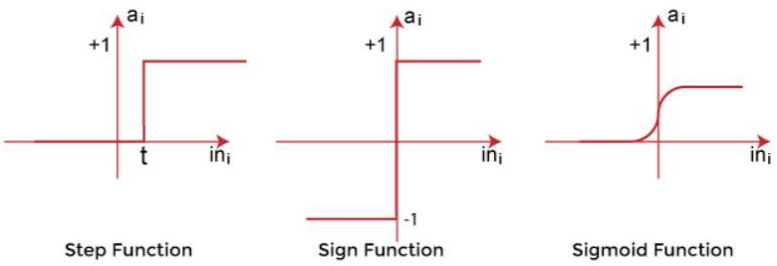
6. **Weighted Summation** - The multiplication of every feature or input value (xi) associated with corresponding weight values (wi) gives us a sum of values that are called weighted summation. This weighted sum is passed on to the so-called activation function

   $$\text{Weights sum} = \sum Wi * Xi \text{ (from } i=1 \text{ to } i=n) + (W0 * 1)$$

7. **Activation/Step Function** – Activation function applies step rule which converts the numerical value to 0 or 1 so that it will be easy for data set to classify. This is a process of reaching to result or outcome that help to determine whether the neuron will fire or not.

   Activation Function can be considered primarily as a step function.
   Types of Activation functions:

   - Sign function
   - Step function, and
   - Sigmoid function



Based on the type of value we need as output we can change the activation function. We can use the step function depending on the value required. Sigmoid function and sign functions can be used for values between 0 and 1 and 1 and -1, respectively. The sign function is a hyperbolic tangent function which is a zero centered function making it easy

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

the multi-layer neural networks. Rectified Linear Unit (ReLu) is another step function that is highly computational and can be used for values approaching zero – value more less than or more than zero.

The data scientist uses the activation function to take a subjective decision based on various problem statements and forms the desired outputs. Activation function may differ (e.g., Sign, Step, and Sigmoid) in perceptron models by checking whether the learning process is slow or has vanishing or exploding gradients.

Relating The Perceptron to The Biological Neuron

Although we don't have a complete model for how the brain works, we do see how the perceptron was modelled after the biological neuron. In this case, we can see how the perceptron takes input through connections with weights on them in a similar fashion to how synapses pass information to other biological neurons.

The Perceptron Learning Algorithm

Perceptron Algorithm is used in a supervised machine learning domain for classification. This algorithm changes the weights in the perceptron model until all input records are all correctly classified. The algorithm will not terminate if the learning input is not linearly separable.

The perceptron learning algorithm initializes the weight vector with small random values or 0.0s at the beginning of training. The perceptron learning algorithm takes each input record, as we can see in Figure 2-3, and computes the output classification to check against the actual classification label. To produce the classification, the columns (features) are matched up to weights where n is the number of dimensions in both our input and weights. The first input value is the bias input, which is always 1.0 because we don't affect the bias input. The first weight is our bias term in this diagram. The dot product of the input vector and the weight vector gives us the input to our activation function, as we've previously discussed.

If the classification is correct, no weight changes are made. If the classification is incorrect, the weights are adjusted accordingly. Weights are updated between individual training examples in an "online learning" fashion. This loop continues until all of the input examples are correctly classified. If the dataset is not linearly separable, the training algorithm will not terminate.

Perceptron Learning Steps

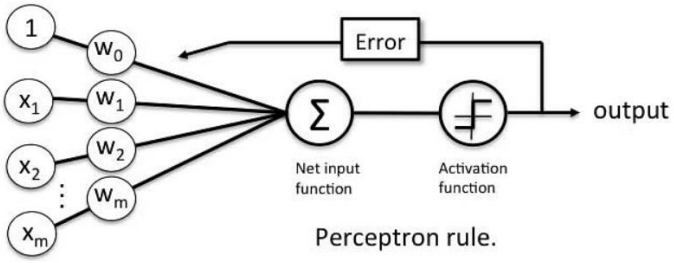
i. Features of the model we want to train should be passed as input to the perceptrons in the first layer.
ii. These inputs will be multiplied by the weights or weight coefficients and the production values from all perceptrons will be added.
iii. Adds the Bias value, to move the output function away from the origin.
iv. This computed value will be fed to the activation function (chosen based on the requirement, if a simple perceptron system activation function is step function).
v. The result value from the activation function is the output value.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Perceptron Learning Rule

Perceptron Learning Rule states that the algorithm would automatically learn the optimal weight coefficients. The input features are then multiplied with these weights to determine if a neuron fires or not.



Perceptron rule.

The Perceptron receives multiple input signals, and if the sum of the input signals exceeds a certain threshold, it either outputs a signal or does not return an output. In the context of supervised learning and classification, this can then be used to predict the class of a sample.

Error in Perceptron

In the Perceptron Learning Rule, the predicted output is compared with the known output. If it does not match, the error is propagated backward to allow weight adjustment to happen. Features added with perceptron make in deep neural networks. Back Propagation is the most important feature in these.

Back Propagation

After performing the first pass (based on the input and randomly given inputs) error will be calculated and the back propagation algorithm performs an iterative backward pass and try to find the optimal values for weights so that the error value will be minimized. To minimize the error back propagation algorithm will calculate partial derivatives from the error function till each neuron's specific weight, this process will give us complete transparency from total error value to a specific weight that is responsible for the error.

Example - Illustrating how a Perceptron works

As an example of how a perceptron works, let's take a closer look at the work of a politician. She is a member of parliament and a new law has to be voted on. Thus, the politician has to decide whether she agrees or disagrees with the bill (abstention is not possible in our example). The perceptron thus has a binary output, namely approval or rejection.

There are various sources of information available to politicians as inputs for their decision. On the one hand, there is an information paper with background information issued by the parliament. Furthermore, the politician can inform herself about various issues on the Internet or discuss them with colleagues. The politician weights her input, i.e., her sources of information, according to how trustworthy she considers them to be. She assigns a relatively low weight to the parliamentary information paper, for example, because she fears that the research is not detailed enough and should already tend in a certain direction. She then takes the sum of the information available to her, along with the weights, and passes it on to the activation function.

In this example, we can imagine this as the head of our politician. She now decides, on the basis of the inputs, whether she should agree to the proposed law or not. Even small details in the inputs can lead to a massive change in the politician's opinion.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Which Applications use Perceptrons?

Neural Networks are based on perceptrons and are mainly used in the field of machine learning. The goal here is mainly to learn structures in previous data and then predict new values. Some examples are:

i. **Object Recognition in Images:** Artificial Neural Networks can recognize objects in images or assign images to a class. Companies use this property in autonomous driving, for example, to recognize objects to which the car should react. Another area of application is in medicine when X-ray images are to be examined to detect an early stage of cancer, for example.

ii. **Customer Sentiment Analysis**: Through the Internet, customers have many channels to make their reviews of the brand or a product public. Therefore, companies need to keep track of whether customers are mostly satisfied or not. With a few reviews, which are classified as good or bad, efficient models can be trained, which can then automatically classify a large number of comments.

iii. **Spam Detection:** In many mail programs there is the possibility to mark concrete emails as spam. This data is used to train Machine Learning models that directly mark future emails as spam so that the end user does not even see them.

Advantages

- ✓ A multi-layered perceptron model can solve complex non-linear problems.
- ✓ With the help of single-layer perceptrons and especially multi-layer perceptrons, the so-called Neural Networks, complex predictions can be learned in the field of supervised learning.
- ✓ A trained perceptron is relatively easy to interpret and the learned weights can be used to make a statement about how important the inputs are.

Limitations

- Single-Layer Perceptrons cannot classify non-linearly separable data points.
- Complex problems, that involve a lot of parameters cannot be solved by Single-Layer Perceptrons.
- Multi-layer perceptrons only work really well with large data sets.
- Training multi-layer perceptrons is usually time-consuming and resource-intensive.

Summary

- ❖ An artificial neuron is a mathematical function conceived as a model of biological neurons, that is, a neural network.
- ❖ The perceptron is an algorithm from the field of supervised learning and represents the basic building block of a neural network. It is a function that maps its input “x,” which is multiplied by the learned weight coefficient, and generates an output value” f(x).
- ❖ Perceptron Learning Rule states that the algorithm would automatically learn the optimal weight coefficients. It can be used to learn complex relationships in data and apply them to new, previously unseen data.
- ❖ Single layer Perceptrons can learn only linearly separable patterns.
- ❖ Multilayer Perceptron or feedforward neural network is when individual perceptrons are built and connected in multiple layers i.e., two or more layers. They have the greater processing power and can process non-linear patterns as well.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

3.Probabilistic Modelling

What is Probabilistic Modelling?

Probabilistic modelling is the application of the principles of statistics to data analysis. It was one of the earliest forms of machine learning, and it's still widely used to this day. One of the main reasons why probabilistic modelling is so popular nowadays is that it provides natural protection against overfitting and allows for completely coherent inferences over complex forms from data.

It is a statistical approach that uses the effect of random occurrences or actions to forecast the possibility of future results. It is a quantitative modelling method that projects several possible outcomes that might even go beyond what has happened recently.

It considers new situations and a wide range of uncertainty while not underestimating dangers. The *three primary building blocks* of probabilistic modelling are adequate probability distributions, correct use of input information for these distribution functions, and proper accounting for the linkages and interactions between variables. The *downside* of the probabilistic modelling technique is that it needs meticulous development, a process that depends on several assumptions and large input data.

Importance of Probabilistic ML Models

One of the most significant advantages of the probabilistic modelling technique is that it provides a comprehensive understanding of the uncertainty associated with predictions. Using this method, we can quickly determine how confident any mobile learning model is and how accurate its prediction is.

An example of a probabilistic classifier that assigns a probability of 0.9 to the 'Dog' class suggests the classifier is quite confident that the animal in the image is a dog. It is heavily dependent on the opposing concepts of uncertainty and confidence. In reality, it is extremely helpful when used to key machine learning applications such as illness detection and autonomous driving. Furthermore, probabilistic outcomes would be beneficial for many Machine Learning related approaches, such as Active Learning.

Examples of Probabilistic Models

Straight Line Modelling

A straight-line probabilistic model is sometimes known as a linear regression model or a best-fit straight line. It's a best-fit line since it tries to reduce the size of all the different error components. A linear regression model may be computed using any basic spreadsheet or statistical software application. However, the basic computation is just dependent on a few variables. This is the implementation that is based on probabilistic modelling.

Weather and Traffic

Weather and traffic are two everyday phenomena that are both unpredictable and appear to have a link with one another. You are all aware that if the weather is cold and snow is falling, traffic will be quite difficult and you will be detained for an extended period of time. We could even go so far as to predict a substantial association between snowy weather and higher traffic mishaps.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Based on available data, we can develop a basic mathematical model of traffic accidents as a function of snowy weather to aid in the analysis of our hypothesis. All of these models are based on probabilistic modelling. It is one of the most effective approaches for assessing weather and traffic relationships.

Naive Bayes Algorithm

The next example of predictive modelling is the Naive Bayes method. It is an algorithm for supervised learning. This method, which is based on the Bayes theorem, is used to solve sorting difficulties. It is mostly employed in text classification using a high-dimensional training dataset.

It is one of the most basic and effective operational Classification algorithms for building fast machine-learning models that can make quick predictions. A probabilistic classifier is the Naive Bayes method. It indicates that it forecasts based on an object's likelihood. The following are more or less common examples of the Naive Bayes Algorithm:

- Spam Detection
- Emotional Analysis
- Article Categorization

Advantages of Probabilistic Models

Theoretically, probabilistic modelling is adequate. In other words, it is based on reliability and may simply indicate how secure any machine learning model is. It is a fantastic tool for dealing with uncertainty in performance evaluation and risk estimates. It offers critical data for operational and strategic decision-making processes.

It may be utilised in a flexible and integrated manner for probabilistic load-flow assessments, reliability analyses, voltage sag evaluation, and general scenario analysis. One of the most important advantages of probabilistic analysis is that it allows managers to participate in meaningful discourse about their risks.

Conclusion

Probabilistic Models are a great way to understand the trends that can be derived from the data and create predictions for the future. As one of the first topics that is taught in Machine Learning, the importance of probabilistic models is understated. These models provide a foundation for the machine learning models to understand the prevalent trends and their behaviour.

4.Early Neural Networks

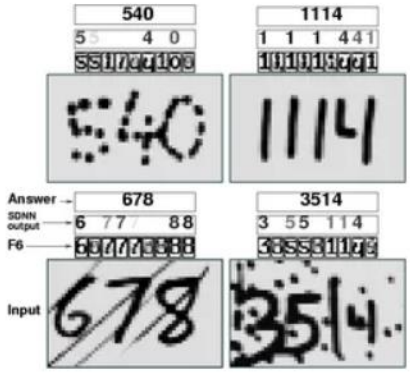
The core ideas of neural networks were investigated in toy forms as early as the 1950s, the approach took decades to get started. For a long time, the missing piece was an efficient way to train large neural networks. This changed in the mid-1980s, when multiple people independently rediscovered the Backpropagation algorithm— a way to train chains of parametric operations using gradient-descent optimization and started applying it to neural networks.

The first successful practical application of neural nets came in 1989 from Bell Labs, when *Yann LeCun* combined the earlier ideas of convolutional neural networks and backpropagation, and applied

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

them to the problem of classifying handwritten digits. The resulting network, dubbed *LeNet*, was used by the United States Postal Service in the 1990s to automate the reading of ZIP codes on mail envelopes.



Handwritten digits recognition. Source: http://yann.lecun.com/ex/research/index.html

https://medium.com/analytics-vidhya/a-brief-history-of-neural-networks-c234639a43f1 This article - A Brief History of Neural Networks is an extract from my book Data Science for Supply Chain Forecast.

5.How Deep Learning different from Machine Learning

The primary reason deep learning took off so quickly is that it offered better performance on many problems. But that's not the only reason. Deep learning also makes problem-solving much easier, because it completely automates what used to be the most crucial step in a machine-learning workflow: *feature engineering.*

Feature engineering

Previous machine-learning techniques—*shallow learning*—only involved transforming the input data into one or two successive representation spaces, usually via simple transformations such as high-dimensional non-linear projections (SVMs) or decision trees. But the refined representations required by complex problems generally can't be attained by such techniques. As such, humans had to go to great lengths to make the initial input data more amenable to processing by these methods: they had to manually engineer good layers of representations for their data. This is called *feature engineering*.

Deep learning, on the other hand, completely automates this step: with deep learning, you learn all features in one pass rather than having to engineer them yourself. This has greatly simplified machine-learning workflows, often replacing sophisticated multistage pipelines with a single, simple, end-to-end deep-learning model.

Could shallow methods be applied repeatedly to emulate the effects of deep learning?

In practice, there are fast-diminishing returns to successive applications of shallow-learning methods, because the optimal first representation layer in a three-layer model isn't the optimal first layer in a one-layer or two-layer model.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Joint Feature Learning

Deep learning allows a model to learn all layers of representation jointly, at the same time, rather than in succession (greedily, as it's called). With joint feature learning, whenever the model adjusts one of its internal features, all other features that depend on it automatically adapt to the change, without requiring human intervention. Everything is supervised by a single feedback signal: every change in the model serves the end goal. This is much more powerful than greedily stacking shallow models, because it allows for complex, abstract representations to be learned by breaking them down into long series of intermediate spaces (layers); each space is only a simple transformation away from the previous one.

Two essential characteristics

These are the two essential characteristics of how deep learning learns from data: the *incremental, layer-by-layer way in which increasingly complex representations are developed,* and the fact that *these intermediate incremental representations are learned jointly,* each layer being updated to follow both the representational needs of the layer above and the needs of the layer below. Together, these two properties have made deep learning vastly more successful than previous approaches to machine learning.

6.Scalars

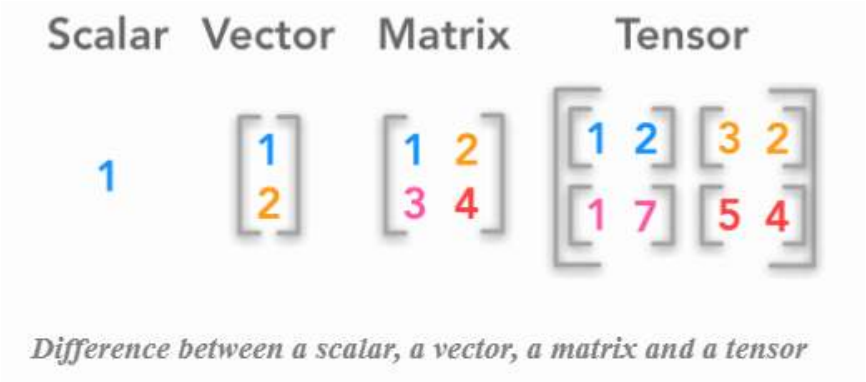
Why Math?

Linear algebra is a form of continuous rather than discrete mathematics, many computer scientists have little experience with it. A good understanding of linear algebra is essential for understanding and working with many machine learning algorithms, especially deep learning algorithms.

Linear algebra, probability and calculus are the 'languages' in which machine learning is formulated. Learning these topics will contribute a deeper understanding of the underlying algorithmic mechanics and allow development of new algorithms.

When confined to smaller levels, everything is math behind deep learning. So, it is essential to understand basic linear algebra before getting started with deep learning and programming it.

The core data structures behind Deep-Learning are Scalars, Vectors, Matrices and Tensors. In general, all current machine-learning systems use tensors as their basic data structure. At its core, a tensor is a container for data—almost always numerical data. So, it's a container for numbers. Before we can move on to tensors, we must first be familiar with scalars, vectors and matrices.



Difference between a scalar, a vector, a matrix and a tensor

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Scalars (0D tensors)

Scalars are single numbers and are an example of a 0th-order tensor. Scalars are written in lowercase and italics. For instance: *n*. A scalar is an element (such as real numbers) used to define a vector space. Scalar could be any type of number, for example, natural number, rational number, or irrational number. When we introduce them, we specify what kind of number they are.

For example, we might say
"Let $s \in R$ be the slope of the line," while defining a real-valued scalar, or "Let $n \in N$ be the number of units," while defining a natural number scalar.

In mathematics, it is necessary to describe the set of values to which a scalar belongs. The notation $x \in R$ states that the (lowercase) scalar value is an element of (or member of) the set of real-valued numbers, R. There are various sets of numbers of interest within machine learning. N represents the set of *positive* integers (1,2, 3…). Z represents the *integers*, which include *positive, negative and zero values*. Q represents the set of *rational* numbers that may be expressed as a fraction of two integers.

Few built-in scalar types are int, float, complex, bytes, Unicode in Python. In In NumPy a python library, there are 24 new fundamental data types to describe different types of scalars.
A tensor that contains only one number is called a scalar (*or scalar tensor, or 0-dimensional tensor, or 0D tensor*). In NumPy, a float32 or float64 number is a scalar tensor (or scalar array).

Note
In machine learning, a category in a classification problem is called a class.
Data points are called samples.
The class associated with a specific sample is called a label.
In the context of tensors, a dimension is often called an axis.

You can display the number of axes of a NumPy tensor via the ndim attribute; a scalar tensor has 0 axes (ndim == 0). The number of axes of a tensor is also called its *rank.*

Here's a Numpy scalar:

```
>>> import numpy as np
>>> x = np.array(12)
>>> x
array(12)
>>> x.ndim
0
```

7.Vectors

Vectors (1D tensors)

Vectors are ordered arrays of single numbers and are an example of 1st-order tensor. The numbers are arranged in order. We can identify each individual number by its index in that ordering. To identify the necessary component of a vector explicitly, the ith scalar element of a vector is written as x[i]. Vectors are written in lowercase, italics and bold type and put elements of a vector as a column

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

enclosed in square brackets. For instance: ***x.*** We also need to indicate the type of numbers stored in the vector.

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

A 1D tensor is said to have exactly one axis. Following is a NumPy vector:

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

This vector has five entries and so is called a 5-dimensional vector. Don't confuse a 5D vector with a 5D tensor! A 5D vector has only one axis and has five dimensions along its axis, whereas a 5D tensor has five axes (and may have any number of dimensions along each axis). Dimensionality can denote either the number of entries along a specific axis (as in the case of our 5D vector) or the number of axes in a tensor (such as a 5D tensor), which can be confusing at times. In the latter case, it's technically more correct to talk about a tensor of rank 5 (the rank of a tensor being the number of axes), but the ambiguous notation 5D tensor is common regardless.

Vectors are fragments of objects known as vector spaces. A vector space can be considered of as the entire collection of all possible vectors of a particular length (or dimension). The three-dimensional real-valued vector space, denoted by ℝ^3 is often used to represent our real-world notion of three-dimensional space mathematically.

In deep learning vectors usually represent feature vectors, with their original components defining how relevant a particular feature is. Such elements could include the related importance of the intensity of a set of pixels in a two-dimensional image or historical price values for a cross-section of financial instruments.

Simply put, we can think of vectors as identifying points in space, each element giving the coordinate along a different axis. In other words, A quantity described by multiple scalars, such as having both direction and magnitude, is called a vector.

For instance, scalars and vectors encode the difference between the speed of a car and its velocity. The velocity contains not only its speed but also its direction of travel.

8.Matrices

Matrices (2D tensors)

Matrices are two dimensional i.e., rectangular arrays consisting of numbers and are an example of 2nd-order tensors. If m and n are positive integers, that is m, n ∈ ℕ then the m×n matrix contains m*n numbers, with m rows and n columns.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Matrices are written in uppercase, italics and bold. For instance: ***A.*** The full m×n matrix can be written as:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

It is often useful to abbreviate the full matrix component display into the following expression. The components of matrix A are written $[a_{ij}]$ where i refers to the row element and j refers to the column element.

$$A=[a_{ij}]_{m\times n}$$

An array of vectors is a matrix, or 2D tensor. A matrix has two axes (often referred to rows and columns). You can visually interpret a matrix as a rectangular grid of numbers. This is a NumPy matrix:

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

The entries from the first axis are called the rows, and the entries from the second axis are called the columns. In the previous example, [5, 78, 2, 34, 0] is the first row of x, and [5, 6, 7] is the first column.

Importance of Matrices

In Deep Learning

In deep learning neural network weights are stored as matrices, while feature inputs are stored as vectors. Formulating the problem in terms of linear algebra allows compact handling of these computations. By casting the problem in terms of tensors and utilising the machinery of linear algebra, rapid training times on modern GPU hardware can be obtained.

In Machine Learning

Matrices and matrix mathematics is important in Machine Learning for a number of reasons:

- Data Cluster Manipulation - Machine Learning operations often involve retrieving, using and storing clusters of data points. Matrices are an efficient way to handle this type of data.
- Mathematical Formulas and Program Code - Matrices can be represented in compact mathematical formulas which can be programmed for easy access.
- Custom Computing Hardware - Custom computing hardware such as GPUs and TPUs are constructed for the efficient and rapid processing of data.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

9.Higher Dimensional Tensors

Tensors

The more general entity of a tensor encapsulates the scalar, vector and the matrix. It is sometimes necessary — both in the physical sciences and machine learning — to make use of tensors with order that exceeds two. i.e., In some cases, we'll need an array with more than two axes. In the general case, an array of numbers arranged on a regular grid with a varying number of axes is called a tensor.

In short, A tensor is a n-dimensional array with n>2.

We denote tensor in bold and uppercase letter, and identify the element of tensor at coordinates (i, j, k) by writing:

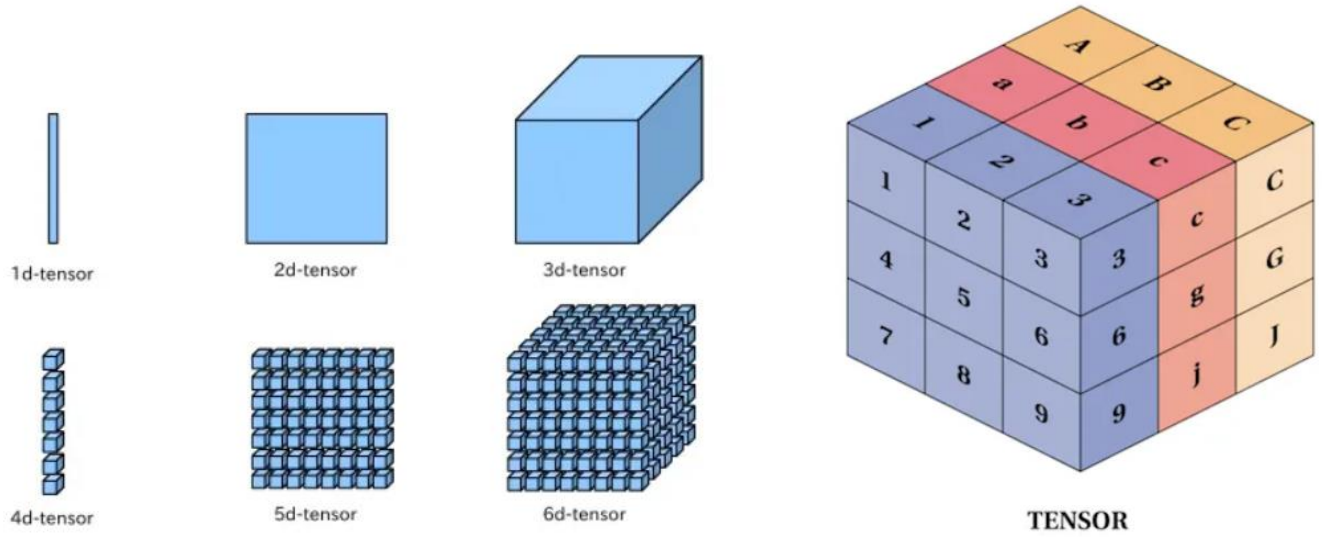
$$\mathbf{X}$$

$$X_{i,j,k}$$

3D Tensors and Higher-Dimensional Tensors

Consider tensor as a container of numbers, and the container could be in any dimension. For example, scalars, vectors, and matrices, are considered as the simplest tensors:

- Scalar is a 0-dimensional tensor
- Vector is a 1-dimensional tensor
- Matrix is a 2-dimensional tensor



Thereby, we can deduce that a 3-D tensor is a cube, 4-D tensor is a vector of cubes, 5-D tensor is a matrix of cubes, 6-D tensor is a cube of cubes, etc.

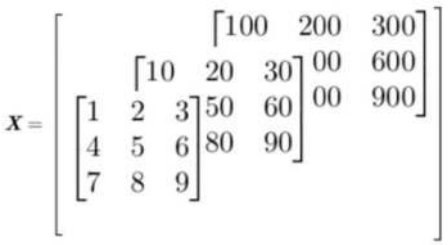
So, if you pack such matrices in a new array, you obtain a 3D tensor, which you can visually interpret as a cube of numbers. By packing 3D tensors in an array, you can create a 4D tensor, and so on.

The following code shows how to create a 3-D tensor with NumPy.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

```
x = np.array([[[1, 2, 3],[4, 5, 6],[7, 8, 9]],
              [[10, 20, 30],[40, 50, 60],[70, 80, 90]],
              [[100, 200, 300],[400, 500, 600],[700, 800, 900]]])
```



3-D tensor

We use Python libraries like TensorFlow or PyTorch in order to declare tensors, rather than nesting matrices.

Application of Tensors in ML and DL

- In machine learning, and deep learning in particular, a 3rd-order tensor can be used to describe the intensity values of multiple channels (red, green and blue) from a two-dimensional image.
- In deep learning, you'll generally manipulate tensors that are 0D to 4D, although you may go up to 5D if you process video data.
- In industry, commonly, we store time series data in 3-D tensor, image data in 4-D tensor, video data in 5-D tensor, etc.

10.Manipulating Tensors

Key Attributes of Tensors

A tensor is defined by three key attributes:

i. Number of axes (rank)—For instance, a 3D tensor has three axes, and a matrix has two axes. This is also called the tensor's ndim in Python libraries such as NumPy.

ii. Shape—This is a tuple of integers that describes how many dimensions the tensor has along each axis. For instance, the previous matrix example has shape (3, 5), and the 3D tensor example has shape (3, 3, 5). A vector has a shape with a single element, such as (5,), whereas a scalar has an empty shape, ().

iii. Data type (usually called dtype in Python libraries)—This is the type of the data contained in the tensor; for instance, a tensor's type could be float32, uint8, float64, and so on. On rare occasions, you may see a char tensor. Note that string tensors don't exist in NumPy (or in most other libraries), because tensors live in pre allocated, contiguous memory segments: and strings, being variable length, would preclude the use of this implementation.

For Example:

Number of axes of the tensor = 3
Shape = (60000, 28, 28)
Data type = uint8

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

So, what we have here is a 3D tensor of 8-bit integers. More precisely, it's an array of 60,000 matrices of 28 × 28 integers. Each such matrix is a grayscale image, with coefficients between 0 and 255.

Manipulating Tensors in NumPy

We'll use the MNIST dataset, a classic in the machine-learning community, which has been around almost as long as the field itself and has been intensively studied. It's a set of 60,000 training images, plus 10,000 test images, assembled by the National Institute of Standards and Technology (the NIST in MNIST) in the 1980s. You can think of "solving" MNIST as the "Hello World" of deep learning—it's what you do to verify that your algorithms are working as expected.

You can see some MNIST samples in figure 2.1.

Figure 2.1 MNIST sample digits

First, we load the MNIST dataset:

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Next, we display the number of axes of the tensor train_images, the ndim attribute:

```
>>> print(train_images.ndim)
3
```

Here's its shape:

```
>>> print(train_images.shape)
(60000, 28, 28)
```

And this is its data type, the dtype attribute:

```
>>> print(train_images.dtype)
uint8
```

Let's display the fourth digit in this 3D tensor, using the library Matplotlib (part of the standard scientific Python suite); see figure 2.2.

```
digit = train_images[4]
import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

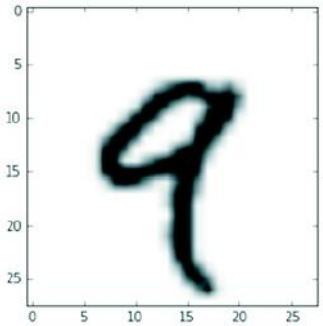


Figure 2.2 The fourth sample in our dataset

We selected a specific digit alongside the first axis using the syntax train_images[i].

Selecting specific elements in a tensor is called *tensor slicing.*

Let's look at the tensor-slicing operations you can do on NumPy arrays.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

The following example selects digits #10 to #100 (#100 isn't included) and puts them in an array of shape (90, 28, 28):

```
>>> my_slice = train_images[10:100]
>>> print(my_slice.shape)
(90, 28, 28)
```

It's equivalent to this more detailed notation, which specifies a start index and stop index for the slice along each tensor axis. Note that: is equivalent to selecting the entire axis:

```
>>> my_slice = train_images[10:100, :, :]
>>> my_slice.shape
(90, 28, 28)
>>> my_slice = train_images[10:100, 0:28, 0:28]
>>> my_slice.shape
(90, 28, 28)
```

Equivalent to the previous example

Also equivalent to the previous example

In general, you may select between any two indices along each tensor axis. For instance, in order to select 14 × 14 pixels in the bottom-right corner of all images, you do this:

```
my_slice = train_images[:, 14:, 14:]
```

It's also possible to use negative indices. Much like negative indices in Python lists, they indicate a position relative to the end of the current axis. In order to crop the images to patches of 14 × 14 pixels centered in the middle, you do this:

```
my_slice = train_images[:, 7:-7, 7:-7]
```

The Notion of Data Batches

In general, the first axis (axis 0, because indexing starts at 0) in all data tensors you'll come across in deep learning will be the *samples axis* (sometimes called the *samples dimension*). In the MNIST example, samples are images of digits.

In addition, deep-learning models don't process an entire dataset at once; rather, they break the data into small batches. Concretely, here's one batch of our MNIST digits, with batch size of 128:

```
batch = train_images[:128]
```

And here's the next batch:

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

And the nth batch:

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

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

11.Vector Data

Real-world examples of Data Tensors

The data you'll manipulate will almost always fall into one of the following categories:

- Vector data—2D tensors of shape *(samples, features)*

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

- Timeseries data or sequence data—3D tensors of shape *(samples, timesteps, features)*
- Images—4D tensors of shape *(samples, height, width, channels)* or *(samples, channels, height, width)*
- Video—5D tensors of shape *(samples, frames, height, width, channels)* or *(samples, frames, channels, height, width)*

Vector data

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

Let's take a look at two examples:

- An actuarial dataset of people, where we consider each person's age, ZIP code, and income. Each person can be characterized as a vector of 3 values, and thus an entire dataset of 100,000 people can be stored in a 2D tensor of shape *(100000, 3).*
- A dataset of text documents, where we represent each document by the counts of how many times each word appears in it (out of a dictionary of 20,000 common words). Each document can be encoded as a vector of 20,000 values (one count per word in the dictionary), and thus an entire dataset of 500 documents can be stored in a tensor of shape *(500, 20000).*

12.Time Series Data (AKA) Sequence Data

Whenever time matters in your data (or the notion of sequence order), it makes sense to store it in a 3D tensor with an explicit time axis. Each sample can be encoded as a sequence of vectors (a 2D tensor), and thus a batch of data will be encoded as a 3D tensor (see figure 2.3).

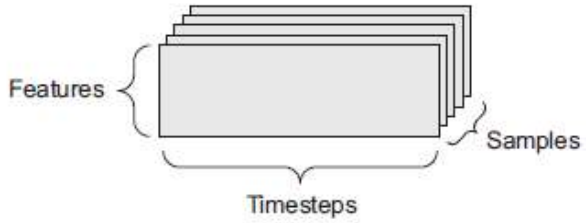


Figure 2.3 A 3D timeseries data tensor

The *time axis* is always the second axis (axis of index 1), by convention. Let's look at a few examples:

- A dataset of stock prices. Every minute, we store the current price of the stock, the highest price in the past minute, and the lowest price in the past minute. Thus, every minute is encoded as a 3D vector, an entire day of trading is encoded as a 2D tensor of shape (390, 3) (there are 390 minutes in a trading day), and 250 days' worth of data can be stored in a 3D tensor of shape (250, 390, 3). Here, each sample would be one day's worth of data.

- A dataset of tweets, where we encode each tweet as a sequence of 280 characters out of an alphabet of 128 unique characters. In this setting, each character can be encoded as a binary vector of size 128 (an all-zeros vector except for a 1 entry at the index corresponding to the character). Then each tweet can be encoded as a 2D tensor of shape (280, 128), and a dataset of 1 million tweets can be stored in a 3D tensor of shape (1000000, 280, 128).

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

13.Image Data

Images typically have three dimensions: *height, width*, and *colour depth*. Although grayscale images (like our MNIST digits) have only a single colour channel and could thus be stored in 2D tensors, by convention image tensors are always 3D, with a one-dimensional colour channel for grayscale images. A batch of 128 grayscale images of size 256 × 256 could thus be stored in a tensor of shape (128, 256, 256, 1), and a batch of 128 colour images could be stored in a tensor of shape (128, 256, 256, 3) (See figure 2.4).

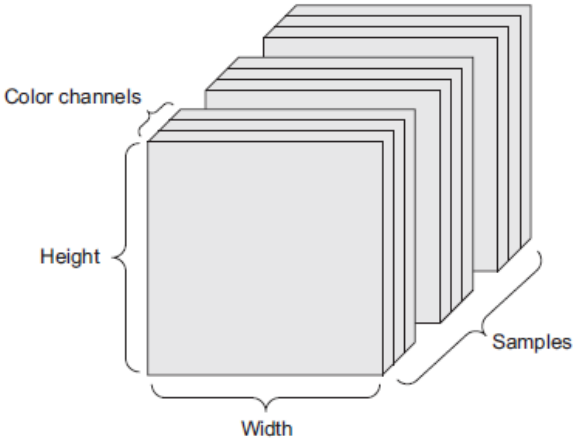


Figure 2.4 A 4D image data tensor (channels-first convention)

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

14.Video Data

Video data is one of the few types of real-world data for which you'll need 5D tensors. A video can be understood as a sequence of frames, each frame being a colour image. Because each frame can be stored in a 3D tensor *(height, width, colour_depth)*, a sequence of frames can be stored in a 4D tensor *(frames, height, width, colour_depth),* and thus a batch of different videos can be stored in a 5D tensor of shape *(samples, frames, height, width, colour_depth).*

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

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

UNIT - II NEURAL NETWORKS (Page No. 34-56)

1.About Neural Network Pg. 37

2.Building Blocks of Neural Network Pg. 38

- Relationship between the Network, Layers, Loss Function and Optimizer
- Layers: The Building Blocks of Neural Network
- Models: Networks of Layers
- Loss Functions and Optimizers: Keys to configuring the Learning Process

3.Optimizers Pg. 40

- Optimization Algorithms and Optimizers
- Different Types of Optimizers and their Advantages
  - Gradient Descent
  - Stochastic Gradient Descent
  - Mini-Batch Gradient Descent
  - Momentum
  - AdaGrad
  - AdaDelta
  - Adam

4.Activation Functions Pg. 42

- Different Types of Activation Functions
  - Linear Activation Function
  - Sigmoid Activation Function
  - Tanh Activation Function
  - Hard Tanh Activation Function
  - Softmax Activation Function
  - Rectified Linear Activation Function

5.Loss Functions Pg. 44

- Loss Function Notation
- Loss Functions for Regression
  - Mean Squared Error (MSE) Loss Function
  - Other loss functions for regression
    - Mean Absolute Error (MAE) Loss Function

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

- Mean Squared Log Error (MSLE) Loss Function
- Mean Absolute Percentage Error (MAPL) Loss Function

➢ Regression Loss Function Discussion

- Loss Functions for Classification
  - ➢ Hinge Loss Function
  - ➢ Logistic Loss Function
- Loss Functions for Reconstruction

6.Data Pre-Processing for Neural Networks Pg. 48

- Vectorization
- Value Normalization
- Handling Missing Values
- Feature Extraction
  - ➢ Algorithms for Feature Extraction

7.Feature Engineering Pg. 50

- Importance of Feature Engineering
- Example for Feature Engineering
  - ➢ Ways to Solve the Problem using Feature Engineering
- History of Feature Engineering
- Feature Engineering in Modern Deep Learning

8.Overfitting and Underfitting Pg. 51

- Dealing with Overfitting
  - ➢ Reasons for Overfitting
  - ➢ Strategies to fix Overfitting
- Dealing with Underfitting
  - ➢ Reasons for Underfitting
  - ➢ Strategies to fix Underfitting
- Optimal Fitting: What is a Good Fit in Machine Learning?

9.Hyperparameters Pg. 54

- Layer Size
- Magnitude Hyperparameters
  - ➢ Learning Rate

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

- Momentum
- Regularization
  - Dropout
  - DropConnect
  - L1 Penalty
  - L2 Penalty
- Sparsity

This document is available free of charge on studocu

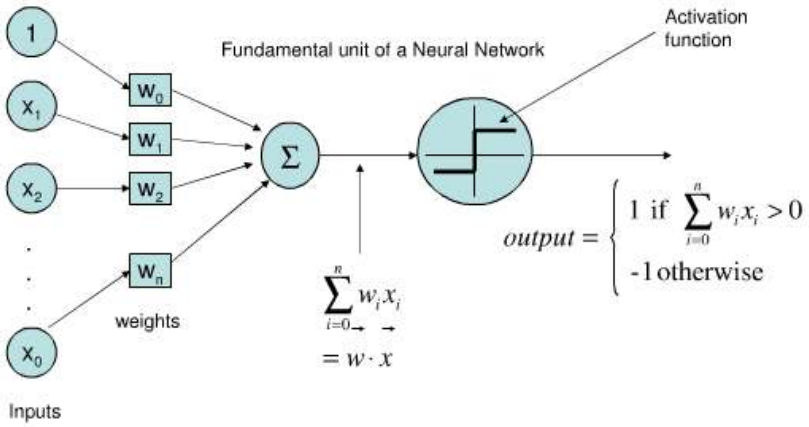
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

1.About Neural Networks

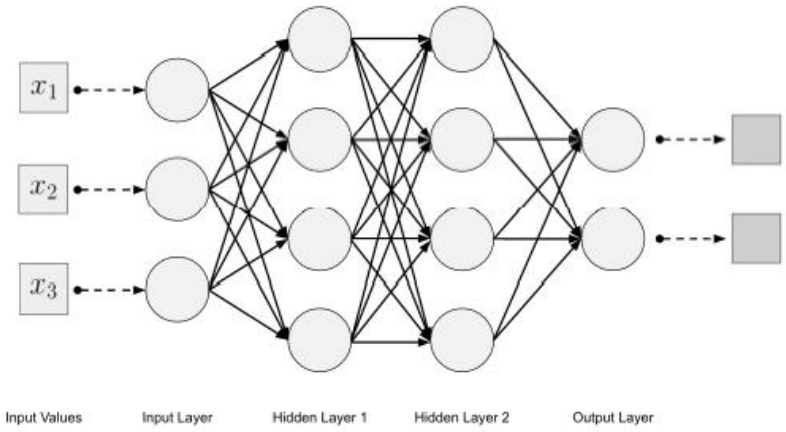
Neural networks are a computational model that shares some properties with the animal brain in which many simple units are working in parallel with no centralized control unit. The weights between the units are the primary means of long-term information storage in neural networks. Updating the weights is the primary way the neural network learns new information.

The behaviour of neural networks is shaped by its network architecture. A network's architecture can be defined (in part) by the following:

- Number of neurons
- Number of layers
- Types of connections between layers



The most well-known and simplest-to-understand neural network is the feedforward multilayer neural network. It has an input layer, one or many hidden layers, and a single output layer. Each layer can have a different number of neurons and each layer is fully connected to the adjacent layer. The connections between the neurons in the layers form an acyclic graph.



Multilayer neural network topology

A feed-forward multilayer neural network can represent any function, given enough artificial neuron units. It is generally trained by a learning algorithm called backpropagation learning. Backpropagation uses gradient descent [*Gradient descent is an optimization algorithm which is commonly-used to train machine learning models and neural networks. Training data helps these models learn over time, and the cost function within gradient descent specifically acts as a barometer, gauging its accuracy with each iteration of parameter updates*.] on the weights of the connections in a neural network to minimize the error on the output of the network.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

2.Building Blocks of Neural Network

Relationship between the Network, Layers, Loss Function and Optimizer

Training a neural network revolves around the following objects:

- Layers, which are combined into a network (or model)
- The input data and corresponding targets
- The loss function, which defines the feedback signal used for learning
- The optimizer, which determines how learning proceeds

You can visualize their interaction as illustrated in figure 3.1: the network, composed of layers that are chained together, maps the input data to predictions. The loss function then compares these predictions to the targets, producing a loss value: a measure of how well the network's predictions match what was expected. The optimizer uses this loss value to update the network's weights.

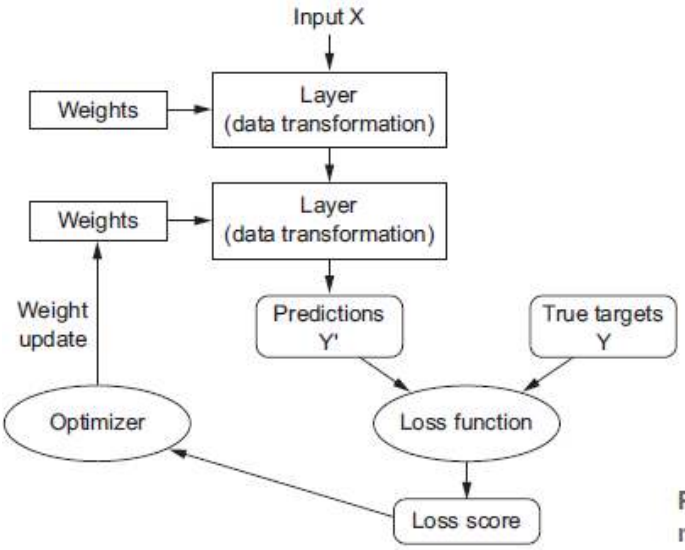


Figure 3.1 Relationship between the network, layers, loss function, and optimizer

Layers: The Building Blocks of Neural Network

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

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

Building deep-learning models in Keras is done by clipping together compatible layers to form useful data-transformation pipelines. The notion of layer compatibility here refers specifically to the fact that every layer will only accept input tensors of a certain shape and will return output tensors of a certain shape.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

```
from keras import layers

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

A dense layer with 32 output units

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

Building on the layer concept, we see that the multilayer neural network has the following:

- **A Single Input Layer** - This layer is how we get input data (vectors) fed into our network. The number of neurons in an input layer is typically the same number as the input feature to the network.

- **One or many Hidden Layers, Fully Connected** - Input layers are followed by one or more hidden layers in a feed-forward neural network. The weight values on the connections between the layers are how neural networks encode the learned information extracted from the raw training data. Hidden layers are the key to allowing neural networks to model nonlinear functions, as we saw from the limitations of the single-layer perceptron networks.

- **A Single Output Layer -** We get the answer or prediction from our model from the output layer. Given that we are mapping an input space to an output space with the neural network. model, the output layer gives us an output based on the input from the input layer. Depending on the setup of the neural network, the final output may be a real-valued output (regression) or a set of probabilities (classification). This is controlled by the type of activation function we use on the neurons in the output layer. The output layer typically uses either a softmax or sigmoid activation function for classification.

- **Connections between layers -** In a fully connected feed-forward network, the connection between layers are the outgoing connections from all neurons in the previous layer to all of the neurons in the next layer. We change these weights progressively as our algorithm finds the best solution it can with the backpropagation learning algorithm. We can understand the weights mathematically by thinking of them as the parameter vector describing the machine learning process as optimizing the parameter vector (e.g., "weights" here) to minimize error.

Models: Networks of Layers

A deep-learning model is a directed, acyclic graph of layers. The most common instance is a linear stack of layers, mapping a single input to a single output. But as you move forward, you'll be exposed to a much broader variety of network topologies.

Some common ones include the following:

- Two-branch networks
- Multi-head networks
- Inception blocks

Loss Functions and Optimizers: Keys to configuring the Learning Process

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

- *Loss function (objective function)*—The quantity that will be minimized during training. It represents a measure of success for the task at hand.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

- *Optimizer*—Determines how the network will be updated based on the loss function. It implements a specific variant of stochastic gradient descent (SGD). i.e., The mechanism through which the network will update itself based on the data it sees and its loss function.

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

3.Optimizers

Optimization Algorithms and Optimizers

Training a model in machine learning involves finding the best set of values for the parameter vector of the model. We use methods of optimization such as gradient descent to find good values for the parameter vector to minimize loss across our training dataset. The right optimization algorithm can reduce training time exponentially.

Many people may be using optimizers while training the neural network without knowing that the method is known as *optimization*. *Optimizers* are algorithms or methods used to change the attributes of your neural network such as weights and learning rate in order to reduce the losses. How you should change your weights or learning rates of your neural network to reduce the losses is defined by the optimizers you use. Optimization algorithms or strategies are responsible for reducing the losses and to provide the most accurate results possible.

Different Types of Optimizers and their Advantages

i. Gradient Descent

Gradient Descent is the most basic but most used optimization algorithm. It's used heavily in linear regression and classification algorithms. Backpropagation in neural networks also uses a gradient descent algorithm.

Gradient descent is a first-order optimization algorithm which is dependent on the first order derivative of a loss function. It calculates that which way the weights should be altered so that the function can reach a minima. Through backpropagation, the loss is transferred from one layer to another and the model's parameters also known as weights are modified depending on the losses so that the loss can be minimized.

Advantages	Disadvantages
✓ Easy computation. ✓ Easy to implement. ✓ Easy to understand.	○ May trap at local minima. ○ Weights are changed after calculating gradient on the whole dataset. So, if the dataset is too large than this may take years to converge to the minima. ○ Requires large memory to calculate gradient on the whole dataset.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

ii. Stochastic Gradient Descent

It's a variant of Gradient Descent. It tries to update the model's parameters more frequently. In this, the model parameters are altered after computation of loss on each training example. So, if the dataset contains 1000 rows SGD will update the model parameters 1000 times in one cycle of dataset instead of one time as in Gradient Descent.

As the model parameters are frequently updated parameters have high variance and fluctuations in loss functions at different intensities.

Advantages	Disadvantages
✓ Frequent updates of model parameters hence, converges in less time. ✓ Requires less memory as no need to store values of loss functions. ✓ May get new minima's.	○ High variance in model parameters. ○ May shoot even after achieving global minima. ○ To get the same convergence as gradient descent needs to slowly reduce the value of learning rate.

iii. Mini-Batch Gradient Descent

It's best among all the variations of gradient descent algorithms. It is an improvement on both SGD and standard gradient descent. It updates the model parameters after every batch. So, the dataset is divided into various batches and after every batch, the parameters are updated.

Advantages
✓ Frequently updates the model parameters and also has less variance. ✓ Requires medium amount of memory.

All types of Gradient Descent have some challenges:

1) Choosing an optimum value of the learning rate. If the learning rate is too small than gradient descent may take ages to converge.
2) Have a constant learning rate for all the parameters. There may be some parameters which we may not want to change at the same rate.
3) May get trapped at local minima.

iv. Momentum

Momentum was invented for reducing high variance in SGD and softens the convergence. It accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction. One more hyperparameter is used in this method known as momentum symbolized by 'γ'.

Advantages	Disadvantages
✓ Reduces the oscillations and high variance of the parameters. ✓ Converges faster than gradient descent.	○ One more hyper-parameter is added which needs to be selected manually and accurately.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

v. AdaGrad

One of the disadvantages of all the optimizers explained is that the learning rate is constant for all parameters and for each cycle. This optimizer changes the learning rate. It changes the learning rate 'η' for each parameter and at every time step 't'. It's a type second order optimization algorithm. It works on the derivative of an error function.

Advantages	Disadvantages
✓ Learning rate changes for each training parameter. ✓ Don't need to manually tune the learning rate. ✓ Able to train on sparse data.	○ Computationally expensive as a need to calculate the second order derivative. ○ The learning rate is always decreasing results in slow training.

vi. AdaDelta

It is an extension of AdaGrad which tends to remove the decaying learning Rate problem of it. Instead of accumulating all previously squared gradients, AdaDelta limits the window of accumulated past gradients to some fixed size w. In this exponentially moving average is used rather than the sum of all the gradients.

Advantages	Disadvantages
✓ Now the learning rate does not decay and the training does not stop.	○ Computationally expensive.

vii. Adam

Adam (Adaptive Moment Estimation) works with momentums of first and second order. The intuition behind the Adam is that we don't want to roll so fast just because we can jump over the minimum, we want to decrease the velocity a little bit for a careful search. In addition to storing an exponentially decaying average of past squared gradients like AdaDelta, Adam also keeps an exponentially decaying average of past gradients.

Advantages	Disadvantages
✓ The method is too fast and converges rapidly. ✓ Rectifies vanishing learning rate, high variance.	○ Computationally costly.

4.Activation Functions

The functions that govern the artificial neuron's behaviour are called *activation functions*. We use activation functions to propagate the output of one layer's nodes forward to the next layer (up to and including the output layer). The transmission of that input is known as *forward propagation*. Activation functions transform the combination of inputs, weights, and biases. Products of these transforms are input for the next node layer. Many (but not all) nonlinear transforms used in neural networks transform the data into a convenient range, such as 0 to 1 or –1 to 1. Activation functions are a scalar-to-scalar function, yielding the neuron's activation. When an artificial neuron passes on a nonzero value to another artificial neuron, it is said to be activated. *Activations* are the values passed on to the next layer from each previous layer. These values are the output of the activation function of each artificial neuron.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

The neurons in each layer all use the same type of activation function (most of the time). For the input layer, the input is the raw vector input. The input to neurons of the other layers is the output (activation) of the previous layer's neurons. As data moves through the network in a feed-forward fashion, it is influenced by the connection weights and the activation function type. We use activation functions for hidden neurons in a neural network to introduce nonlinearity into the network's modelling capabilities.

Refer Page No. 17 & 18 for more information (Covered in Unit-1 Page No. 17)

Different Types of Activation Functions

a) Linear Activation Function

A linear transform (see Figure 2-11) is basically the identity function, and f(x) = Wx, where the dependent variable has a direct, proportional relationship with the independent variable. In practical terms, it means the function passes the signal through unchanged. We see this activation function used in the input layer of neural networks.

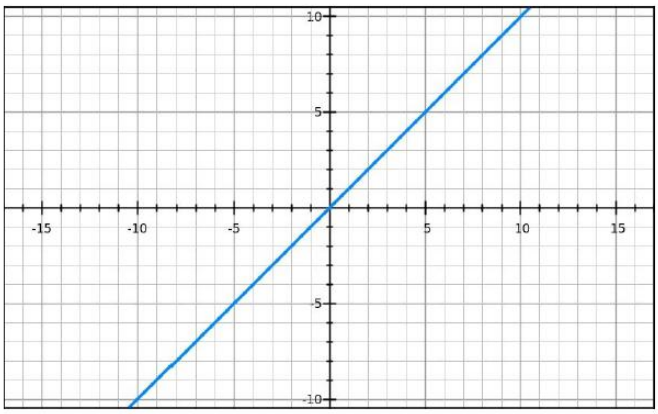


Figure 2-11. Linear activation function

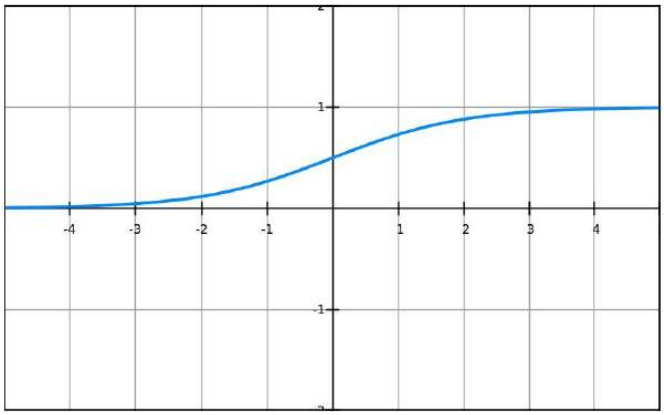


Figure 2-12. Sigmoid activation function

b) Sigmoid Activation Function

Like all logistic transforms, sigmoid can reduce extreme values or outliers in data without removing them. The vertical line in Figure 2-12 is the decision boundary. A sigmoid activation function outputs an independent probability for each class.

c) Tanh Activation Function

Pronounced "tanh," tanh is a hyperbolic trigonometric function (see Figure 2-13). Unlike the Sigmoid function, the normalized range of tanh is –1 to 1. The advantage of tanh is that it can deal more easily with negative numbers.

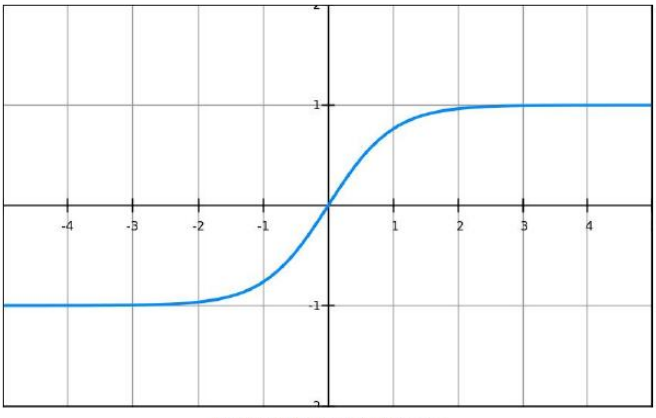


Figure 2-13. Tanh activation function

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

d) Hard Tanh Activation Function

Similar to tanh, hard tanh simply applies hard caps to the normalized range. Anything more than 1 is made into 1, and anything less than –1 is made into –1. This allows for a more robust activation function that allows for a limited decision boundary.

e) Softmax Activation Function

Softmax is a generalization of logistic regression inasmuch as it can be applied to continuous data (rather than classifying binary) and can contain multiple decision boundaries. It handles multinomial labelling systems. Softmax is the function you will often find at the output layer of a classifier. The softmax activation function returns the probability distribution over mutually exclusive output classes.

For the case in which we have a large set of labels (e.g., thousands of labels), we'd use the variant of the softmax activation function called the *hierarchical softmax activation function*. This variant decomposes the labels into a tree structure, and the softmax classifier is trained at each node of the tree to direct the branching for classification.

f) Rectified Linear Activation Function

Rectified linear is a more interesting transform that activates a node only if the input is above a certain quantity. While the input is below zero, the output is zero, but when the input rises above a certain threshold, it has a linear relationship with the dependent variable f(x) = max(0, x), as demonstrated in Figure 2-14.

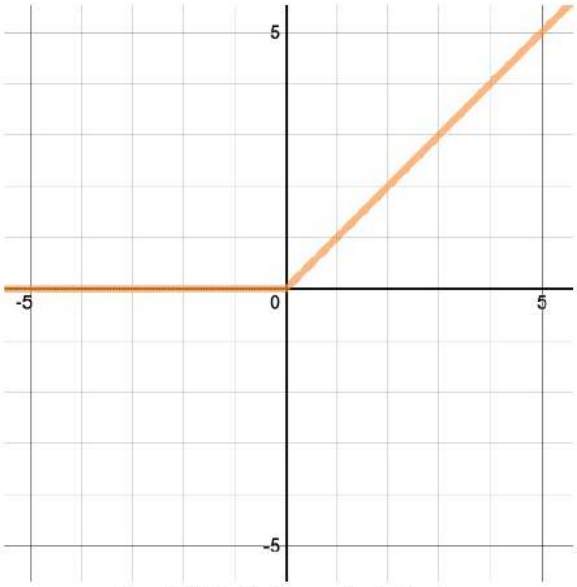


Figure 2-14. Rectified linear activation function

Rectified linear units (ReLU) are the current state of the art because they have proven to work in many different situations. ReLU activation functions have shown to train better in practice than sigmoid activation functions.

5.Loss Functions

The Loss functions quantify how close a given neural network is to the ideal toward which it is training. The idea is simple. We calculate a metric based on the error we observe in the network's predictions. We then aggregate these errors over the entire dataset and average them and now we have a single number representative of how close the neural network is to its ideal.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Looking for this ideal state is equivalent to finding the parameters (weights and biases) that will minimize the "loss" incurred from the errors. In this way, loss functions help reframe training neural networks as an optimization problem. In most cases, these parameters cannot be solved for analytically, but, more often than not, they can be approximated well with iterative optimization algorithms like gradient descent.

Loss Function Notation

- Consider the dataset gathered to train a neural net. Let "N" denote the number of samples (set of inputs with corresponding outcomes) that have been gathered.
- Consider the nature of the input and output collected. Each data point records some set of unique input features and output features. Let "P" denote the number of input features gathered and "M" denote the number of output features that have been observed.
- We will use (X, Y) to denote the input and output data we collected. Note that there will be N such pairs where the input is a collection of P values and the output Y is a collection of M values. We will denote the *ith* pair in the dataset as X_i and Y_i.
- We will use $\hat{Y}$ to denote the output of the neural net. Of course, $\hat{Y}$ is the network's guess at Y and therefore it will also have M features.
- We will use the notation $h(X_i) = \hat{Y}$ to denote the neural network transforming the input X_i to give the output $\hat{Y}_i$. We will alter this notation a little later to emphasize its dependence on weights and biases.
- When referring to *jth* output feature, we will use it as a subscript firmly linking our notation to a matrix where the rows are different data points and the columns are the different unique features. Thus $y_{i,j}$ refers to the *jth* feature observed in the *ith* sample collected.
- We will represent the loss function by L(W, b).

Given the data available, the loss function notation indicates that its value depends only on W and b, the weights and the biases of the neural network. This cannot be emphasized enough. In the universe of a given neural network with a set number of layers, configurations, and so on that will be trained on a given set of data, the value of the loss function depends exclusively on the state of the network, as defined by the weights and biases. Wiggle those and our losses wiggle. Wiggle those for a given input and our output wiggles.

So, our notation $h(X) = \hat{Y}$ should be conditioned on a set of weights and biases; thus, we will amend our notation to say $h_{w,b}(X) = \hat{Y}$.

The loss function boils-down the difference between desired and predicted, be that they are vectors, into a single number.

Loss Functions for Regression

- Mean Squared Error (MSE) Loss Function

When working on a regression model that requires a real valued output, we use the squared loss function, much like the case of ordinary least squares in linear regression. Consider the case in which we have to predict only one output feature (M = 1). The error in a prediction is squared and is averaged over the number of data points, plain and simple, as we see in the following equation for MSE loss:

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

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

What if M is greater than one and we are looking to predict multiple output features for a given set of input features? In this case, the desired and predicted entities, Y and Ŷ, respectively, are an ordered list of numbers, or, in other words, vectors. Let's now look at another variation of the MSE loss function:

$$L(W, b) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{M}\sum_{j=1}^{M}(\hat{y}_{ij} - y_{ij})^2$$

The inner sigma in the preceding equation is the square of the Euclidean distance. Note that N, the size of your dataset, and M, the number of features the network has to predict, are constants. In a lot of use cases, the M is dropped and a division by two is added for mathematical convenience. In the following equation, we see the version of MSE for regression:

$$L(W, b) = \frac{1}{2N}\sum_{i=1}^{N}\sum_{j=1}^{M}(\hat{y}_{ij} - y_{ij})^2$$

In a technical sense, the MSE is a *convex loss function*. However, when dealing with hidden layers in neural networks, the convex property no longer holds true, because we could have multiple parameters sets of values resulting in the same loss value. Optimizing the MSE is equivalent to optimizing for the mean.

Other loss functions for regression

Although the MSE is used widely, it is quite sensitive to outliers, and this is something that you should consider when picking a loss function. When picking a stock to invest in, we want to take the outliers into account. But perhaps when buying a house, we don't. In this case, what is of most interest is what most people would pay for it. In which case, we are more interested in the median and less so in the mean.

- Mean Absolute Error (MAE) Loss Function

In a similar vein, an alternative to the MSE loss is the mean absolute error (MAE) loss, as shown in the following equation:

$$L(W, b) = \frac{1}{2N}\sum_{i=1}^{N}\sum_{j=1}^{M} | \hat{y}_{ij} - y_{ij} |$$

This simply averages the absolute error over the entire dataset.

- Mean Squared Log Error (MSLE) Loss Function

Another loss function used for regression is the mean squared log error (MSLE):

$$L(W, b) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}(\log \hat{y}_{ij} - \log y_{ij})^2$$

- Mean Absolute Percentage Error (MAPL) Loss Function

Finally, we have mean absolute percentage error (MAPE) loss:

$$L(W, b) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}\frac{100 \times | \hat{y}_{ij} - y_{ij} |}{y_{ij}}$$

Regression Loss Function Discussion

These are all valid choices, and there certainly is no single loss function that will outperform all other loss functions for every scenario. The MSE is very widely used and is a safe bet in most cases. So is the MAE. The MSLE and the MAPE are worth taking into consideration if our network is predicting outputs that vary largely in range. Suppose that a network is to predict two output variables: one in the

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

range of [0, 10] and the other in the range of [0, 100]. In this case, the MAE and the MSE will penalize the error in the second output more significantly than the first. The MAPE makes it a relative error and therefore doesn't discriminate based on the range. The MSLE squishes the range of all the outputs down, simply how 10 and 100 translate to 1 and 2 (in log base 10). Although MSLE and MAPE are approaches to handling large ranges, *common practice with neural networks* is to normalize inputs to a suitable range and use the MSE or MAE to optimize for either the mean or the median.

Loss Functions for Classification

We can build neural networks to bin data points into different categories; for example, fraud | not fraud. However, when building neural networks for classification problems, the focus is often on attaching probabilities to these classifications (30 percent fraud|70 percent not fraud). These differing scenarios require different loss functions.

- Hinge Loss Function

Hinge loss is the most commonly used loss function when the network must be optimized for a hard classification. For example, 0 = no fraud and 1 = fraud, which by convention is called a 0-1 classifier. The 0,1 choice is somewhat arbitrary and –1, 1 is also seen in lieu of 0–1. Hinge loss is also seen in a class of models called maximum-margin classification models (e.g., support vector machines).
Following is the equation for hinge loss when data points must be categorized as –1 or 1:

$$L(W, b) = \frac{1}{N}\sum_{i=1}^{N} max(0, 1 - y_{ij} \times \hat{y}_{ij})$$

The hinge loss is mostly used for binary classifications. There are also extensions for multiclass classification (e.g., "one versus all," "one versus one") for the hinge loss. Note that like the MSE, the hinge loss is known to be a convex function.

- Logistic Loss Function

Logistic loss functions are used when probabilities are of greater interest than hard classifications. Great examples of these would be flagging potential fraud, with a human-in-the-loop solution or predicting the "probability of someone clicking on an ad," which can then be linked to a currency number.

Predicting valid probabilities means generating numbers between 0 and 1. Predicting valid probabilities also means making sure the probability of mutually exclusive outcomes should sum to one. For this reason, it is essential that the very last layer of a neural network used in classification is a softmax. Note that the sigmoid activation function also will give valid values between 0 and 1. However, you cannot use it for scenarios in which the outputs are mutually exclusive, because it does not model the dependencies between the output values.

Now that we have made sure our neural network will produce valid probabilities for the classes we have. We want to optimize for what is formally called the "*maximum likelihood*." In other words, we want to maximize the probability we predict for the correct class AND we want to do so for every single sample we have.

$$L(W, b) = -\sum_{i=1}^{N}\sum_{j=1}^{M} y_{i,j} \times \log \hat{y}_{i,j}$$

Loss Functions for Reconstruction

This set of loss functions relates to what is called reconstruction. The idea is simple. A neural network is trained to recreate its input as closely as possible. So, why is this any different from memorizing the entire dataset? The key here is to tweak the scenario so that the network is forced to learn commonalities and features across the dataset.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

In one approach, the number of parameters in the network is constrained such that the network is forced to compress the data and then re-create it. Another often-used approach is to corrupt the input with meaningless "noise" and train the network to ignore the noise and learn the data. Examples of these kinds of neural nets are restricted Boltzmann machines, autoencoders, and so on. These neural networks all use loss functions that are rooted in information theory.

Following is the equation for Kullback-Leibler (KL) Divergence:

$$D_{KL}(Y \mid\mid \hat{Y}) = -\sum_{i=1}^{N} Y_i \times \log\left(\frac{Y_i}{\hat{Y}_i}\right)$$

The Kullback-Leibler Divergence score, or KL divergence score, quantifies how much one probability distribution differs from another probability distribution. The KL divergence between two distributions Q and P is often stated using the following notation: KL(P || Q).

6.Data Pre-Processing for Neural Networks

Data pre-processing aims at making the raw data at hand more amenable to neural networks. This includes vectorization, normalization, handling missing values, and feature extraction.

Vectorization

All inputs and targets in a neural network must be tensors of floating-point data (or, in specific cases, tensors of integers). Whatever data you need to process—sound, images, text—you must first turn into tensors, a step called *data vectorization*. For instance, in the text-classification, starting from text represented as lists of integers (standing for sequences of words), use one-hot encoding to turn them into a tensor of *float32* data.

Value Normalization

In general, it isn't safe to feed into a neural network data that takes relatively large values (For example, multidigit integers, which are much larger than the initial values taken by the weights of a network) or data that is heterogeneous (for example, data where one feature is in the range 0–1 and another is in the range 100–200). Doing so can trigger large gradient updates that will prevent the network from converging. To make learning easier for your network, your data should have the following characteristics:

- *Take small values*—Typically, most values should be in the 0–1 range.
- *Be homogenous*—That is, all features should take values in roughly the same range.

Additionally, the following stricter normalization practice is common and can help, although it isn't always necessary.

- Normalize each feature independently to have a mean of 0.
- Normalize each feature independently to have a standard deviation of 1.

Handling Missing Values

You may sometimes have missing values in your data. For instance, in the house-price example, the first feature (the column of index 0 in the data) is the per capita crime rate. What if this feature wasn't available for all samples? You'd then have missing values in the training or test data.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

In general, with neural networks, it's safe to input missing values as 0, with the condition that 0 isn't already a meaningful value. The network will learn from exposure to the data that the value 0 means missing data and will start ignoring the value.

Note that if you're expecting missing values in the test data, but the network was trained on data without any missing values, the network won't have learned to ignore missing values! In this situation, you should artificially generate training samples with missing entries: copy some training samples several times, and drop some of the features that you expect are likely to be missing in the test data.

Feature Extraction

Feature Extraction aims to reduce the number of features in a dataset by creating new features from the existing ones (and then discarding the original features). These new reduced set of features should then be able to summarize most of the information contained in the original set of features. In this way, a summarised version of the original features can be created from a combination of the original set. Another definition, Feature extraction refers to the process of transforming raw data into numerical features that can be processed while preserving the information in the original data set. It yields better results than applying machine learning directly to the raw data.

Feature extraction can be accomplished *manually or automatically*:

- Manual feature extraction requires identifying and describing the features that are relevant for a given problem and implementing a way to extract those features. In many situations, having a good understanding of the background or domain can help make informed decisions as to which features could be useful.

- Automated feature extraction uses specialized algorithms or deep networks to extract features automatically from signals or images without the need for human intervention. This technique can be very useful when you want to move quickly from raw data to developing machine learning algorithms. Wavelet scattering is an example of automated feature extraction. (A *wavelet scattering* network automates the extraction of low-variance features from real-valued time series and image data for use in machine learning and deep learning applications.)

Algorithms for Feature Extraction

In case of linear relationships between the different features
- Principle Components Analysis (PCA)
- Independent Component Analysis (ICA)
- Linear Discriminant Analysis (LDA)

To deal with non-linear cases
- Manifold Learning Algorithms such as Isomap, Locally Linear Embedding, Modified Locally Linear Embedding, Hessian Eigen Mapping, etc…
- t-distributed Stochastic Neighbor Embedding (t-SNE)
- Autoencoders

Another commonly used technique to reduce the number of features in a dataset is *Feature Selection*. The difference between Feature Selection and Feature Extraction is that feature selection aims instead to rank the importance of the existing features in the dataset and discard less important ones (no new features are created).

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

7.Feature Engineering

Feature engineering is the process of using your own knowledge about the data and about the machine-learning algorithm at hand (in this case, a neural network) to make the algorithm work better by applying hardcoded (non-learned) transformations to the data before it goes into the model. In many cases, it isn't reasonable to expect a machine learning model to be able to learn from completely arbitrary data. The data needs to be presented to the model in a way that will make the model's job easier.

Importance of Feature Engineering

Handcrafting features has been a hallmark of machine learning for a long time. Practitioners who win competitions in machine learning often study the dataset thoroughly and use many arcane tricks to make the learning process as simple as possible for their learning algorithm. These datasets are often columnar/tabular text data and we can apply domain knowledge to specific columns so feature creation is more direct. Handcrafted features tend to produce highly accurate models but take a lot of time and experience to produce. From a knowledge representation perspective, it's like reading a poorly written book versus a book that is well-written and easy to read. The former takes us a lot longer to read and we need to spend more energy to get the same out of it as the latter.

Image classification is an interesting example because handcrafting image features is more difficult than creating features for tabular data. The information in images is not constrained to stay in the same column and can be influenced by lighting, angle, and other issues. Feature extraction and creation for images needed a new approach, which in some part drove the evolution of CNNs.

Example for Feature Engineering

Let's look at an intuitive example. Suppose you're trying to develop a model that can take as input an image of a clock and can output the time of day (see figure 4.3).

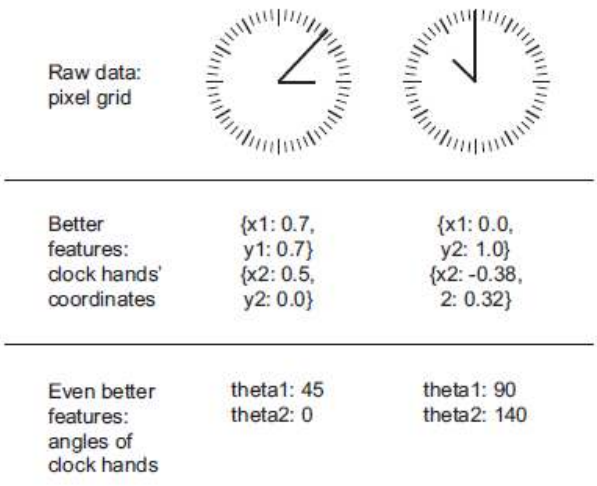


Figure 4.3 Feature engineering for reading the time on a clock

If you choose to use the raw pixels of the image as input data, then you have a difficult machine-learning problem on your hands. You'll need a convolutional neural network to solve it, and you'll have to expend quite a bit of computational resources to train the network.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Ways to Solve the Problem using Feature Engineering

But if you already understand the problem at a high level (you understand how humans read time on a clock face), then you can come up with much better input features for a machine-learning algorithm: for instance, it's easy to write a five-line Python script to follow the black pixels of the clock hands and output the (x, y) coordinates of the tip of each hand. Then a simple machine-learning algorithm can learn to associate these coordinates with the appropriate time of day.

You can go even further: do a coordinate change, and express the (x, y) coordinates as polar coordinates with regard to the centre of the image. Your input will become the angle *theta* of each clock hand. At this point, your features are making the problem so easy that no machine learning is required; a simple rounding operation and dictionary lookup are enough to recover the approximate time of day.

That's the *essence of feature engineering*: making a problem easier by expressing it in a simpler way. It usually requires understanding the problem in depth.

History of Feature Engineering

Before deep learning, feature engineering used to be critical, because classical shallow algorithms didn't have hypothesis spaces rich enough to learn useful features by themselves. The way you presented the data to the algorithm was essential to its success. For instance, before convolutional neural networks became successful on the MNIST digit-classification problem, solutions were typically based on hardcoded features such as the number of loops in a digit image, the height of each digit in an image, a histogram of pixel values, and so on.

Feature Engineering in Modern Deep Learning

Fortunately, modern deep learning removes the need for most feature engineering, because neural networks are capable of automatically extracting useful features from raw data. Does this mean you don't have to worry about feature engineering as long as you're using deep neural networks? *No*, for two reasons:

- Good features still allow you to solve problems more elegantly while using fewer resources. For instance, it would be ridiculous to solve the problem of reading a clock face using a convolutional neural network.
- Good features let you solve a problem with far less data. The ability of deep learning models to learn features on their own relies on having lots of training data available; if you have only a few samples, then the information value in their features becomes critical.

8.Overfitting and Underfitting

The fundamental issue in machine learning is the tension between optimization and generalization. *Optimization* refers to the process of adjusting a model to get the best performance possible on the training data (the learning in machine learning), whereas *generalization* refers to how well the trained model performs on data it has never seen before. The goal of the game is to get good generalization, of course, but you don't control generalization; you can only adjust the model based on its training data.

The Optimization algorithms first attempt to solve the problem of underfitting; that is, of taking a line that does not approximate the data well and making it approximate the data better. A straight line cutting across a curving scatterplot would be a good example of underfitting, as is illustrated in Figure 1-7.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

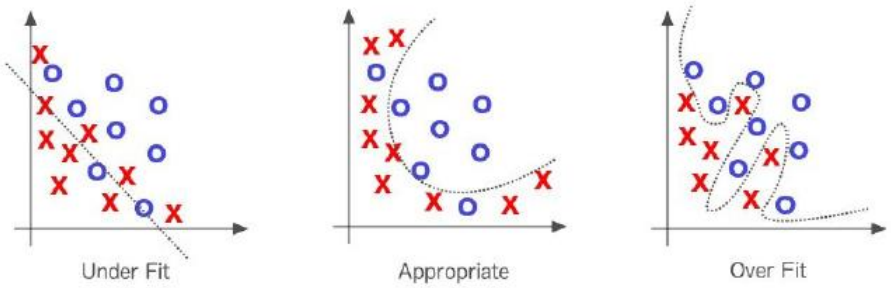
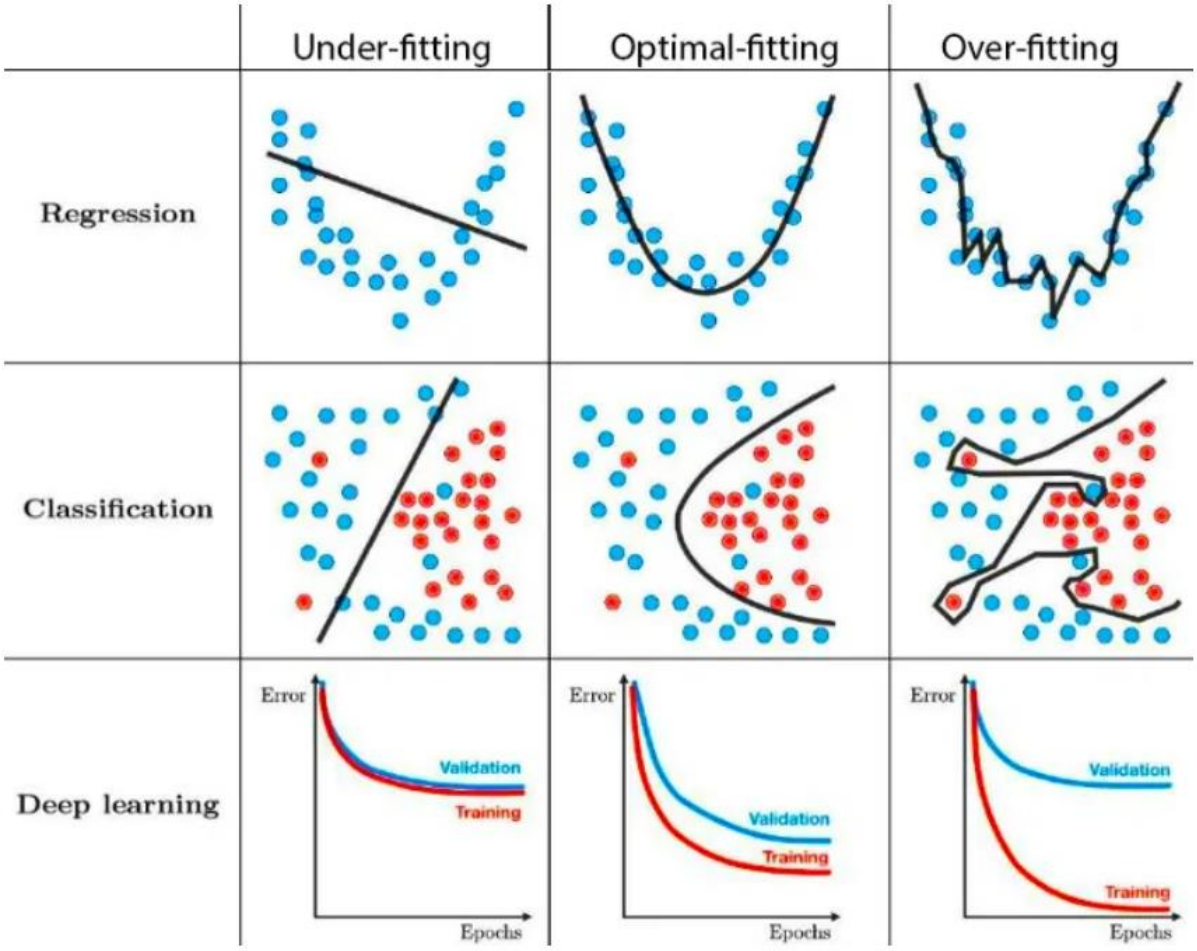


Figure 1-7. Underfitting and overfitting in machine learning

If the line fits the data too well, we have the opposite problem, called "**Overfitting**." Overfitting is the tendency of machine learning workflows to learn the training dataset so well that it performs poorly on unseen datasets. We'd say that this model does not generalize well to larger datasets. Solving underfitting is the priority, but much effort in machine learning is spent attempting not to overfit the line to the data. When we say a model overfits a dataset, we mean that it may have a low error rate for the training data, but it does not generalize well to the overall population of data in which we're interested.

Another way of explaining overfitting is by thinking about probable distributions of data. The training set of data that we're trying to draw a line through is just a sample of a larger unknown set, and the line we draw will need to fit the larger set equally well if it is to have any predictive power. We must assume, therefore, that our sample is loosely representative of a larger set. Our goal with models in machine learning is *to generalize* the information contained in our training dataset such that the model will perform well on more data from similar sources.



This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Dealing with Overfitting

All machine learning workflows have the tendency to overfit to some degree, the trick is to know when to stop training so that our model generalizes well yet picks up the least amount of overfitting possible. As we model more complex datasets with deep learning networks, we invariably use larger parameter counts in the networks. The trade-off becomes that we must add parameters at a rate that allows us to model more complex datasets while not doing it so fast that we introduce unnecessary overfitting.

Reasons for Overfitting

- Data used for training is not cleaned and contains noise (garbage values) in it.
- The model has a high variance.
- The size of the training dataset used is not enough.
- The model is too complex.

To detect overfitting, we can evaluate the performance on a held-out test set. It is important that this test set isn't used for training; otherwise, it won't inform us as to how the network will perform on unseen data.

Strategies to fix Overfitting

We can use one or more of the following:

- Increased regularization (L1, L2, Dropout, DropConnect).
- Early stopping.
- A larger training dataset.
- A smaller network.

Overfitting Rule and Parameter Count

When the model is not allowed to have enough parameters, it will not be accurate. As the model takes on too many parameters, the accuracy on the training data will look good, but the model will overfit and the holdout data accuracy will diverge from the training data accuracy.

Dealing with Underfitting

When a model has not learned the patterns in the training data well and is unable to generalize well on the new data, it is known as underfitting. An underfit model has poor performance on the training data and will result in unreliable predictions. Underfitting occurs due to high bias and low variance.

Reasons for Underfitting

- Data used for training is not cleaned and contains noise (garbage values) in it.
- The model has a high bias.
- The size of the training dataset used is not enough.
- The model is too simple.

Strategies to fix Underfitting

- Increase the number of features in the dataset.
- Increase model complexity.
- Reduce noise in the data.
- Increase the duration of training the data.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Optimal fitting: What is a Good Fit in Machine Learning?

To find the good fit model, you need to look at the performance of a machine learning model over time with the training data. As the algorithm learns over time, the error for the model on the training data reduces, as well as the error on the test dataset. If you train the model for too long, the model may learn the unnecessary details and the noise in the training set and hence lead to overfitting. In order to achieve a good fit, you need to stop training at a point where the error starts to increase.

9.Hyperparameters

In machine learning, we have both model parameters and then we have parameters we tune to make networks train better and faster. These tuning parameters are called hyperparameters, and they deal with controlling optimization function and model selection during training with our learning algorithm. Hyperparameter selection focuses on ensuring that the model neither underfits nor overfits the training dataset, while learning the structure of the data as quickly as possible. In other words, we define a hyperparameter as any configuration setting that is free to be chosen by the user that might affect performance.

A Few Cautionary notes about Hyperparameters:

- Some hyperparameters apply only some of the time.
- Moreover, changing a specific hyperparameter might affect the best settings for other hyperparameters.
- Some hyperparameters are incompatible with one another (e.g., AdaGrad + momentum).

Hyperparameters fall into several categories:

- ❖ Layer size
- ❖ Magnitude (Learning rate, Momentum)
- ❖ Regularization (Dropout, Drop Connect, L1, L2)
- ❖ Sparsity

Layer Size

Layer size is defined by the number of neurons in a given layer. Input and output layers are relatively easy to figure out because they correspond directly to how our modelling problem handles input and output. For the input layer, this will match up to the number of features in the input vector. For the output layer, this will either be a single output neuron or a number of neurons matching the number of classes we are trying to predict.

Deciding on neuron counts for each hidden layer is where hyperparameter tuning becomes a challenge. We can use an arbitrary number of neurons to define a layer and there are no rules about how big or small this number can be. However, how complex of a problem we can model is directly correlated to how many neurons are in the hidden layers of our networks. This might push you to begin with a large number of neurons from the start but these neurons come with a cost.

Depending on the deep network architecture, the connection schema between layers can vary. However, the weights on the connections are the parameters we must train. As we include more parameters in our model, we increase the amount of effort needed to train the network. Large parameter counts can lead to long training times and models that struggle to find convergence. There are also cases in which a larger model will sometimes converge easier (overfitting) because it will simply “memorize” the training data.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Magnitude Hyperparameters

Hyperparameters in the magnitude group involve the gradient, step size, and momentum.

Learning rate

Learning rate is considered one of the *key hyperparameters* in neural networks. The learning rate in machine learning is how fast we change the parameter vector as we move through search space. If the learning rate becomes too high, we can move toward our goal faster (least amount of error for the function being evaluated), but we might also take a step so large that we shoot right past the best answer to the problem, as well. Another side effect of learning rates that are large is that we run the risk of having unstable training that does not converge over time.

If we make our learning rate too small, it might take a lot longer than we'd like for our training process to complete. A low learning rate can make our learning algorithm inefficient. Learning rates are tricky because they end up being specific to the dataset and even to other hyperparameters. This creates a lot of overhead for finding the right setting for hyperparameters. We also can schedule learning rates to decrease over time according to some rule.

Momentum

We can speed up our training by increasing momentum, but we might lower the chance that the model will reach minimal error by overshooting the optimal parameter values. Momentum is a factor between 0.0 and 1.0 that is applied to the change rate of the weights over time. Typically, we see the value for momentum between 0.9 and 0.99.

Momentum helps the learning algorithm get out of spots in the search space where it would otherwise become stuck and it helps us produce better quality models.

Regularization

Regularization helps with the effects of out-of-control parameters by using different methods to minimize parameter size over time. Regularization is a measure taken against overfitting. Regularization's *main purpose* is to control overfitting in machine learning. *Overfitting* occurs when a model describes the training set but cannot generalize well over new inputs. Overfitted models have no predictive capacity for data that they haven't seen. *Geoffery Hinton* described the best way to build a neural network model: *Cause it to overfit, and then regularize it to death.*

Regularization for hyperparameters helps modify the gradient so that it doesn't step in directions that lead to overfit. Regularization includes the following:

- **Dropout** - A mechanism used to improve the training of neural networks by omitting a hidden unit. It also speeds training. Dropout is driven by randomly dropping a neuron so that it will not contribute to the forward pass and backpropagation. Dropout is a frequently used and usually quite effective approach to regularizing neural networks.
- **DropConnect** - Does the same thing as Dropout, but instead of choosing a hidden unit, it mutes the connection between two neurons.

  The *penalty methods* L1 and L2, in contrast, are a way of preventing the neural network parameter space from getting too big in one direction. They make large weights smaller.

- **L1 penalty** - *L1 regularization* is considered computationally inefficient in the non-sparse case, has sparse outputs, and includes built-in feature selection. L1 regularization multiplies the

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

absolute value of weights rather than their squares. This function drives many weights to zero while allowing a few to grow large, making it easier to interpret the weights.

- **L2 penalty** - In contrast, *L2 regularization* is computationally efficient due to it having analytical solutions and non-sparse outputs, but it does not do feature selection automatically for us. The "L2" regularization function, a common and simple hyperparameter, adds a term to the objective function that decreases the squared weights. You multiply half the sum of the squared weights by a coefficient called the weight-cost. L2 improves generalization, smooths the output of the model as input changes, and helps the network ignore weights it does not use.

In mathematical notation, we see regularization represented by the coefficient *lambda*, controlling the trade-off between finding a good fit and keeping the value of certain feature weights low as the exponents on features increase.

Regularization coefficients L1 and L2 help fight overfitting by making certain weights smaller. Smaller-valued weights lead to simpler hypotheses, and simpler hypotheses are the most generalizable. Unregularized weights with several higher-order polynomials in the feature set tend to overfit the training set.

As the input training set size grows, the effect of regularization decreases and the parameters tend to increase in magnitude. This is appropriate, because an excess of features relative to training set examples leads to overfitting in the first place. Bigger data is the ultimate regularizer.

Sparsity

The sparsity hyperparameter recognizes that for some inputs only a few features are relevant. For example, let's assume that a network can classify a million images. Any one of those images will be indicated by a limited number of features. But to effectively classify millions of images a network must be able to recognize considerably more features, many of which don't appear most of the time. An example of this would be how photos of sea urchins don't contain noses and hooves. This contrasts to how in submarine images the nose and hoof features will be 0.

The features that indicate sea urchins will be few and far between, in the vastness of the neural network's layers. That's a problem, because sparse features can limit the number of nodes that activate and impede a network's ability to learn. In response to sparsity, biases force neurons to activate and the activations stay around a mean that keeps the network from becoming stuck.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

UNIT - III CONVOLUTIONAL NEURAL NETWORK (CNN) (Page No. 57-117)

1.About CNN Pg. 62

- Definition of CNN
- Application of CNN
- Biological Inspiration
- CNNs & Structure in Data
- CNN Architecture Overview
  - What is Convolution?

2.Linear Time Invariant Pg. 64

- About Invariance
- Definition of Linear Time-Invariant Systems (LTI Systems)
- Properties of LTI Systems
- Convolutional Neural Networks: LTI Interpretation
  - LTIs inside CNNs

3.Image Processing Filtering Pg. 67

- Image Processing
- Role of CNN in Image Processing
- What is a Digital Image?
- Image Gradient & Convolution
- Image Processing Filers
  - Edge Detection Filters
  - Denoising Filters
    - Gaussian Filters
    - Median Filters
    - Non-Local Means

4.Building a Convolutional Neural Network (CNN) Pg. 71

- What is Convolutional Neural Network?
- Why do we need Convolutional Neural Network?
- How CNNs work?
- Building a CNN
  - Step1 – Import Required Libraries
  - Step1 – Import Required Libraries
    - Filters
    - Kernel_size

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

- Feature Map AKA Activation Map
- A typical CNN has the following sequence of CNN Layers

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

- The Architecture of the Model

17.AlexNet Pg.94

- What is AlexNet?
- The Architecture of AlexNet

18.VGG 16 Pg.35

- What is VGG16?
- The Architecture of VGG16

19.ResNet Pg.96

- What is ResNet?
- Residual Blocks
- Architecture of ResNet

Summary of some of the more popular Architectures of CNNs Pg.98

- LeNet – 5 (1998)
- AlexNet (2012)
- ZF Net (2013)
- GoogleNet / Inception (2014)
- VGGNet (2014)
- ResNet (2015)

20.Transfer Learning with Image Data Pg.99

- What is Transfer Learning?
- When to use Transfer Learning?
- How to implement Transfer Learning?
  - 1. Obtain the pre-trained model
  - 2. Create a base model
  - 3. Freeze layers so they don't change during training
  - 4. Add new trainable layers
  - 5. Train the new layers on the dataset
  - 6. Improve the model via fine-tuning

21.Transfer Learning using Inception Oxford VGG Model Pg.102

- The Oxford VGG Models

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

- Load the VGG Model in Keras
- Develop a Simple Photo Classifier
  - 1. Get a Sample Image
  - 2. Load the VGG Model
  - 3. Load and Prepare Image
  - 4. Make a Prediction
  - 5. Interpret Prediction
- Complete Example

22.Google Inception Model Pg. 106

- Inception
- Inception V1
- What makes the Inception V3 model better?

23.Microsoft ResNet Model Pg. 107

24.R-CNN Pg. 108

- Object Detection
- Need for R-CNN
- R-CNN
- Problems with R-CNN

25.Fast R-CNN Pg. 110

26.Faster R-CNN Pg. 111

Comparing R-CNN, Fast R-CNN and Faster R-CNN Pg. 113

27.Mask -RCNN Pg. 114

- What is Mask R-CNN?
- Semantic Segmentation
- Instance Segmentation
- How does Mask R-CNN work?
- Advantages of Mask R-CNN

28.YOLO Pg. 116

- What is YOLO?
- Biggest Advantages

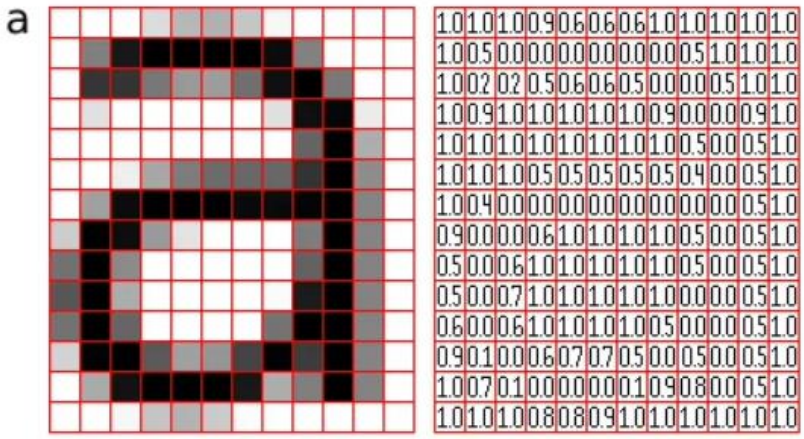
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

1.About CNN

Definition of CNN

Convolutional neural networks also known as *CNN* or *ConvNet* are a specialized type of artificial neural networks that use a mathematical operation called convolution in place of general matrix multiplication in at least one of their layers.

They are specifically designed to process pixel data and are used in image recognition and processing. CNN specializes in processing data that has a grid-like topology, such as an image. A digital image is a binary representation of visual data. It contains a series of pixels arranged in a grid-like fashion that contains pixel values to denote how bright and what colour each pixel should be.



Representation of image as a grid of pixels

Application of CNN

The *goal of a CNN* is to learn higher-order features in the data via convolutions. They are well suited to object recognition with images and consistently top image classification competitions. They can identify faces, individuals, street signs, platypuses, and many other aspects of visual data. CNNs overlap with text analysis via optical character recognition, but they are also useful when analysing words as discrete textual units. They're also good at analysing sound. The efficacy of CNNs in image recognition is one of the main reasons why the world recognizes the power of deep learning. CNNs are powering major advances in machine vision, which has obvious applications for self-driving cars, robotics, drones, and treatments for the visually impaired. CNNs also have been used in other tasks such as natural language translation/generation and sentiment analysis. A convolution is a powerful concept for helping to build a more robust feature space based on a signal.

Biological Inspiration

The human brain processes a huge amount of information the second we see an image. Each neuron works in its own receptive field and is connected to other neurons in a way that they cover the entire visual field. Just as each neuron responds to stimuli only in the restricted region of the visual field called the receptive field in the biological vision system, each neuron in a CNN processes data only in its receptive field as well. The layers are arranged in such a way so that they detect simpler patterns first (lines, curves, etc.) and more complex patterns (faces, objects, etc.) further along. By using a CNN, one can enable sight to computers.

CNNs and Structure in Data

CNNs tend to be most useful when there is some structure to the input data. An example would be how images and audio data that have a specific set of repeating patterns and input values next to each

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

other are related spatially. Conversely, the columnar data exported from a relational database management system (RDBMS) tends to have no structural relationships spatially. Columns next to one another just happen to be materialized that way in the database exported materialized view.

The structure of image data allows us to change the architecture of a neural network in a way that we can take advantage of this structure. With CNNs, we can arrange the neurons in a three-dimensional structure for which we have the following:

- Width
- Height
- Depth

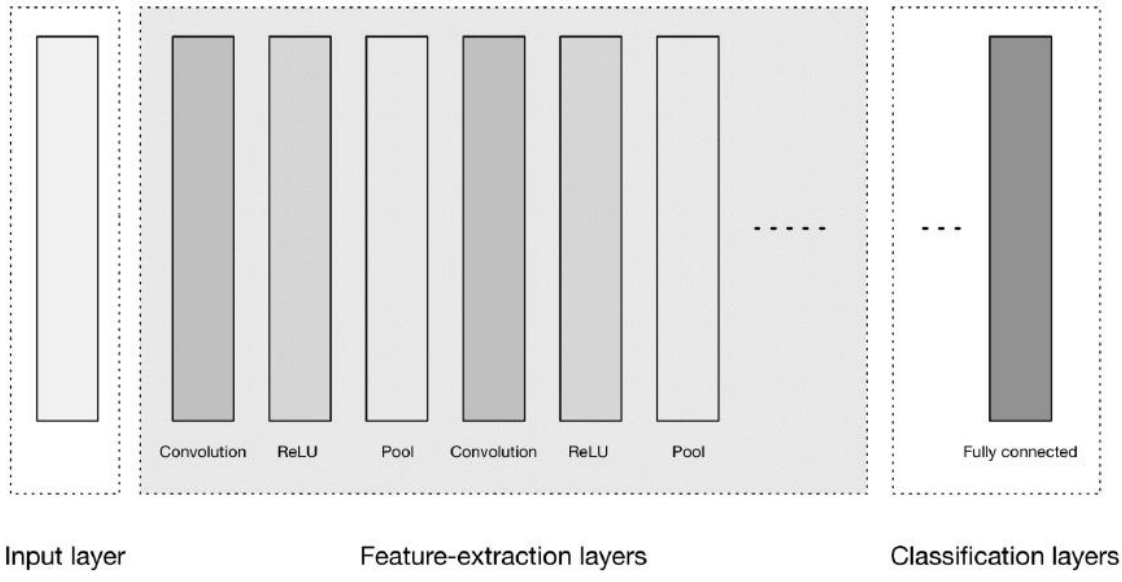
These attributes of the input match up to an image structure for which we have:

- Image width in pixels
- Image height in pixels
- RGB channels as the depth

We can consider this structure to be a three-dimensional volume of neurons. A significant aspect to how CNNs evolved from previous feed-forward variants is how they achieved computational efficiency with new layer types.

CNN Architecture Overview

CNNs transform the input data from the input layer through all connected layers into a set of class scores given by the output layer. There are many variations of the CNN architecture, but they are based on the pattern of layers, as demonstrated below.



High-level general CNN architecture

The CNN architecture depicts three major groups:

1. Input layer
2. Feature-extraction (learning) layers
3. Classification layers

The input layer accepts three-dimensional input generally in the form spatially of the size (width × height) of the image and has a depth representing the colour channels (generally three for RGB colour channels).

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

The feature-extraction layers have a general repeating pattern of the sequence:
1. Convolution layer -We express the Rectified Linear Unit (ReLU) activation function as a layer in the diagram.
2. Pooling layer

These layers find a number of features in the images and progressively construct higher-order features. This corresponds directly to the ongoing theme in deep learning by which features are automatically learned as opposed to traditionally hand engineered.

What is Convolution?

Mathematically, convolution is the summation of the element-wise product of 2 matrices. Let us consider an image 'X' & a filter 'Y' (More about filter will be covered later). Both of them, i.e. X & Y, are matrices (image X is being expressed in the state of pixels). When we convolve the image 'X' using filter 'Y', we produce the output in a matrix, say' Z'.

1	2	3
2	0	0
7	9	1

*

3	2	0
3	0	1
0	5	2

=

1*3 = 3	2*2 = 4	3*0 = 0
2*3 = 6	0*0 = 0	0*1 = 0
7*0 = 0	9*5 = 45	1*2 = 2

Finally, we compute the sum of all the elements in 'Z' to get a scalar number, i.e., 3+4+0+6+0+0+0+45+2 = 60

Finally, we have the classification layers in which we have one or more fully connected layers to take the higher-order features and produce class probabilities or scores. These layers are fully connected to all of the neurons in the previous layer, as their name implies. The output of these layers produces typically a two-dimensional output of the dimensions [b × N], where b is the number of examples in the mini-batch and N is the number of classes we're interested in scoring.

Note:

The *mini-batch* is a fixed number of training examples that is less than the actual dataset. So, in each iteration, we train the network on a different group of samples until all samples of the dataset are used.

2.Linear Time Invariant

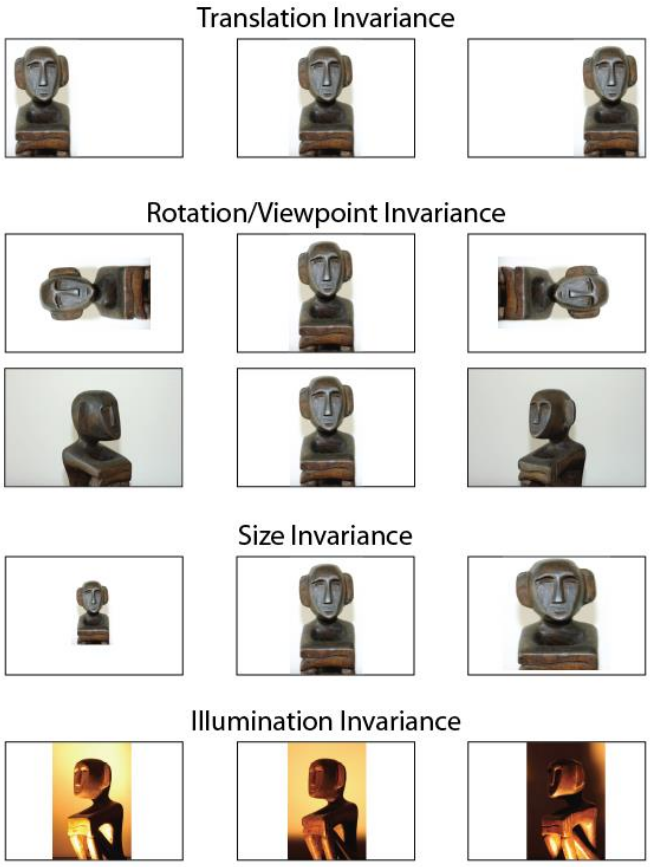
About Invariance

Invariance means that you can recognize an object as an object, even when its appearance varies in some way. This is generally a good thing, because it preserves the object's identity, category, (etc) across changes in the specifics of the visual input, like relative positions of the viewer/camera and the object.

The image below contains many views of the same statue. You (and well-trained neural networks) can recognize that the same object appears in every picture, even though the actual pixel values are quite different.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)



Definition of Linear Time-Invariant Systems (LTI Systems)

Linear Time-Invariant Systems (LTI Systems) are a class of systems used in signals and systems that are both linear and time-invariant. Linear systems are systems whose outputs for a linear combination of inputs are the same as a linear combination of individual responses to those inputs. Time-invariant systems are systems where the output does not depend on when an input was applied. These properties make LTI systems easy to represent and understand graphically.

LTI systems are superior to simple state machines for representation because they have more memory. LTI systems, unlike state machines, have a memory of past states and have the ability to predict the future. LTI systems are used to predict long-term behaviour in a system. So, they are often used to model systems like power plants. Another important application of LTI systems is electrical circuits. These circuits, made up of inductors, transistors, and resistors, are the basis upon which modern technology is built.

Properties of LTI Systems

LTI systems are those that are both linear and time-invariant.

- Linear systems have the property that the *output is linearly related to the input*. Changing the input in a linear way will change the output in the same linear way. So, if the input $x_1(t)$ produces the output $y_1(t)$ and the input $x_2(t)$ produces the output $y_2(t)$, then linear combinations of those inputs will produce linear combinations of those outputs. The input $(x_1(t) + x_2(t))$ will produce the output $(y_1(t) + y_2(t))$.
- Time-invariant systems are systems where the output for a particular input does not change depending on when that input was applied. A time-invariant systems that takes in signal x(t) and

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

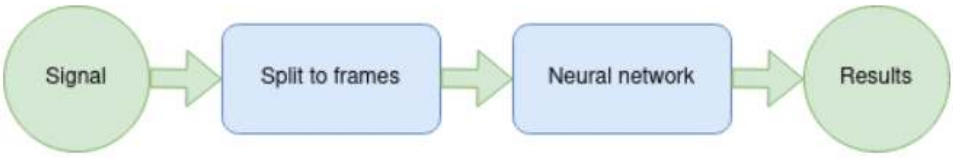
produces output y(t), will also, when excited by signal x(t+σ), produce the time-shifted output y(t+σ).

Convolutional Neural Networks: LTI Interpretation

Convolutional Neural Networks (CNN) have improved results especially in the pattern recognition tasks for images. As the name implies, convolution is playing a central role in their inner workings. Let us apply well known insights from Linear Time-Invariant (LTI) Systems and see how they might help us to understand (and maybe improve) our CNNs. We will concentrate in 1 dimensional signal, since our concepts are most easily understandable with them. The principles however apply to other signals (e.g., images) as well.

Traditional machine learning pipeline for audio

One big thing that CNNs have provided us, is the possibility to directly provide "raw" signal for the learning system. Before them, we wouldn't get decent results without some amount of feature engineering. The convolutional layers embed the feature extraction to happen inside the learning process, and the features themselves are learned from the training data. It is however important to understand that CNNs are not a replacement for feature engineering as whole. Feature extraction (and other pre-processing like resampling and pre-emphasis) beforehand makes it possible to benefit from a-priori knowledge of the problem and reduce the computational burden.



CNN is able to handle the original audio frames as its input

LTIs inside CNNs

It turns out that many operations on signals are well described by LTI systems. Further, it turns out that the operation of any LTI system on an input signal is equivalent to a convolution of the input signal with a special signal associated with the system, denoted the impulse response.

A *convolution* is an operation on two vectors, matrices, or tensors, that returns a third vector, matrix, or tensor. Convolutions arise naturally when considering linear time invariant systems because the operation of an LTI system on any signal is simply given via the convolution of the signal with the system's impulse response.

In signal processing, filters are specified by means of LTI systems or convolutions and are often hand-crafted. One may specify filters for smoothing, edge detection, frequency reshaping, and similar operations. However, with neural networks the idea is to automatically learn the filters and use many of them in conjunction with non-linear operations (activation functions). The system can be further classified based on the "length" of their impulse response: if it is finite, the system is called FIR filter; if impulse response is infinite, the system is called IIR filter.

Understanding the basics of LTI systems can help when designing the structure for CNN model, and well-known analysis methods like Fourier transform can help one to gain insights from inside the "black box".

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

3.Image Processing Filtering

Image Processing

Image processing is a method to perform operations on an image to extract information from it or enhance it. There are *two main types* of image processing: Image Filtering and Image Warping. *Image Filtering* changes the range (i.e., the pixel values) of an image, so the colours of the image are altered without changing the pixel positions, while *Image Warping* changes the domain (i.e., the pixel positions) of an image, where points are mapped to other points without changing the colours.

The goal of using filters is to modify or enhance image properties and/or to extract valuable information from the pictures such as edges, corners, and blobs.

Role of CNN in Image Processing

In Deep Learning, a Convolutional Neural Network (CNN) is a special type of neural network that is designed to process data through multiple layers of arrays. A CNN is well suited for applications like image recognition, and in particular is often used in face recognition software.

A major breakthrough came in the field of Image Processing with the advancement of Deep Learning. Artificial Intelligence (AI) techniques with complex algorithms helps to implement the face recognition, object detection and recognition, finds its use in image and video analytics, medical scan analysis, Robot and drone vision etc.

In CNN, convolutional layers are the fundamental building blocks which make all the magic happens. In a typical image recognition application, a convolutional layer is made up of several filters to detect the various features of an image. Understanding how this work is best illustrated with an analogy.

Suppose you saw someone walking towards you from a distance. From afar, your eyes will try to detect the edges of the figure, and you try to differentiate that figure from other objects, such as buildings, or cars, etc. As the person walks closer towards you, you try to focus on the shape of the person, trying to deduce if the person is male or female, slim or fat, etc. As the person gets nearer, your focus shifted towards other features of that person, such as his facial features, if his is wearing specs, etc. In general, your focus shifted from broad features to specific features.

Likewise, in a CNN, you have several layers containing various filters (or kernels as they are commonly called) in charge of detecting specific features of the target you are trying to detect. The early layer tries to focus on broad features, while the latter layers try to detect very specific features.

In a CNN, the values for the various filters in each convolutional layer is obtained by training on a particular training set. At the end of the training, you would have a unique set of filter values that are used for detecting specific features in the dataset. Using this set of filter values, you would apply them on new images so that you can make a prediction on what is contained within the image. The combination of various filters in each convolutional layer is what makes the prediction possible in a CNN.

What is a Digital Image?

A Digital Image can be defined as a group of numbers in a matrix form. If an image is of 500*500 size means there are 500 number in ‘x’ and 500 numbers in ‘y’ co-ordinate. For example:

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

11	15	22	25	30
200	118	234	88	10
65	9	54	66	43
42	80	23	37	87
78	98	84	44	32

Digital Image in a matrix form

These numbers are also referred as a pixel and each of these numbers represents the intensity of that pixel. This intensity can range for an 8-bit image from 0–255 where 0 being black pixel and 255 being a white. By changing the numbers of the image, we can alter its properties. For example, whenever we are adjusting the contrast, brightness, blurring it or any other property of an image we basically make changes in these numbers altogether. One of the wonders of image processing is being able to see the pixel values of each image and perform different mathematical operations on it. Since the image is already an array or a matrix, then we can already perform different operations like multiplying, adding, subtracting, and dividing.

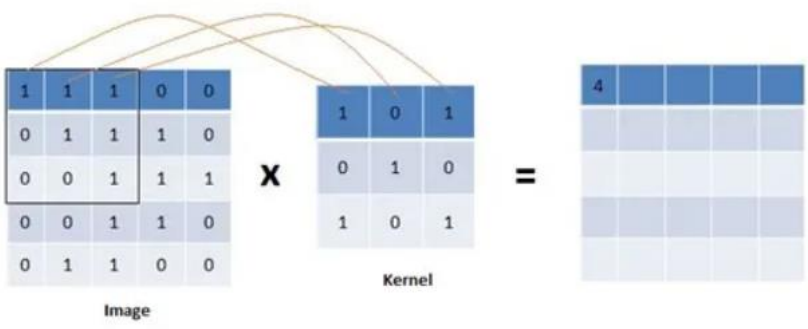
Object Detection

In an image the captured object has properties, it would have shape, colour etc and in digital image the information is collected on the basis of patterns of pixels and change in the intensity of the pixels. Object detection is the process of finding instances of objects in images. In the case of deep learning, object detection is a subset of object recognition, where the object is not only identified but also located in an image. This allows for multiple objects to be identified and located within the same image.

Image Gradient & Convolution

One of the most used operations in image processing is convolution, which is primarily used for filtering the image. The process usually stems from convolving an image array into a user-defined array or matrix. This user-defined matrix is known as kernels and are usually have a different effect depending on the value, they have especially the sign orientation.

Gradient literally means a 'slope'. In an image a directional change in the intensity or pixel values is termed as 'Image Gradient' which forms the base of image processing and usually computed by convolution of the image with a kernel.



Process of Convolution

Here 'Convolution' is nothing but a matrix multiplication. A 'Kernel' is a filter or a matrix with fixed size and is used for extracting the various properties or features of an image by its multiplication with that image.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

The process of convolution yields derivatives in x and y direction by which the magnitude and direction of the gradient is calculated. Each block in the image represents one stride. Once the matrix mapping is done for all the blocks as above, Kernel moves one stride to the right and same process gets repeated.

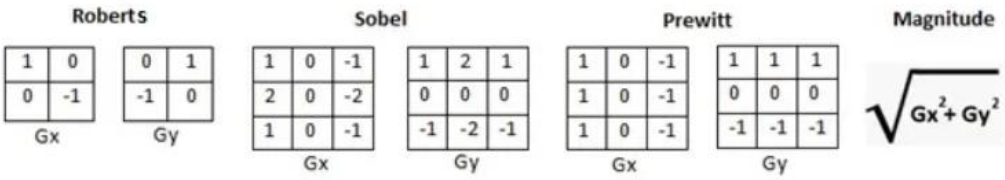
CNNs and computer vision

Image Processing Filters

Image filtering is the first step in any Deep Learning or Object Detection algorithms. In actual deep learning process the selection of filters is done by system itself and there would be layers of filters created.

- **Edge Detection Filters**

Shape is an important property to identify an object distinctively in an image. But what could it mean when digital images are considered? For a machine to understand what is 'a shape' concept of 'edge' is important. As we have understood that they are nothing but arrays of numbers, the edge could be identified as a sharp change in colour or pixel and to identify it, we simply subtract the values on either side of the pixel from various filters such as like Roberts, Sobel and Prewitt



Here, Roberts has two operators, both are rotated by 90 degrees. These operators are designed to run the edges on 45 degrees in the pixel grid. Sobel and Prewitt can be considered as an extension of the Roberts itself. The root of the summation of derivatives in both the directions is used to calculate the magnitude.

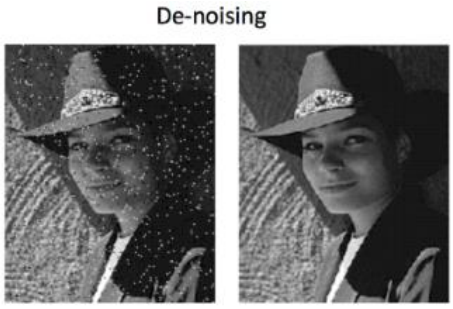
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

As it can be observed there is a very slight difference in all the three images. But these are the edge filter operations applied on a smooth image. Highly smooth image or an image high in noise makes edge preservation difficult.

- **Denoising Filters**

While capturing an image, unwanted features get recorded due to several factors such as lenses quality, temperature or exposure to light etc. These features are termed as '*Noise*'. It is random variation of brightness or colour information in images. *Denoising* is the process of removal of these noises.

This objective is achieved through the convolution of filters. If the edge detection is applied on the noisy image, the edges are not preserved well.



❖ **Gaussian Filters**

Gaussian filtering is a smoothening operation determined by 'Gaussian Function'. The process is often referred as 'Gaussian Blur'. A Gaussian function which is popularly known for normal distribution or Bell-shaped curve and represents the probability that events are centered around the mean value. It is a linear filter and 2-D convolution operator and used as a 'Point Spread'.

❖ **Median Filters**

The median filter is a non-linear digital filtering technique. Median in a data is a value which is in the middle. The median filter is a non-linear digital filtering technique. The median is calculated by first sorting all the pixel values from into numerical order, and then replacing the distinct pixel with the middle pixel value. When the noise is such there is an inconsistency. Most of the pixels have the same value but few have distinctively different value, in such a case, median filters can be used.

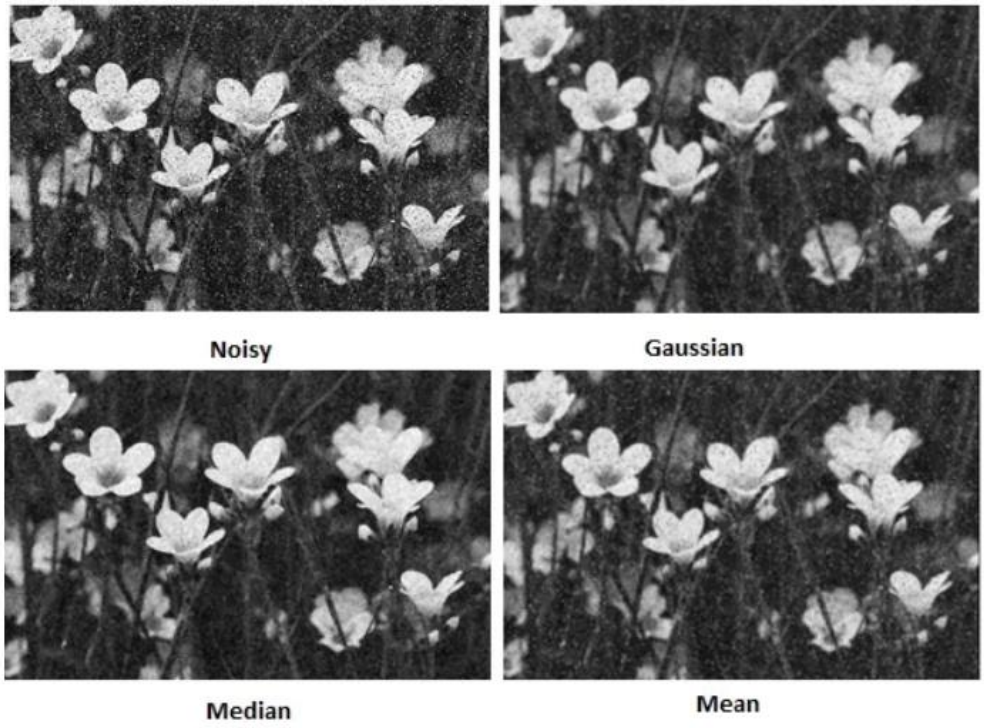
❖ **Non-Local Means**

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

'Non-Locals' means- it is not just taking a local mean with in a specific grid size but looking for it in the other region as well. "Estimated value is the weighted average of all pixel in the image but the family of weights depends on the similarity between the pixels i and j".

In other words, similar pixel in neighbourhood is given the larger weights. Which means that the entire picture has some darker and lighter regions spread all over the image. As they are varying on the range of darkness, they must be having similarity in pixels as compare to the other lighter regions. So, all these pictures would be assigned the lager weights, that is algorithm is calculating where the similar pixels are and assigning the weight accordingly.



Here we can see that 'Median' filter seems to be performing better as expected. Because the noise we introduced artificially was 'Salt and Pepper' which is caused by sharp and sudden disturbances hence displays usual peaks in pixel values which are easy to be normalized by a 'Median' value. Again, it also depends upon the sigma values. One need to choose the optimal sigma value by 'Try and Error' method. Apart from these there are many filters such as 'Canny' which performs edge preservation and smoothing together.

4.Building a Convolutional Neural Network

What is Convolutional Neural Network?

Convolutional Neural Network is a specialized neural network designed for visual data, such as images & videos. But CNNs also work well for non-image data (especially in NLP & text classification). Its concept is similar to that of a vanilla neural network (multilayer perceptron) – It follows the same general principle of forwarding & backward propagation.

Why do we need Convolutional Neural Network?

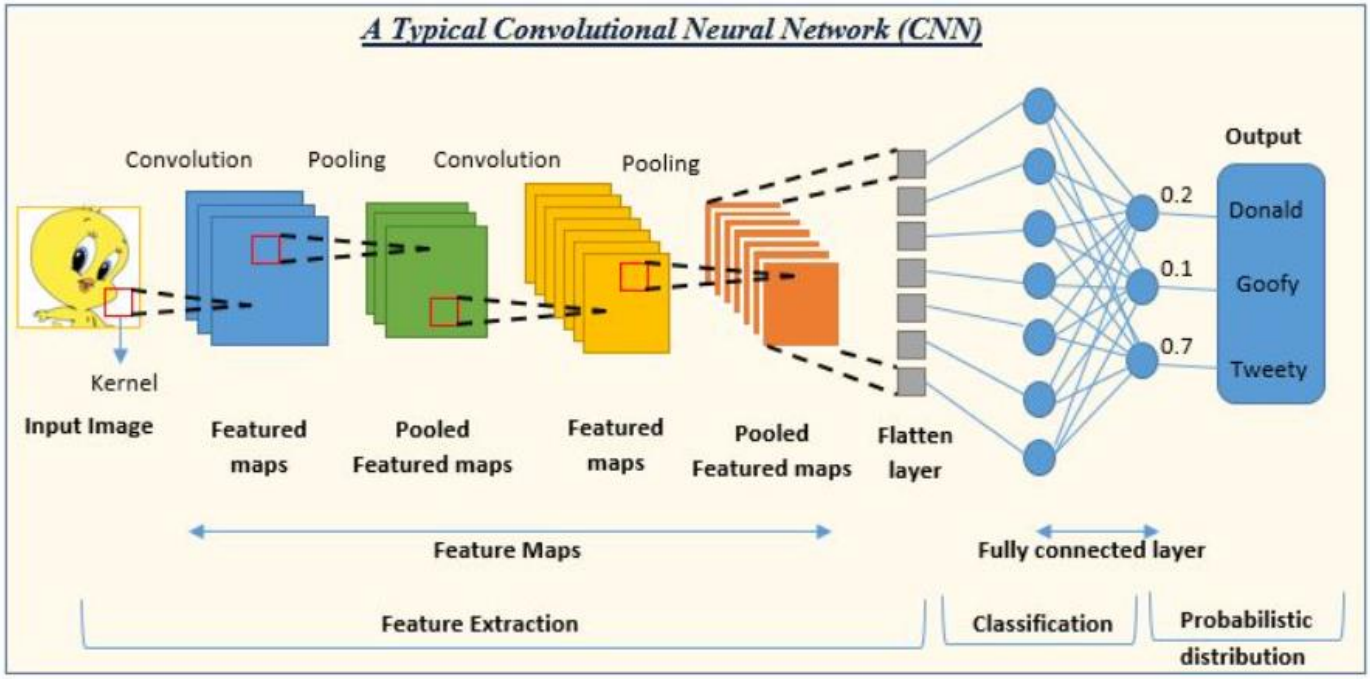
- Although vanilla neural networks (MLPs) can learn highly complex functions, their architecture does not exploit what we know about how the brain reads & processes images.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

- The architecture of Convolutional Neural Network uses many of the working principles of the animal visual system & it has been able to achieve extraordinary results in image-related learning tasks.
- For this reason, MLPs haven't been able to achieve any significant breakthroughs in the image processing domain.

How CNNs Work?

A Convolutional Neural Network can have tens or hundreds of layers that each learn to detect different features of an image. Filters are applied to each training image at different resolutions, and the output of each convolved image is used as the input to the next layer. The filters can start as very simple features, such as brightness and edges, and increase in complexity to features that uniquely define the object.



A CNN typically has ***three layers***: a convolutional layer, a pooling layer, and a fully connected layer.

1) Convolutional Layer: In a typical neural network each input neuron is connected to the next hidden layer. In CNN, only a small region of the input layer neurons connects to the neuron hidden layer.

2) Pooling Layer: The pooling layer is used to reduce the dimensionality of the feature map. There will be multiple activation & pooling layers inside the hidden layer of the CNN.

3) Fully-Connected layer: Fully Connected Layers form the last few layers in the network. The input to the fully connected layer is the output from the final Pooling or Convolutional Layer, which is flattened and then fed into the fully connected layer.

This key characteristic gives convnets ***two interesting properties***:

- *The patterns they learn are translation invariant*. After learning a certain pattern in the lower-right corner of a picture, a convnet can recognize it anywhere: for example, in the upper-left corner. A densely connected network would have to learn the pattern anew if it appeared at a new location. This makes convnets data efficient when processing images (because the visual

This document is available free of charge on studocu
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

world is fundamentally translation invariant): they need fewer training samples to learn representations that have generalization power.

- *They can learn spatial hierarchies of patterns*. (see figure 5.2) A first convolution layer will learn small local patterns such as edges, a second convolution layer will learn larger patterns made of the features of the first layers, and so on. This allows convnets to efficiently learn increasingly complex and abstract visual concepts (because the visual world is fundamentally spatially hierarchical).

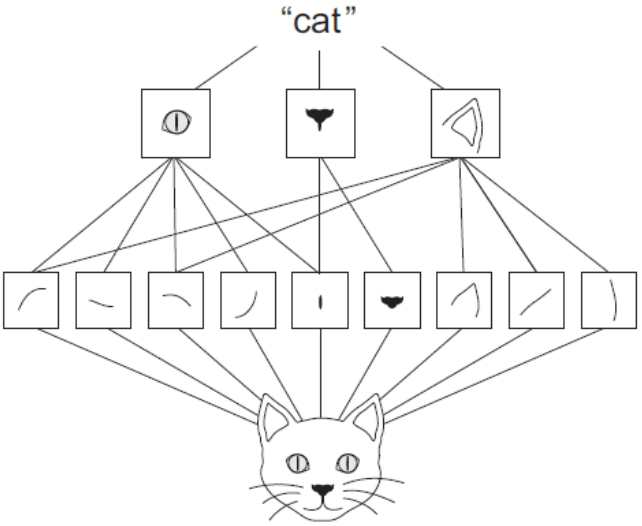


Figure 5.2 The visual world forms a spatial hierarchy of visual modules: hyperlocal edges combine into local objects such as eyes or ears, which combine into high-level concepts such as "cat."

Building a CNN

Let's discuss the building of CNN using the Keras library along with an explanation of the working of CNN in the following 6 steps.

- **Step1 – Import Required Libraries**

- **Step2 – Initializing CNN & Add a Convolutional Layer**

  We first need to initiate sequential class since there are various layers to build CNN which all must be in sequence. Then we add the first convolutional layer where we need to specify 5 arguments. So, let's discuss each argument and its purpose.

  - **Filters -** The primary purpose of convolution is to find features in the image using a feature detector. Then put them into a feature map, which preserves distinct features of images. Feature detector which is known as a filter also is initialized randomly and then after a lot of iteration, filter matrix parameter selected which will be best for separating images. For instance, animals' eye, nose, etc. will be considered as a feature which is used for classifying images using filter or feature detectors. Here we are using 16 features.

  - **Kernel_size -** Kernel_size refers to filter matrix size. Here we are using a 2*2 filter size.

  - **Padding –**
    Let's discuss what is problem with CNN and how the padding operation will solve the problem.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

- Every time we apply a convolution operator, our image shrinks (in the above example, our vision shrunk from 6 x 6 to 4 x 4). If we convolve the output again with a filter, our image shrinks. If we continue this process, we lose a lot of information because of image shrinking, which is one of the downsides of convolution.
- During convolution, the pixels in the corners & the edges are considered only once. This is the 2nd downside of convolution. If we consider any pixel in the middle, many (f x f) regions overlap the pixel (we shift the filter & observe the image through it, i.e., convolve). Thus, the pixels on the corners or the edges are used much less in the output. So, we throw away a lot of information near the edge of the image. In short, Pixels, located on corners are contributed very little compared to middle pixels.

So, then to mitigate these problems, padding operation is done. *Padding* is a simple process of adding layers with 0 or -1 to input images so to avoid above mentioned problems. Here we are using Padding = Same arguments, which depicts that output images have the same dimensions as input images.

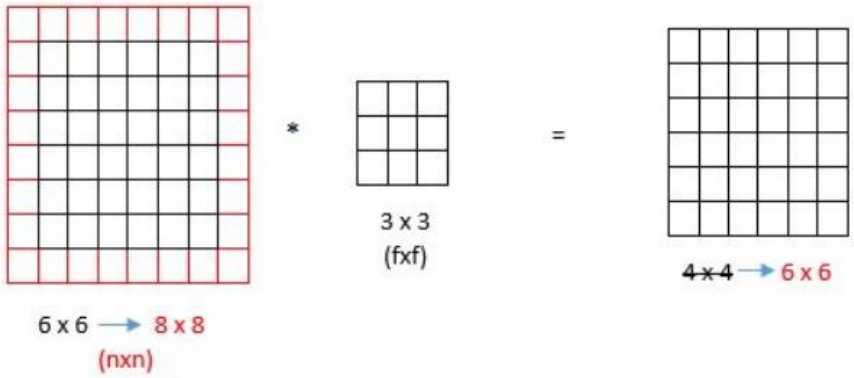


Fig: Padding the input image

Let P be padding. In this example, p = 1 because we padded all around the input image with an extra border of 1 pixel.

∴ Output Size = (n + 2p –f +1) x (n + 2p –f +1)

= (6 + 2×1 – 3 +1) x (6 + 2×1 – 3 +1)

= 6 x 6

To avoid shrinkage of the original input image, we calculate 'p = padding size'. The output image size achieved after convolving the padded input image is equal to that of the original input image size.

∴ Output size after convolving padded image = Original input image size

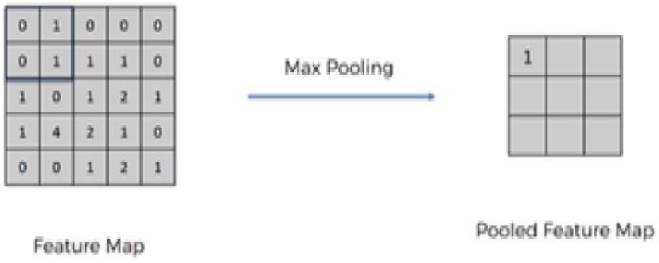
- **Activation Function – ReLU -** Since images are non-linear, to bring non-linearity, the ReLU activation function is applied after the convolutional operation. ReLU stands for Rectified Linear Unit Activation Function. ReLU function will output the input directly if it is positive, otherwise, it will output zero.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

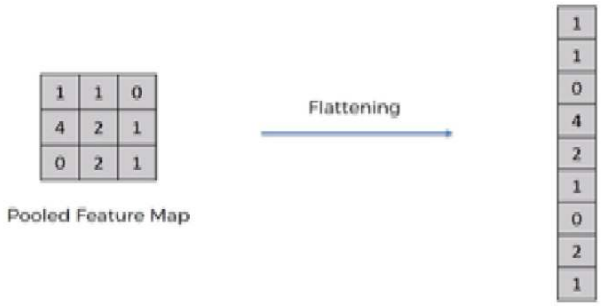
- **Input Shape -** This argument shows image size – 224*224*3. Since the images in RGB format so, the third dimension of the image is 3.

➢ **<u>Step3</u> – Pooling Operation** - We need to apply the pooling operation after initializing CNN. Pooling is an operation of down sampling of the image. The pooling layer is used to reduce the dimensions of the feature maps. Thus, the Pooling layer reduces the number of parameters to learn and reduces computation in the neural network. Future operations are performed on summarized features created by the pooling layer. instead of precisely positioned features generated by the convolution layer. This leads the model more robust to variations in the orientation of the feature in the image.



➢ **<u>Step4</u> – Add Two Convolutional Layers** - In order to add two more convolutional layers, we need to repeat steps 2 &3 with slight modification in the number of filters.

➢ **<u>Step5</u> – Flattening Operation** - Flattening operation is converting the dataset into a 1-D array for input into the next layer which is the fully connected layer.



After finishing the 3 steps, now we have pooled feature map. We are now flattening our output after two steps into a column. Because we need to insert this 1-D data into an artificial neural network layer.

➢ **<u>Step6</u> – Fully Connected Layer & Output Layer -** The output of the flattening operation work as input for the neural network. The aim of the artificial neural network makes the convolutional neural network more advanced and capable enough of classifying images. Here we are using a dense class from the Keras library from creating a fully connected layer and output layer.

The *SoftMax Activation Function* is used for building the Output Layer. It is used as the last activation function of a neural network to bring the output of the neural network to a probability distribution over predicting classes. The output of Softmax is in probabilities of each possible outcome for predicting class. The probabilities sum should be one for all possible predicting classes.

Training and Evaluation of the CNN

➢ **Step 1 – Compile CNN model**

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

```
model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
```

Here we are using 3 Arguments: -

- **Loss Function** - We are using the categorical_crossentropy loss function that is used in the classification task. This loss is a very good measure of how distinguishable two discrete probability distributions are from each other.
- **Optimizer -** We are using Adam Optimizer that is used to update neural network weights and learning rate. Optimizers are used to solve optimization problems by minimizing the function.
- **Metrics Arguments -** Here, we are using Accuracy as a metrics to evaluate the performance of the Convolutional neural network algorithm.

➢ **Step 2 – Fit Model on Training Set**

```
model.fit_generator(training_set,validation_data=test_set,epochs=50,steps_per_epoch=len(trai
ning_set), validation_steps=len(test_set) )
```

We are fitting the CNN model on the training dataset with 50 iterations and each iteration has different steps for training and evaluating steps based on the length of the test and training set.

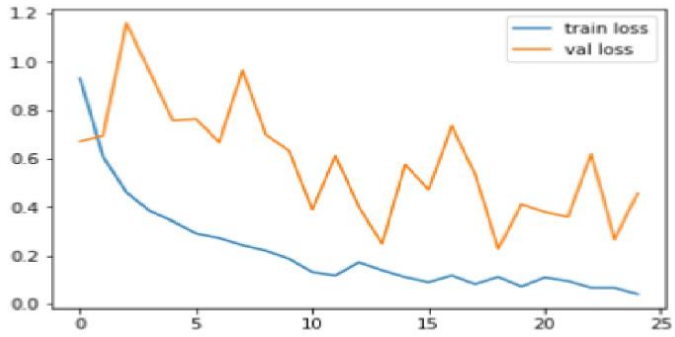
➢ **Step 3 – Evaluate Result**

We compare the accuracy and loss function for both the training and test dataset.

Code: Plotting Loss Graph

```
plt.plot(r.history['loss'], label='train loss')
plt.plot(r.history['val_loss'], label='val loss')
plt.legend()
plt.show()
plt.savefig('LossVal_loss')
```

Output



Loss is the penalty for a bad prediction. The aim is to make the validation loss as low as possible. Some overfitting is nearly always a good thing. All that matters, in the end, is: is the validation loss as low as you can get it.

Code: Plotting Accuracy Graph

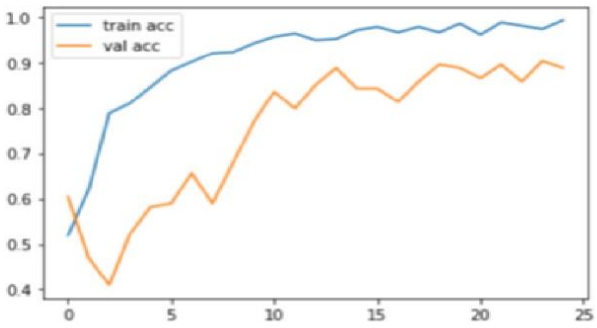
```
plt.plot(r.history['accuracy'], label='train acc')
plt.plot(r.history['val_accuracy'], label='val acc')
```


This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

```
plt.legend()
plt.show()
plt.savefig('AccVal_acc')
```

Output



Accuracy is one metric for evaluating classification models. Informally, accuracy is the fraction of predictions our model got right. Here, we can observe that accuracy inches towards 90% on validating test which depicts a CNN model is performing well on accuracy metrics.

5.Input Layers

Input layers are where we load and store the raw input data of the image for processing in the network. This input data specifies the width, height, and number of channels. Typically, the number of channels is three, for the RGB values for each pixel.

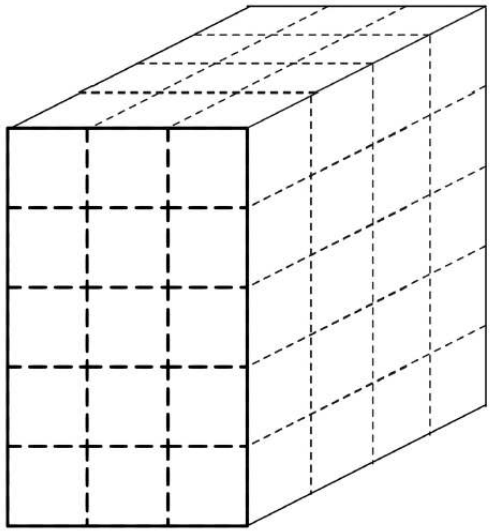
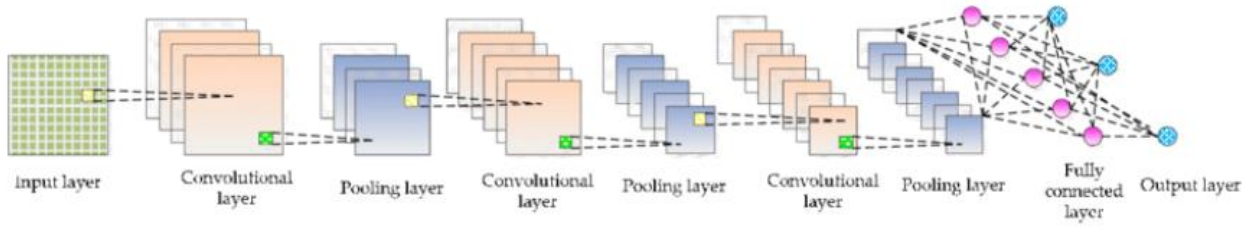


Figure 4-10. Input layer 3D volume

The input layer of a neural network is composed of artificial input neurons, and brings the initial data into the system for further processing by subsequent layers of artificial neurons. The input layer is the very beginning of the workflow for the artificial neural network. Artificial neural networks are typically composed of input layers, hidden layers and output layers. Other components may include convolutional layers and encoding or decoding layers.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)



One of the distinct characteristics of the input layer is that artificial neurons in the input layer have a different role to play – experts explain this as the input layer being constituted of "passive" neurons that do not take in information from previous layers because they are the very first layer of the network. In general, artificial neurons are likely to have a set of weighted inputs and function on the basis of those weighted inputs – however, in theory, an input layer can be composed of artificial neurons that do not have weighted inputs, or where weights are calculated differently, for example, randomly, because the information is coming into the system for the first time. What is common in the neural network model is that the input layer sends the data to subsequent layers, in which the neurons do have weighted inputs.

6.Convolution Layers

Convolutional layers are considered the core building blocks of CNN architectures. As Figure 4-11 illustrates, convolutional layers transform the input data by using a patch of locally connecting neurons from the previous layer. The layer will compute a dot product between the region of the neurons in the input layer and the weights to which they are locally connected in the output layer.

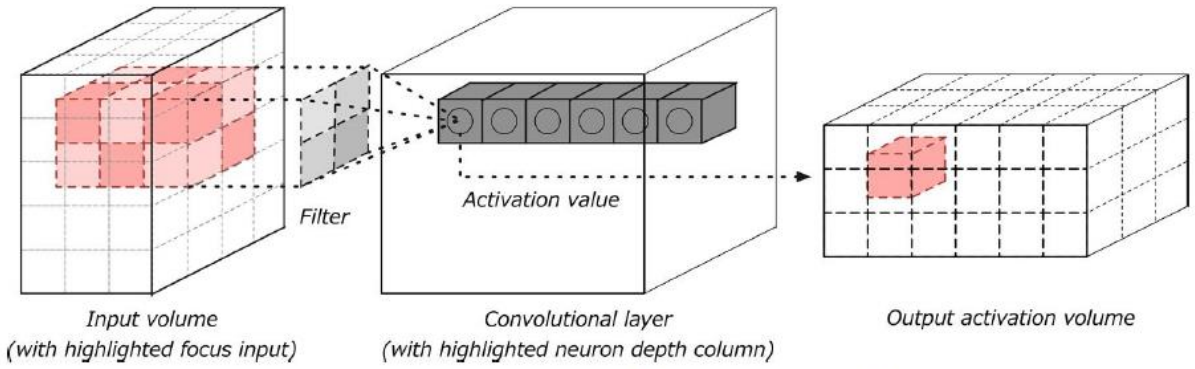


Figure 4-11. Convolution layer with input and output volumes

The resulting output generally has the same spatial dimensions (or smaller spatial dimensions) but sometimes increases the number of elements in the third dimension of the output (depth dimension). Let's take a closer look at a key concept in these layers, called a convolution.

Convolution

A Convolution is defined as a mathematical operation describing a rule for how to merge two sets of information. It is important in both physics and mathematics and defines a bridge between the space/time domain and the frequency domain through the use of Fourier transforms. It takes input, applies a convolution kernel, and gives us a feature map as output.

The convolution operation, shown in Figure 4-12, is known as the feature detector of a CNN. The input to a convolution can be raw data or a feature map output from another convolution. It is often interpreted as a filter in which the kernel filters input data for certain kinds of information; for example, an edge kernel lets pass through only information from the edge of an image.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

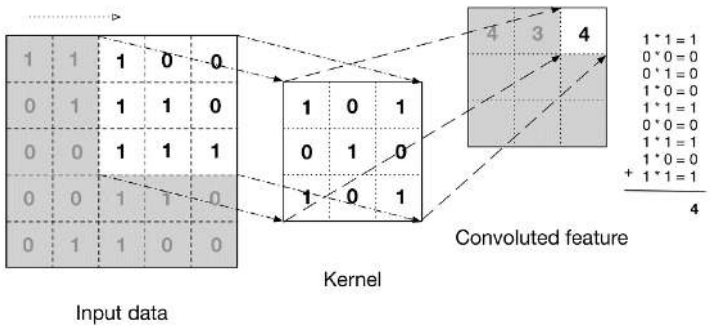


Figure 4-12. The convolution operation

The figure illustrates how the kernel is slid across the input data to produce the convoluted feature (output) data. At each step, the kernel is multiplied by the input data values within its bounds, creating a single entry in the output feature map. In practice the output is large if the feature we're looking for is detected in the input.

We commonly refer to the sets of weights in a convolutional layer as a filter (or kernel). This filter is convolved with the input and the result is a feature map (or activation map). Convolutional layers perform transformations on the input data volume that are a function of the activations in the input volume and the parameters (weights and biases of the neurons). The activation map for each filter is stacked together along the depth dimension to construct the 3D output volume.

Convolutional layers have parameters for the layer and additional hyperparameters. Gradient descent is used to train the parameters in this layer such that the class scores are consistent with the labels in the training set.

Following are the major components of convolutional layers:

- Filters
- Activation Maps
- Parameter Sharing
- Layer-Specific Hyperparameters

7.Pooling Layers

Pooling layers are commonly inserted between successive convolutional layers. We want to follow convolutional layers with pooling layers to progressively reduce the spatial size (width and height) of the data representation. Pooling layers reduce the data representation progressively over the network and help control overfitting. The pooling layer operates independently on every depth slice of the input.

Common Down Sampling Operations

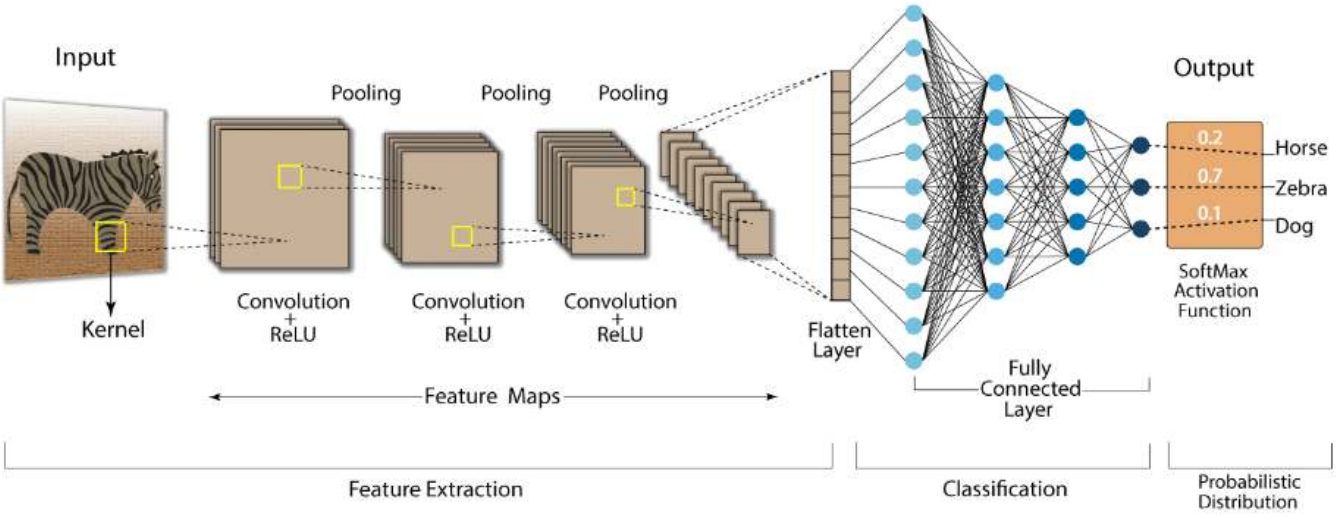
The most common down sampling operation is the max operation. The next most common operation is average pooling.

The pooling layer uses the max() operation to resize the input data spatially (width, height). This operation is referred to as max pooling. With a 2 × 2 filter size, the max() operation is taking the largest of four numbers in the filter area. This operation does not affect the depth dimension.

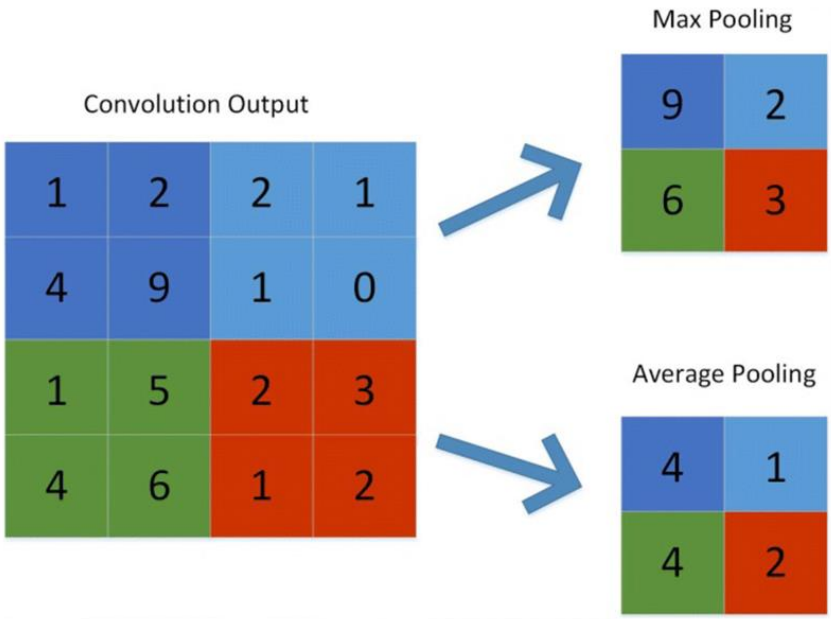
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Pooling layers use filters to perform the down sampling process on the input volume. These layers perform down sampling operations along the spatial dimension of the input data. This means that if the input image were 32 pixels wide by 32 pixels tall, the output image would be smaller in width and height (e.g., 16 pixels wide by 16 pixels tall). The most common setup for a pooling layer is to apply 2 × 2 filters with a stride of 2. This will down sample each depth slice in the input volume by a factor of two on the spatial dimensions (width and height). This down sampling operation will result in 75 percent of the activations being discarded.

Pooling layers do not have parameters for the layer but do have additional hyperparameters. This layer does not involve parameters, because it computes a fixed function of the input volume. It is not common to use zero-padding for pooling layers.



Its purpose is to gradually shrink the representation's spatial size to reduce the number of parameters and computations in the network. The pooling layer treats each feature map separately.



The following are some methods for pooling:

This document is available free of charge on studocu
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Max-Pooling: It chooses the most significant element from the feature map. The feature map's significant features are stored in the resulting max-pooled layer. It is the most popular method since it produces the best outcomes.

Average Pooling: It entails calculating the average for each region of the feature map.

Pooling gradually reduces the spatial dimension of the representation to reduce the number of parameters and computations in the network, as well as to prevent overfitting. If there is no pooling, the output has the same resolution as the input.

8.Dense Layers

A dense layer also referred to as a fully connected layer is a layer that is used in the final stages of the neural network. This layer helps in changing the dimensionality of the output from the preceding layer so that the model can easily define the relationship between the values of the data in which the model is working.

What is a Dense Layer?

In any neural network, a dense layer is a layer that is deeply connected with its preceding layer which means the neurons of the layer are connected to every neuron of its preceding layer. This layer is the most commonly used layer in artificial neural network networks.

The dense layer's neuron in a model receives output from every neuron of its preceding layer, where neurons of the dense layer perform matrix-vector multiplication. Matrix vector multiplication is a procedure where the row vector of the output from the preceding layers is equal to the column vector of the dense layer. The general rule of matrix-vector multiplication is that the row vector must have as many columns like the column vector.

The general formula for a matrix-vector product is:

$$\mathbf{Ax} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix}.$$

Values under the matrix are the trained parameters of the preceding layers and also can be updated by the backpropagation. Backpropagation is the most commonly used algorithm for training the feedforward neural networks. Generally, backpropagation in a neural network computes the gradient of the loss function with respect to the weights of the network for single input or output. From the above intuition, we can say that the output coming from the dense layer will be an N-dimensional

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

vector. We can see that it is reducing the dimension of the vectors. So basically, a dense layer is used for changing the dimension of the vectors by using every neuron.

As discussed before, results from every neuron of the preceding layers go to every single neuron of the dense layer. So, we can say that if the preceding layer outputs a (M x N) matrix by combining results from every neuron, this output goes through the dense layer where the count of neurons in a dense layer should be N.

Basic Operations with Dense Layer

Dense layer does the below operation on the input and return the output.

```
output = activation(dot(input, kernel) + bias)
```

where,

- **input** represent the input data
- **kernel** represent the weight data
- **dot** represent NumPy dot product of all input and its corresponding weights
- **bias** represent a biased value used in machine learning to optimize the model
- **activation** represent the activation function.

Let us consider sample input and weights as below and try to find the result –

- input as 2 x 2 matrix [[1, 2], [3, 4]]
- kernel as 2 x 2 matrix [[0.5, 0.75], [0.25, 0.5]]
- bias value as 0
- activation as linear.

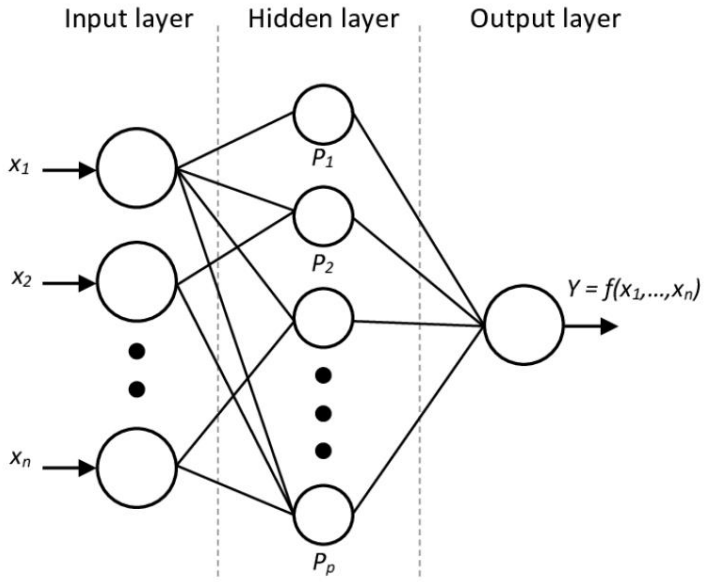
```
>>> import numpy as np

>>> input = [ [1, 2], [3, 4] ]
>>> kernel = [ [0.5, 0.75], [0.25, 0.5] ]
>>> result = np.dot(input, kernel)
>>> result array([[1. , 1.75], [2.5 , 4.25]])
>>>
```

- result is the output and it will be passed into the next layer.

This document is available free of charge on studocu

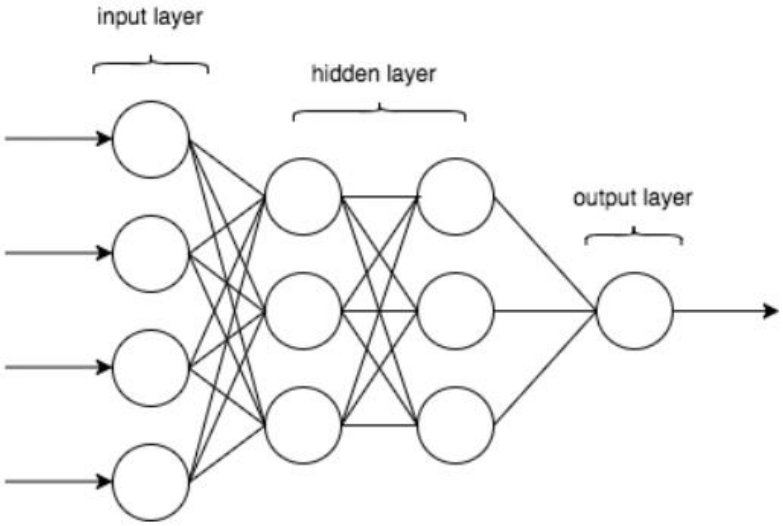
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

The output shape of the Dense layer will be affected by the number of neuron / units specified in the Dense layer. For example, if the input shape is (8,) and number of units is 16, then the output shape is (16,). All layer will have batch size as the first dimension and so, input shape will be represented by (None, 8) and the output shape as (None, 16). Currently, batch size is None as it is not set. Batch size is usually set during training phase.



The above image represents the neural network with one hidden layer. If we consider the hidden layer as the dense layer the image can represent the neural network with a single dense layer.

After defining the input layer once we don't need to define the input layer for every dense layer.



The above image represents the neural network with one hidden layer. If we consider the hidden layer as the dense layer the image can represent the neural network with multiple dense layers.

Note – The Dense layer is an input layer because after calling the layer we cannot change the attributes because as the input shape for the dense layer passes through the dense layer the Keras defines an input layer before the current dense layer.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

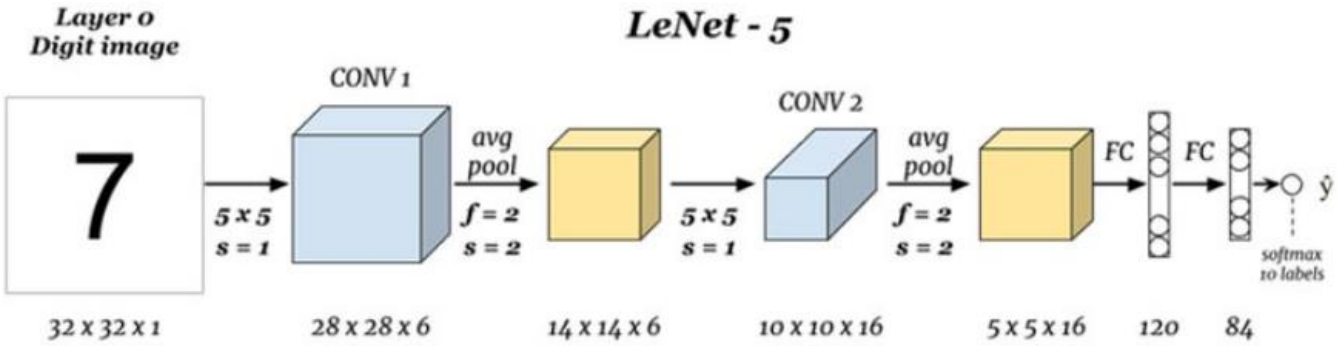
9.Back Propagation through the Convolutional Layers

About Convolutional Layer & Pooling Layer

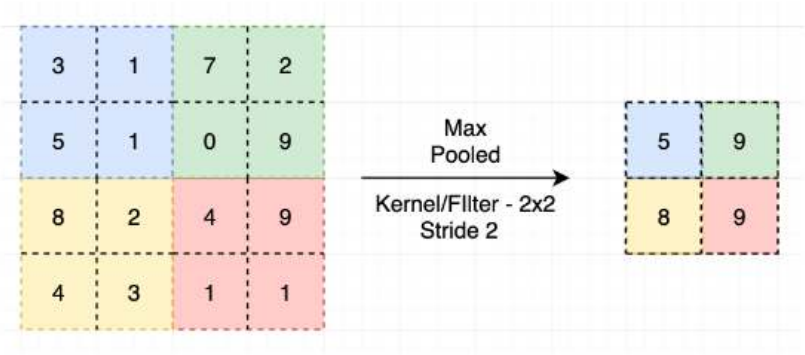
A Convolutional layer in a convolutional neural network represents and maps the features of an input image. The output feature map of each convolutional process is highly sensitive to the location of the feature in the input, and it represents a problem. One way to manage it is to down sample the feature maps, making the resulting down-sampled feature maps more robust.

In this sense, the Pooling layers provide us the possibility to summarize the presence of those features in the input, which means, it reduces the spatial dimension of the input volume for the next layers. Affecting just the weight and height but not depth and there are no learnable parameters in this layer. Two common pooling methods are average pooling and max pooling.

Usually, a pooling layer is a new layer added after the convolutional layer. Specifically, after a nonlinearity (e.g., ReLU) has been applied to the feature maps output by a convolutional layer; for example, the layers in a model may look as follows:
Image => Convolutional Layer => Nonlinearity => Pooling Layer



The pooling layer takes an input volume of size w1×h1×c1 and the two hyperparameters are used: filter and stride, and the output volume is of size is w2xh2xc2 where w2 = (W1−F) / S+1, h2 = (h1−f) / s+1, c1 and c2 are same.



So, we could infer, that when we are doing the forward and backpropagation these layers need to be treated as a different layer.

Backpropagation

Backpropagation is one of the most important phases during the training of neural networks. As a target, it determines the neural network's knowledge to be understood as the ability to respond properly to future urges. The terms "knowledge" and "properly" must consider two aspects: (i) the first is given by experience, or better, by the information (features) that the network learns to extract from

This document is available free of charge on studocu
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

the samples of the dataset (ii) the second by the error calculus or also called loss. In supervised learning, each sample is labelled with a class (classification), with one or more values (regression) or both.

Let y_actual be the label assigned to the sample by the supervisor and y_pred the network prediction, the backpropagation phase will ensure that the distance (error) between the two is acceptable. Both errors and features have weights associated with them, which when modified during backpropagation, constitute the network's learned knowledge.

Backpropagation has *two phases*. The first one is the calculation of the gradient, or better, of the derivative of the loss function to each network's weights. The second is the updating of them (delta rule). This last one can be performed with various optimization algorithms, to name a few Momentum, Adam etc,

10.Filters and Feature Maps

Filters

What are Filters/Kernels?

- A filter provides a measure for how close a patch or a region of the input resembles a feature. A feature may be any prominent aspect – a vertical edge, a horizontal edge, an arch, a diagonal, etc.
- A filter acts as a single template or pattern, which, when convolved across the input, finds similarities between the stored template & different locations/regions in the input image.
- Let us consider an example of detecting a vertical edge in the input image.
- Each column of the 4×4 output matrix looks at exactly three columns & three rows (the coloured boxes show the output of the filter as it moves over the input image). The values in the output matrix represent the change in the intensity along the horizontal direction w.r.t the columns in the input image.
- The output image has the value 0 in the 1st & last column. It means there is no change in intensity in the first three columns & the previous three columns of the input image. On the other hand, the output is 30 in the 2nd & 3rd column, indicating a change in the intensity of the corresponding columns of the input image.

Multiple Filters for Multiple Features

We can use multiple filters to detect various features simultaneously. Let us consider the following example in which we see vertical edge & curve in the input RGB image. We will have to use two different filters for this task, and the output image will thus have two feature maps.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

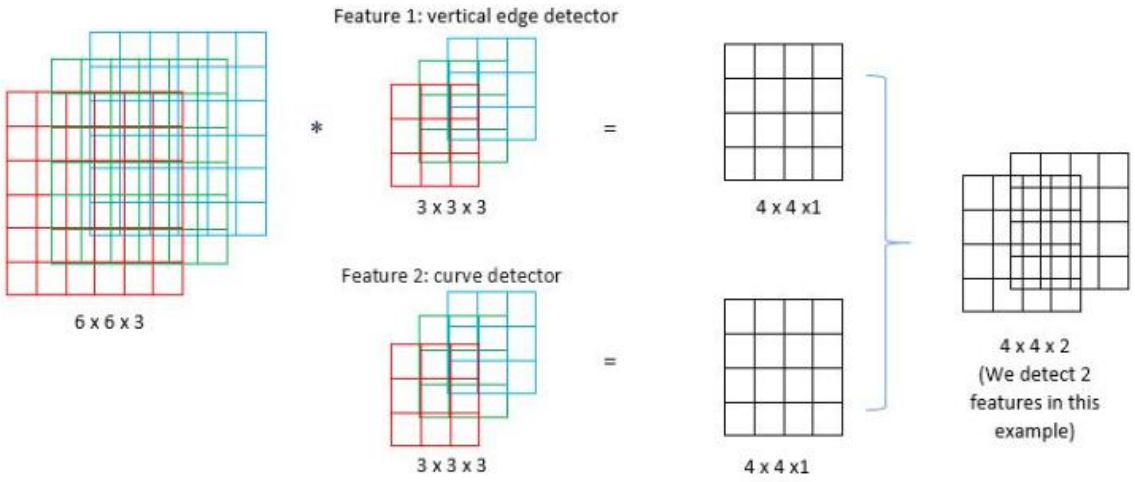


Fig: Convolution using multiple filters

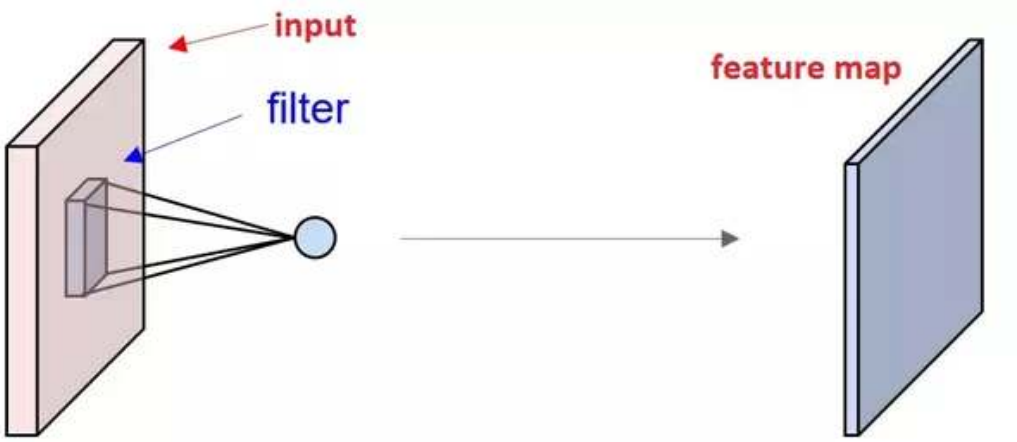
Key Points about Convolution Layers and Filters

- The depth of a filter in a CNN must match the depth of the input image. The number of colour channels in the filter must remain the same as the input image.
- Different Conv2D filters are created for each of the three channels for a colour image.
- Filters for each layer are randomly initialized based on either Normal or Gaussian distribution.
- Initial layers of a convolutional network extract high-level features from the image, so use fewer filters. As we build further deeper layers, we increase the number of filters to twice or thrice the size of the filter of the previous layer.
- Filters of the deeper layers learn more features but are computationally very intensive.
- The filters are learned during training (i.e., during backpropagation). Hence, the individual values of the filters are often called the weights of CNN.
- A neuron is a filter whose weights are learned during training. E.g., a (3,3,3) filter (or neuron) has 27 units. Each neuron looks at a particular region in the output (i.e., its 'receptive field').

Feature Maps

Feature Map AKA Activation Map

A *Feature Map* is a collection of multiple neurons, each looking at different inputs with the same weights. All neurons in a feature map extract the same feature (but from other input regions). It is called a 'feature map' because it maps where a particular part is found in the image.



In Convolutional Neural Networks (CNNs), a feature map is a 3D array that represents the output of a particular convolutional layer. Each element in the feature map corresponds to a single neuron in

This document is available free of charge on studocu
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

the layer, and its value represents the activation level of that neuron. Feature maps are created by applying a set of learned filters (also called kernels or weights) to the input data, which produces a set of convolved output maps. These convolved output maps are then passed through a non-linear activation function, such as the Rectified Linear Unit (ReLU), to introduce non-linearity into the network. The resulting feature maps are then passed to the next layer in the CNN for further processing. Feature maps are a key component of CNNs and are used to extract meaningful features from images or other types of input data.

Feature Map and *Activation Map* mean exactly the same thing. It is called an activation map because it is a mapping that corresponds to the activation of different parts of the image, and also a feature map because it is also a mapping of where a certain kind of feature is found in the image. A high activation means a certain feature was found.

A "*Rectified Feature Map*" is just a feature map that was created using ReLU. The term "feature map" used for the result of the dot products because this is also really a map of where certain features are in the image.

A typical CNN has the following sequence of CNN Layers

- ✓ We have an input image using multiple filters to create various feature maps.
- ✓ Each feature map of size (C, C) is pooled to generate a (C/2, C/2) output (for a standard 2×2 pooling).
- ✓ The above pattern is referred to as one Convolutional Neural Network layer or one unit. Multiple such CNN layers are stacked on top of each other to create deep Convolutional Neural Network networks.
- ✓ The output of the convolution layer contains features, and these features are fed into a dense neural network.

11.Back Propagation through the Pooling Layers

Need for Pooling Layer

It was found that applying the pooling layer after the convolution layer improves performance helping the network to generalize better and reduce overfitting. This is because, given a certain grid (pooling height x pooling width) we sample only one value from it ignoring particular elements and suppressing noise. Moreover, because pooling reduces the spatial dimension of the feature maps coming from the previous layer and it doesn't add any parameters to learn, it helps in decreasing the model complexity, computational costs and results in faster training.

Forward Propagation

We assume that after the Convolution operation we get an output of shape 4x4. We want then to do max pooling with pooling height, pooling width and stride all equal to 2. Pooling is similar to convolution, but instead of doing an element-wise multiplication between the weights and a region in the input and summing them up to get the element for a certain cell in the output matrix, we simply select the maximum element from that region.

The output shape after pooling operation is obtained using the following formula:

```
H_out = floor(1 + (H – pool_height)/stride)
W_out = floor(1 + (W – pool_width)/stride)
```

Where,

H is height of the input, pool_height is height of the pooling region
W is width of the input, pool_width is width of the pooling region

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

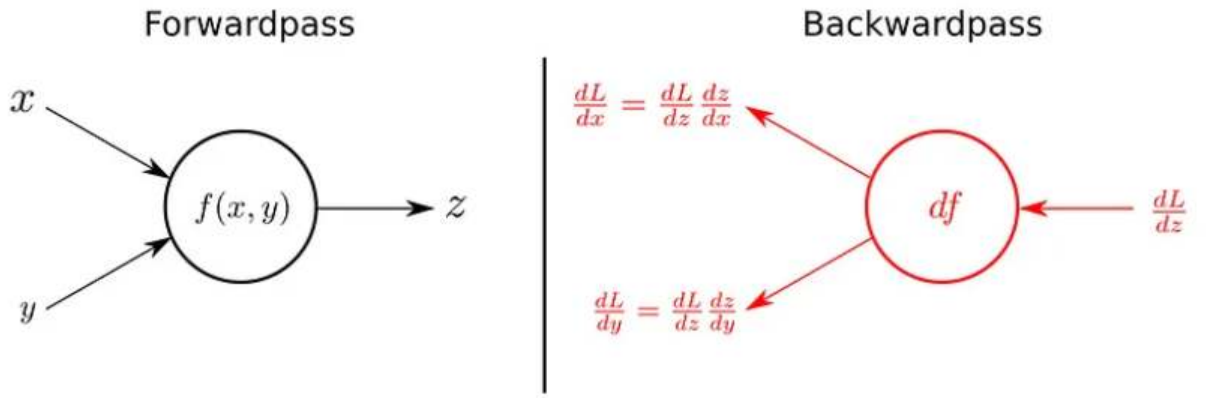
Backpropagation

Differently from convolution operations, we do not have to compute here weights and bias derivatives as there are no parameters in a pooling operation. Thus, the only derivative we need to compute is with respect to the input, ∂Y/ ∂X. We know that the derivative with respect to the inputs will have the same shape as the input.

For the backward in a max pool layer, we pass of the gradient, we start with a zero matrix and fill the max index of this matrix with the gradient from above. On the other hand, if we treat it as an average pool layer, we need to fill each cell with the value of the gradient from above.

The Chain Rule

The following figure summarises the use of chain rule for the backward pass in computational graphs.



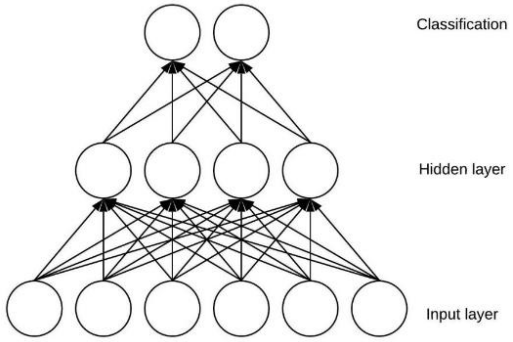
The forward pass on the left calculates **z** as a function **f(x,y)** using the input variables **x** and **y.** The right side of the figures shows the backward pass. Receiving **dL/dz**, the gradient of the loss function with respect to **z** from above, the gradients of **x** and **y** on the loss function can be calculate by applying the chain rule, as shown in the figure (borrowed from this post)

12.Dropout Layers and Regularization

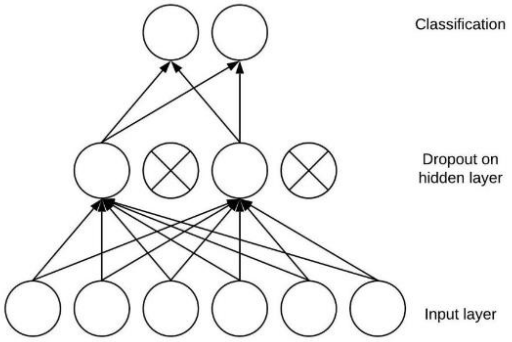
When you go out to buy a clothing for yourself, you will not buy something which is very fit for your body because if you become fat, it will not be convenient and you will not buy something that is very loose because then it looks like a cloth hanging on a skeleton, you will try to buy a right fit for your body. This problem of overfitting and underfitting happens in the machine learning project as well, and there are techniques to tackle this overfitting and under fitting issue and these techniques are called *regularization* techniques. A simple and powerful regularization technique for neural networks and deep learning models is *dropout*.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)



Without Dropout



With Dropout

What's Dropout?

In machine learning, "dropout" refers to the practice of disregarding certain nodes in a layer at random during training. A dropout is a regularization approach that prevents overfitting by ensuring that no units are co-dependent with one another.

Dropout Regularization

When you have training data, if you try to train your model too much, it might overfit, and when you get the actual test data for making predictions, it will not probably perform well. Dropout regularization is one technique used to tackle overfitting problems in deep learning.

Training with Drop-Out Layers

Dropout is a regularization method approximating concurrent training of many neural networks with various designs. During training, some layer outputs are ignored or dropped at random. This makes the layer appear and is regarded as having a different number of nodes and connectedness to the preceding layer. In practice, each layer update during training is carried out with a different perspective of the specified layer. Dropout makes the training process noisy, requiring nodes within a layer to take on more or less responsible for the inputs on a probabilistic basis.

According to this conception, dropout may break apart circumstances in which network tiers co-adapt to fix mistakes committed by prior layers, making the model more robust. Dropout is implemented per layer in a neural network. It works with the vast majority of layers, including dense, fully connected, convolutional, and recurrent layers such as the long short-term memory network layer. Dropout can occur on any or all of the network's hidden layers as well as the visible or input layer. It is not used on the output layer.

Why will dropout help with overfitting?

- ✓ It can't rely on one input as it might be randomly dropped out.

  You can imagine that if neurons are randomly dropped out of the network during training, that other neurons will have to step in and handle the representation required to make predictions for the missing neurons. This is believed to result in multiple independent internal representations being learned by the network.

- ✓ Neurons will not learn redundant details of inputs.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

The effect is that the network becomes less sensitive to the specific weights of neurons. This in turn results in a network that is capable of better generalization and is less likely to overfit the training data.

The Drawbacks of Dropout

Although dropout is a potent tool, it has certain downsides. A dropout network may take 2-3 times longer to train than a normal network. Finding a regularizer virtually comparable to a dropout layer is one method to reap the benefits of dropout without slowing down training. This regularizer is a modified variant of L2 regularisation for linear regression. An analogous regularizer for more complex models has yet to be discovered until that time when doubt drops out.

How to tune the dropout level on your problem?

Below are some of the useful heuristics to consider when using dropout in practice.

- Generally, use a small dropout value of 20%-50% of neurons with 20% providing a good starting point. A probability too low has minimal effect and a value too high results in under-learning by the network.
- Use a larger network. You are likely to get better performance when dropout is used on a larger network, giving the model more of an opportunity to learn independent representations.
- Use dropout on incoming (visible) as well as hidden units. Application of dropout at each layer of the network has shown good results.
- Use a large learning rate with decay and a large momentum. Increase your learning rate by a factor of 10 to 100 and use a high momentum value of 0.9 or 0.99.
- Constrain the size of network weights. A large learning rate can result in very large network weights. Imposing a constraint on the size of network weights such as max-norm regularization with a size of 4 or 5 has been shown to improve results.

13.Batch Normalization

The regularization techniques help to improve a model and allows it to converge faster. We have several regularization tools, some of them are early stopping, dropout, weight initialization techniques, and batch normalization. The regularization helps in preventing the over-fitting of the model and the learning process becomes more efficient.

The Term Normalization

Normalization is a data pre-processing tool used to bring the numerical data to a common scale without distorting its shape.

Generally, when we input the data to a machine or deep learning algorithm, we tend to change the values to a balanced scale. The reason we normalize is partly to ensure that our model can generalize appropriately.

What is Batch Normalization?

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Batch normalization, it is a process to make neural networks faster and more stable through adding extra layers in a deep neural network. The new layer performs the standardizing and normalizing operations on the input of a layer coming from a previous layer.

But what is the reason behind the term "Batch" in batch normalization? A typical neural network is trained using a collected set of input data called batch. Similarly, the normalizing process in batch normalization takes place in batches, not as a single input.

In other words, if we are able to somehow normalize the activations from each previous layer then the gradient descent will converge better during training. This is precisely what the Batch Norm layer does for us.

Batch Norm is just another network layer that gets inserted between a hidden layer and the next hidden layer. Its job is to take the outputs from the first hidden layer and normalize them before passing them on as the input of the next hidden layer.

How does Batch Normalization work?

It is a two-step process. First, the input is normalized, and later rescaling and offsetting is performed.

- ***Normalization of the Input***

Normalization is the process of transforming the data to have a mean zero and standard deviation one. In this step we have our batch input from layer h, first, we need to calculate the mean of this hidden activation.

$$\mu = \frac{1}{m} \sum h_i$$

Here, m is the number of neurons at layer h.

Once we have meant at our end, the next step is to calculate the standard deviation of the hidden activations.

$$\sigma = \left[\frac{1}{m} \sum (h_i - \mu)^2 \right]^{1/2}$$

Further, as we have the mean and the standard deviation ready. We will normalize the hidden activations using these values. For this, we will subtract the mean from each input and divide the whole value with the sum of standard deviation and the smoothing term (ε).

The smoothing term(ε) assures numerical stability within the operation by stopping a division by a zero value.

$$h_{i(norm)} = \frac{(h_i - \mu)}{\sigma + \varepsilon}$$

- ***Rescaling of Offsetting***

In the final operation, the re-scaling and offsetting of the input take place. Here two components of the BN algorithm come into the picture, γ(gamma) and β (beta). These parameters are used for re-scaling (γ) and shifting(β) of the vector containing values from the previous operations.

$$h_i = \gamma h_{i(norm)} + \beta$$

These two are learnable parameters, during the training neural network ensures the optimal values of γ and β are used. That will enable the accurate normalization of each batch.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Advantages of Batch Normalization

- ✓ Speed Up the Training
  By Normalizing the hidden layer activation, the Batch normalization speeds up the training process.

- ✓ Handles internal covariate shift
  It solves the problem of internal covariate shift. Through this, we ensure that the input for every layer is distributed around the same mean and standard deviation.

- ✓ Smoothens the Loss Function
  Batch normalization smoothens the loss function that in turn by optimizing the model parameters improves the training speed of the model.

14.Various Activation Functions

Activation Function

One of the most important parameters of the CNN model is the activation function. They are used to learn and approximate any kind of continuous and complex relationship between variables of the network. In simple words, it decides which information of the model should fire in the forward direction and which ones should not at the end of the network.

It adds non-linearity to the network. There are several commonly used activation functions such as the ReLU, Softmax, tanH and the Sigmoid functions. Each of these functions have a specific usage. For a binary classification CNN model, sigmoid and softmax functions are preferred a for a multi-class classification, generally softmax us used. In simple terms, activation functions in a CNN model determine whether a neuron should be activated or not. It decides whether the input to the work is important or not to predict using mathematical operations.

Refer Page No. 43 for more information (Covered in Unit-2)

15.Various Optimizers

Optimizers in Deep Learning

While neural networks are all the hype at the moment, an optimizer is something that is much more fundamental to the learning of a neural network. While neural networks can learn on their own, with no previous knowledge, an optimizer is a program that runs with the neural network, and allows it to learn much faster. It does this, in short, by altering the parameters of the neural network in such a way as to make training with that neural network far quicker and easier. These optimizers are what allow neural networks to work in real-time and training only takes a few minutes. Without them, training would easily take days.

Why is it difficult to optimize deep learning algorithms?

Deep learning systems are generally considered hard to optimize, because they are large and complex, often involving multiple layers and non-linearities. Optimizers need to deal with a huge system, which is not easy to understand. Some deep learning tools provide only a limited number of parameters that can be tuned, which limits their utility. There are, however, some effective ways to optimize deep learning models and improve their generalization.

Different Optimizers in Deep Learning

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

In deep learning, optimizers are used to adjust the parameters for a model. The purpose of an optimizer is to adjust model weights to maximize a loss function. The loss function is used as a way to measure how well the model is performing. An optimizer must be used when training a neural network model. There are a variety of different optimizers that can be used with a deep learning model. Some of the most popular optimizers are the RMSprop, momentum, and Adam.

- **RMSProp**

It is an improvement to the AdaGrad optimizer. This aims to reduce the aggressiveness of the learning rate by taking an exponential average of the gradients instead of the cumulative sum of squared gradients. Adaptive learning rate remains intact as now exponential average will punish larger learning rate in conditions when there are fewer updates and smaller rate in a higher number of updates.

- **Adam**

Adaptive Moment Estimation combines the power of RMSProp (root-mean-square prop) and momentum-based GD. In Adam optimizers, the power of momentum GD to hold the history of updates and the adaptive learning rate provided by RMSProp makes Adam optimizer a powerful method. It also introduces two new hyper-parameters beta1 and beta2 which are usually kept around 0.9 and 0.99 but you can change them according to your use case.

Refer Page No. 40 for more information (Covered in Unit-2)

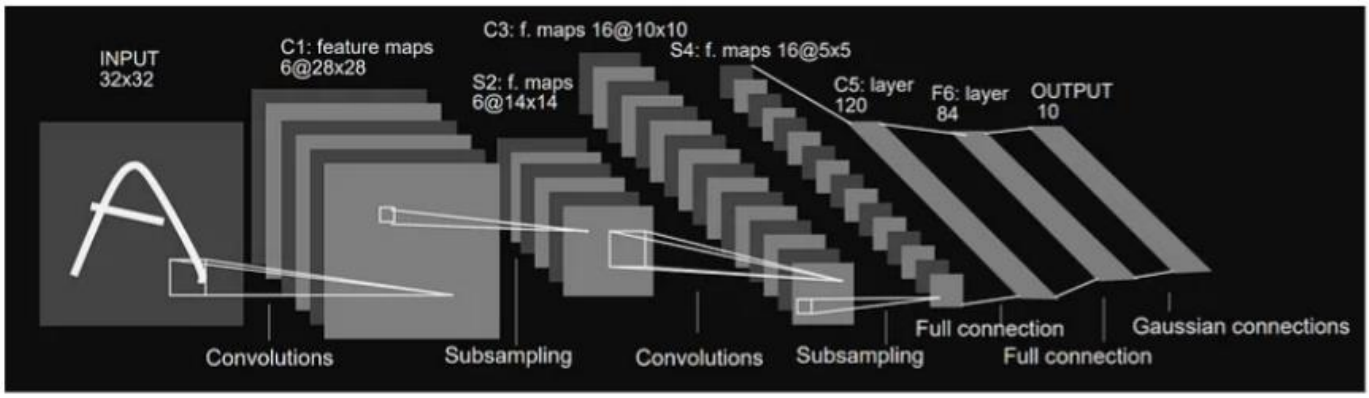
16.LeNet

What is LeNet5?

LeNet-5 is one of the earliest pre-trained models proposed by Yann LeCun and others in the year 1998, in the research paper Gradient-Based Learning Applied to Document Recognition. They used this architecture for recognizing the handwritten and machine-printed characters. The main reason behind the popularity of this model was its simple and straightforward architecture. It is a multi-layer convolution neural network for image classification.

The Architecture of the Model

LeNet-5 CNN architecture is made up of 7 layers. The layer composition consists of 3 convolutional layers, 2 subsampling layers and 2 fully connected layers.



LeNet-5 Architecture

The first layer is the input layer — this is generally not considered a layer of the network as nothing is learnt in this layer. The input layer is built to take in 32x32, and these are the dimensions of images that

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

are passed into the next layer. Those who are familiar with the MNIST dataset will be aware that the MNIST dataset images have the dimensions 28x28. To get the MNIST images dimension to the meet the requirements of the input layer, the 28x28 images are padded.

The grayscale images used in the research paper had their pixel values normalized from 0 to 255, to values between -0.1 and 1.175. The reason for normalization is to ensure that the batch of images have a mean of 0 and a standard deviation of 1, the benefits of this is seen in the reduction in the amount of training time.

Finally, to summarize the network has

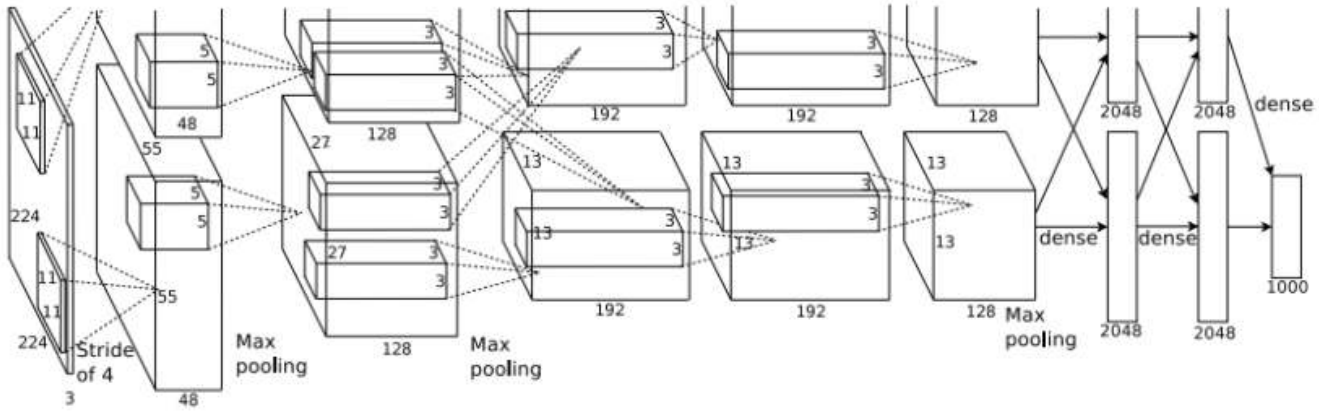
- 5 layers with learnable parameters.
- The input to the model is a grayscale image.
- It has 3 convolution layers, two average pooling layers, and two fully connected layers with a softmax classifier.
- The number of trainable parameters is around 60000.

17.AlexNet

What is AlexNet?

AlexNet was designed by Hinton, winner of the 2012 ImageNet competition, and his student Alex Krizhevsky. It was also after that year that more and deeper neural networks were proposed, such as the excellent VGG, GoogleNet. Its official data model has an accuracy rate of 57.1% and top 1-5 reaches 80.2%. This is already quite outstanding for traditional machine learning classification algorithms.

The Architecture of AlexNet



The AlexNet has eight layers with learnable parameters. The model consists of five layers with a combination of max pooling followed by 3 fully connected layers and they use ReLU activation in each of these layers except the output layer.

They found out that using the ReLU as an activation function accelerated the speed of the training process by almost six times. They also used the dropout layers, that prevented their model from overfitting. Further, the model is trained on the ImageNet dataset. The ImageNet dataset has almost 14 million images across a thousand classes.

To quickly summarize the architecture.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

- It has 8 layers with learnable parameters.
- The input to the Model is RGB images.
- It has 5 convolution layers with a combination of max-pooling layers.
- Then it has 3 fully connected layers.
- The activation function used in all layers is ReLU.
- It used two Dropout layers.
- The activation function used in the output layer is Softmax.
- The total number of parameters in this architecture is 62.3 million.

18.VGG 16

What is VGG16?

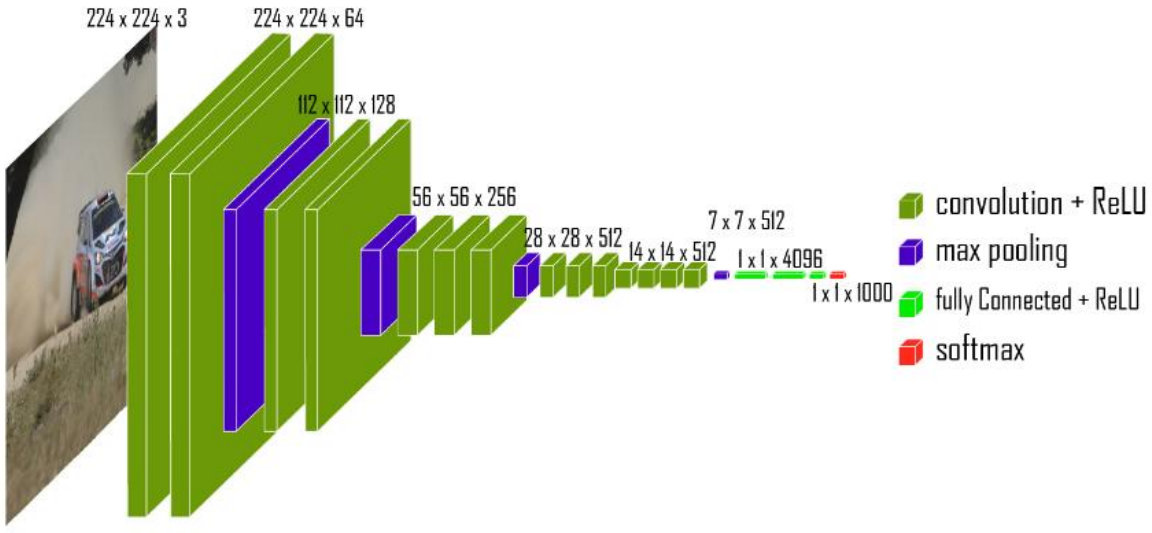
VGG16 proved to be a significant milestone in the quest of mankind to make computers "see" the world. A lot of effort has been put into improving this ability under the discipline of Computer Vision (CV) for a number of decades. VGG16 is one of the significant innovations that paved the way for several innovations that followed in this field.

It is a Convolutional Neural Network (CNN) model proposed by Karen Simonyan and Andrew Zisserman at the University of Oxford. The idea of the model was proposed in 2013, but the actual model was submitted during the ILSVRC ImageNet Challenge in 2014. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) was an annual competition that evaluated algorithms for image classification (and object detection) at a large scale. They did well in the challenge but couldn't win.

The Architecture of VGG16

VGG16, as its name suggests, is a 16-layer deep neural network. VGG16 is thus a relatively extensive network with a total of 138 million parameters—it's huge even by today's standards. However, the simplicity of the VGGNet16 architecture is its main attraction.

The VGGNet architecture incorporates the most important convolution neural network features.



VGG-16 architecture

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

A VGG network consists of small convolution filters. VGG16 has three fully connected layers and 13 convolutional layers.

Here is a quick outline of the VGG architecture:

- **Input**—VGGNet receives a 224×224 image input. In the ImageNet competition, the model's creators kept the image input size constant by cropping a 224×224 section from the centre of each image.
- **Convolutional layers**—the convolutional filters of VGG use the smallest possible receptive field of 3×3. VGG also uses a 1×1 convolution filter as the input's linear transformation.
- **ReLu activation**—next is the Rectified Linear Unit Activation Function (ReLU) component, Alex Net's major innovation for reducing training time. ReLU is a linear function that provides a matching output for positive inputs and outputs zero for negative inputs. VGG has a set convolution stride of 1 pixel to preserve the spatial resolution after convolution (the stride value reflects how many pixels the filter "moves" to cover the entire space of the image).
- **Hidden layers**—all the VGG network's hidden layers use ReLU instead of Local Response Normalization like AlexNet. The latter increases training time and memory consumption with little improvement to overall accuracy.
- **Pooling layers**–A pooling layer follows several convolutional layers—this helps reduce the dimensionality and the number of parameters of the feature maps created by each convolution step. Pooling is crucial given the rapid growth of the number of available filters from 64 to 128, 256, and eventually 512 in the final layers.
- **Fully connected layers**—VGGNet includes three fully connected layers. The first two layers each have 4096 channels, and the third layer has 1000 channels, one for every class.

19.ResNet

What is ResNet?

ResNet, short for Residual Network is a specific type of neural network that was introduced in 2015 by Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun in their paper "Deep Residual Learning for Image Recognition". The ResNet models were extremely successful which you can guess from the following:

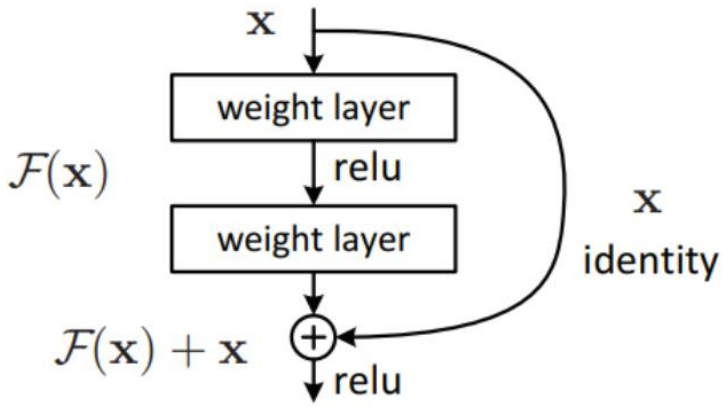
- Won 1st place in the ILSVRC 2015 classification competition with a top-5 error rate of 3.57% (An ensemble model)
- Won the 1st place in ILSVRC and COCO 2015 competition in ImageNet Detection, ImageNet localization, Coco detection and Coco segmentation.
- Replacing VGG-16 layers in Faster R-CNN with ResNet-101. They observed relative improvements of 28%
- Efficiently trained networks with 100 layers and 1000 layers also.

Residual Blocks

The problem of training very deep networks has been relieved with the introduction of these Residual blocks and the ResNet model is made up of these blocks.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)



Skip (Shortcut) connection

The problem of training very deep networks has been relieved with the introduction of these Residual blocks and the ResNet model is made up of these blocks.

In the above figure, the very first thing we can notice is that there is a direct connection that skips some layers of the model. This connection is called 'skip connection' and is the heart of residual blocks. The output is not the same due to this skip connection. Without the skip connection, input 'X gets multiplied by the weights of the layer followed by adding a bias term.

Then comes the activation function, f() and we get the output as H(x).

```
H(x)=f( wx + b ) or H(x)=f(x)
```

Now with the introduction of a new skip connection technique, the output is H(x) is changed to

```
H(x)=f(x)+x
```

But the dimension of the input may be varying from that of the output which might happen with a convolutional layer or pooling layers. Hence, this problem can be handled with these two approaches:

- Zero is padded with the skip connection to increase its dimensions.
- 1×1 convolutional layers are added to the input to match the dimensions. In such a case, the output is: `H(x)=f(x)+w1.x`

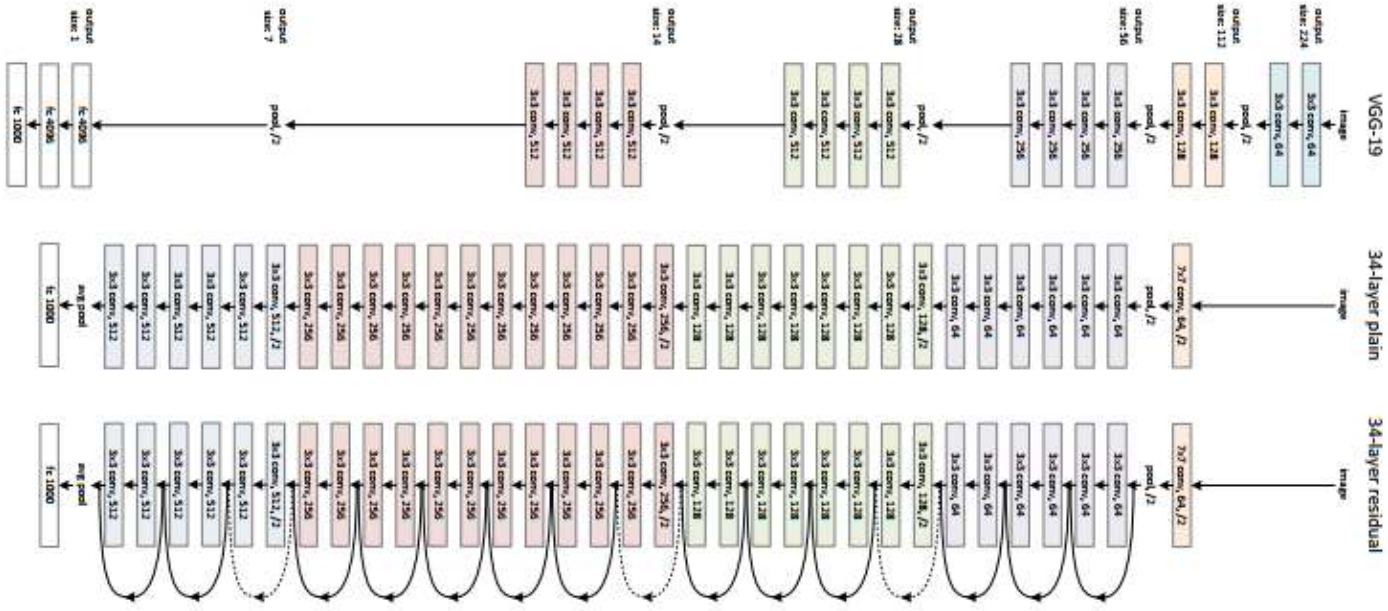
Here an additional parameter w1 is added whereas no additional parameter is added when using the first approach.

These skip connections technique in ResNet solves the problem of vanishing gradient in deep CNNs by allowing alternate shortcut path for the gradient to flow through. Also, the skip connection helps if any layer hurts the performance of architecture, then it will be skipped by regularization.

Architecture of ResNet

There is a 34-layer plain network in the architecture that is inspired by VGG-19 in which the shortcut connection or the skip connections are added. These skip connections or the residual blocks then convert the architecture into the residual network as shown in the figure below.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)



Summary of some of the more popular Architectures of CNNs

- **LeNet – 5 (1998)**
  - One of the earliest successful architectures of CNNs.
  - Developed by Yann LeCun.
  - Originally used to read digits in images.

- **AlexNet (2012)**
  - Helped popularize CNNs in computer vision.
  - Developed by Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton.
  - A 2012 network that made history. It has eight layers; the first five are convolutional layers, some of them followed by max-pooling layers, and the last three are fully connected layers. AlexNet used 11×11 convolutions in the first layer with a stride of 4. It also had an architecture that was running on two parallel GPUs (it predates the current deep learning frameworks).
  - Won the ILSVRC 2012. (ImageNet Large Scale Visual Recognition Challenge)

LSVRC (Large Scale Visual Recognition Challenge) *is a competition where research teams evaluate their algorithms on a huge dataset of labelled images (ImageNet) and compete to achieve higher accuracy on several visual recognition tasks. This made a huge impact on how teams approach the completion afterward.*

- **ZF Net (2013)**
  - Won the ILSVRC 2013.
  - Developed by Matthew Zeiler and Rob Fergus.
  - Introduced the visualization concept of the Deconvolutional Network.

- **GoogleNet / Inception (2014)**

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

- Won the ILSVRC 2014.
- Developed by Christian Szegedy and his team at Google.
- Codenamed "Inception," one variation has 22 layers.
- Inception (GoogleNet): This network (or better stated as 'family of networks') is in many ways close to the current state of the art of general convolutional neural networks. It is heavily used today. Key is the usage of inception modules which concatenate several convolutional operations together.

❖ **VGGNet (2014)**
- Runner-Up in the ILSVRC 2014.
- Developed by Karen Simonyan and Andrew Zisserman.
- Showed that depth of network was a critical factor in good performance.
- This was the next step after AlexNet and is still used today. To contrast VGG with AlexNet it is useful to consider the concept of the receptive field. The receptive field of a neuron in a CNN is defined as the region in the input space that the neuron is looking at. VGG uses combinations to smaller convolutions to achieve similar receptive fields to AlexNet (which uses bigger convolutions).

❖ **ResNet (2015)**
- Trained on very deep networks (up to 1,200 layers).
- Won first in the ILSVRC 2015 classification task.
- The ResNet (residual network) architecture introduces a new idea: residual connections. The main idea is to have layers work on the sum of the output from the direct previous layer as well as several layers back (2 or more). This architecture allows to overcome training problems that occur with very deep networks.

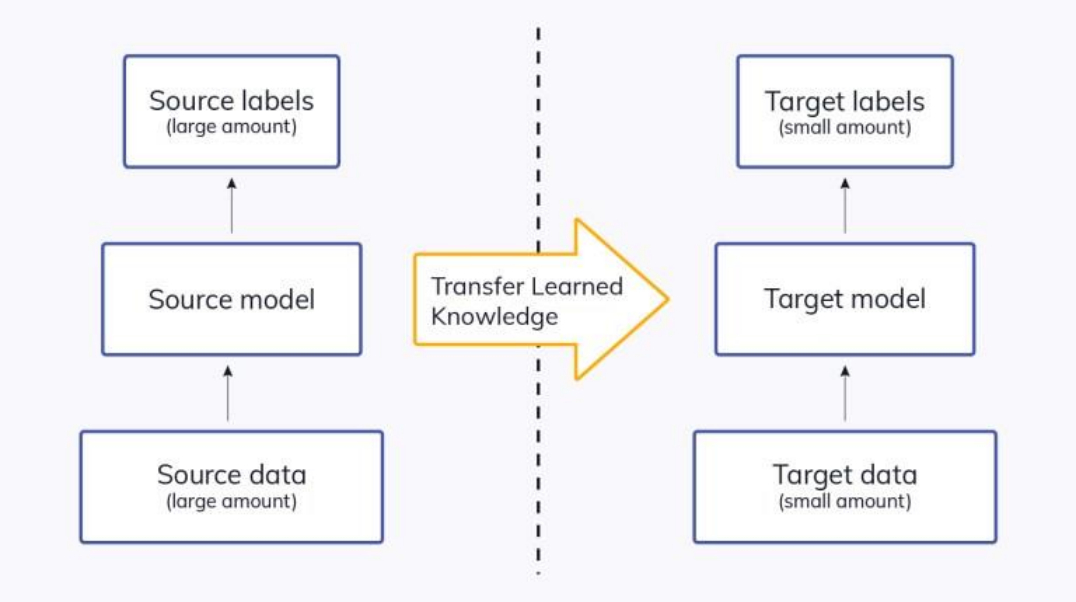
20.Transfer Learning with Image Data

What is Transfer Learning?

Transfer learning is about leveraging feature representations from a pre-trained model, so you don't have to train a new model from scratch. Transfer learning, used in machine learning, is the reuse of a pre-trained model on a new problem. In transfer learning, a machine exploits the knowledge gained from a previous task to improve generalization about another. The pre-trained models are usually trained on massive datasets that are a standard benchmark in the computer vision frontier. The weights obtained from the models can be reused in other computer vision tasks.

These models can be used directly in making predictions on new tasks or integrated into the process of training a new model. Including the pre-trained models in a new model leads to lower training time and lower generalization error. Transfer learning is particularly very useful when you have a small training dataset. In this case, you can, for example, use the weights from the pre-trained models to initialize the weights of the new model. *For example*, if you trained a simple classifier to predict whether an image contains a backpack, you could use the knowledge that the model gained during its training to recognize other objects like sunglasses.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)



With transfer learning, we basically try to exploit what has been learned in one task to improve generalization in another. We transfer the weights that a network has learned at "task A" to a new "task B." The general idea is to use the knowledge a model has learned from a task with a lot of available labelled training data in a new task that doesn't have much data. Instead of starting the learning process from scratch, we start with patterns learned from solving a related task.

Transfer learning is mostly used in computer vision and natural language processing tasks like sentiment analysis due to the huge amount of computational power required.

Transfer learning isn't really a machine learning technique, but can be seen as a "design methodology" within the field, for example, active learning. It is also not an exclusive part or study-area of machine learning. Nevertheless, it has become quite popular in combination with neural networks that require huge amounts of data and computational power.

When to use Transfer Learning?

As is always the case in machine learning, it is hard to form rules that are generally applicable, but here are some guidelines on when transfer learning might be used:

- There isn't enough labelled training data to train your network from scratch.
- There already exists a network that is pre-trained on a similar task, which is usually trained on massive amounts of data.
- When task 1 and task 2 have the same input.

If the original model was trained using an open-source library like TensorFlow, you can simply restore it and retrain some layers for your task. Keep in mind, however, that transfer learning only works if the features learned from the first task are general, meaning they can be useful for another related task as

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

well. Also, the input of the model needs to have the same size as it was initially trained with. If you don't have that, add a pre-processing step to resize your input to the needed size.

What is the difference between transfer learning and fine-tuning?

Fine-tuning is an optional step in transfer learning. Fine-tuning will usually improve the performance of the model. However, since you have to retrain the entire model, you'll likely overfit. Overfitting is avoidable. Just retrain the model or part of it using a low learning rate. This is important because it prevents significant updates to the gradient. These updates result in poor performance. Using a call-back to stop the training process when the model has stopped improving is also helpful.

How to implement Transfer Learning?

We can implement transfer learning in these six general steps.

1. Obtain the pre-trained model

The first step is to get the pre-trained model that you would like to use for your problem. There are some pre-trained machine learning models out there that are quite popular. One of them is the Inception-v3 model, which was trained for the ImageNet "Large Visual Recognition Challenge." Microsoft also offers some pre-trained models, available for both R and Python development, through the MicrosoftML R package and the MicrosoftML Python package. Other quite popular models are ResNet and AlexNet.

2. Create a base model

Usually, the first step is to instantiate the base model using one of the architectures such as *ResNet* or *Xception*. You can also optionally download the pre-trained weights. If you don't download the weights, you will have to use the architecture to train your model from scratch. Recall that the base model will usually have more units in the final output layer than you require. When creating the base model, you, therefore, have to remove the final output layer. Later on, you will add a final output layer that is compatible with your problem.

3. Freeze layers so they don't change during training

Freezing the layers from the pre-trained model is vital. This is because you don't want the weights in those layers to be re-initialized. If they are, then you will lose all the learning that has already taken place. This will be no different from training the model from scratch.

4. Add new trainable layers

The next step is to add new trainable layers that will turn old features into predictions on the new dataset. This is important because the pre-trained model is loaded without the final output layer.

5. Train the new layers on the dataset

Remember that the pre-trained model's final output will most likely be different from the output that you want for your model. For example, pre-trained models trained on the ImageNet dataset will output 1000 classes. However, your model might just have two classes. In this case, you have to train the model with a new output layer in place. Therefore, you will add some new dense layers as you please, but most importantly, a final dense layer with units corresponding to the number of outputs expected by your model.

6. Improve the model via fine-tuning

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Once you have done the previous step, you will have a model that can make predictions on your dataset. Optionally, you can improve its performance through fine-tuning. Fine-tuning is done by unfreezing the base model or part of it and training the entire model again on the whole dataset at a very low learning rate. The low learning rate will increase the performance of the model on the new dataset while preventing overfitting.

The learning rate has to be low because the model is quite large while the dataset is small. This is a recipe for overfitting, hence the low learning rate. Recompile the model once you have made these changes so that they can take effect. This is because the behaviour of a model is frozen whenever you call the compile function. That means that you have to call the compile function again whenever you want to change the model's behaviour. The next step will be to train the model again while monitoring it via call-backs to ensure it does not overfit.

21.Transfer Learning using Inception Oxford VGG Model

The Oxford VGG Models

Researchers from the *Oxford Visual Geometry Group*, or VGG for short, participate in the ILSVRC challenge. In 2014, convolutional neural network models (CNN) developed by the VGG won the image classification tasks. VGG released two different CNN models, specifically a 16-layer model and a 19-layer model. The VGG models are no longer state-of-the-art by only a few percentage points. Nevertheless, they are very powerful models and useful both as image classifiers and as the basis for new models that use image inputs. In the next section, we will see how we can use the VGG model directly in Keras.

Load the VGG Model in Keras

The VGG model can be loaded and used in the Keras deep learning library. Keras provides an Applications interface for loading and using pre-trained models. Using this interface, you can create a VGG model using the pre-trained weights provided by the Oxford group and use it as a starting point in your own model, or use it as a model directly for classifying images. In this tutorial, we will focus on the use case of classifying new images using the VGG model. Keras provides both the 16-layer and 19-layer version via the VGG16 and VGG19 classes. Let's focus on the VGG16 model.

The model can be created as follows:

```
from keras.applications.vgg16 import VGG16
model = VGG16()
```

That's it.

The first time you run this example, Keras will download the weight files from the Internet and store them in the `~/.keras/models` directory.

Note that the weights are about 528 megabytes, so the download may take a few minutes depending on the speed of your Internet connection. The weights are only downloaded once. The next time you run the example, the weights are loaded locally and the model should be ready to use in seconds.

We can use the standard Keras tools for inspecting the model structure. For example, you can print a summary of the network layers as follows:

```
from keras.applications.vgg16 import VGG16
model = VGG16()
print(model.summary())
```


This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

You can see that the model is huge. You can also see that, by default, the model expects images as input with the size 224 x 224 pixels with 3 channels (e.g., colour).

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 224, 224, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
fc1 (Dense)                  (None, 4096)              102764544
_________________________________________________________________
fc2 (Dense)                  (None, 4096)              16781312
_________________________________________________________________
predictions (Dense)          (None, 1000)              4097000
=================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
_________________________________________________________________
```

We can also create a plot of the layers in the VGG model, as follows: Again, because the model is large, the plot is a little too large and perhaps unreadable.

```
from keras.applications.vgg16 import VGG16
from keras.utils.vis_utils import plot_model
model = VGG16()
plot_model(model, to_file='vgg.png')
```


Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

The VGG() class takes a few arguments that may only interest you if you are looking to use the model in your own project, e.g. for transfer learning.

For example:

- **include_top** (True): Whether or not to include the output layers for the model. You don't need these if you are fitting the model on your own problem.
- **weights** ('imagenet'): What weights to load. You can specify None to not load pre-trained weights if you are interested in training the model yourself from scratch.
- **input_tensor** (None): A new input layer if you intend to fit the model on new data of a different size.
- **input_shape** (None): The size of images that the model is expected to take if you change the input layer.
- **pooling** (None): The type of pooling to use when you are training a new set of output layers.
- **classes** (1000): The number of classes (e.g., size of output vector) for the model.

Develop a Simple Photo Classifier

1. Get a Sample Image - First, we need an image we can classify. You can download a random photograph of a coffee mug. Download the image and save it to your current working directory with the filename 'mug.jpg'.

2. Load the VGG Model - Load the weights for the VGG-16 model

```
from keras.applications.vgg16 import VGG16
# load the model
model = VGG16()
```

3. Load and Prepare Image - Next, we can load the image as pixel data and prepare it to be presented to the network. Keras provides some tools to help with this step. First, we can use the load_img() function to load the image and resize it to the required size of 224×224 pixels.

```
from keras.preprocessing.image import load_img
# load an image from file
image = load_img('mug.jpg', target_size=(224, 224))
```

Next, we can convert the pixels to a NumPy array so that we can work with it in Keras. We can use the img_to_array() function for this.

```
from keras.preprocessing.image import img_to_array
# convert the image pixels to a numpy array
image = img_to_array(image)
```

The network expects one or more images as input; that means the input array will need to be 4-dimensional: samples, rows, columns, and channels.
We only have one sample (one image). We can reshape the array by calling reshape() and adding the extra dimension.

```
# reshape data for the model
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
```

Next, the image pixels need to be prepared in the same way as the ImageNet training data was prepared. Keras provides a function called preprocess_input() to prepare new input for the network.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

```
from keras.applications.vgg16 import preprocess_input
# prepare the image for the VGG model
image = preprocess_input(image)
```

We are now ready to make a prediction for our loaded and prepared image.

4. Make a Prediction - We can call the predict() function on the model in order to get a prediction of the probability of the image belonging to each of the 1000 known object types.

```
# predict the probability across all output classes
yhat = model.predict(image)
```

Nearly there, now we need to interpret the probabilities.

5. Interpret Prediction - Keras provides a function to interpret the probabilities called decode_predictions(). It can return a list of classes and their probabilities in case you would like to present the top 3 objects that may be in the photo. We will just report the first most likely object.

```
from keras.applications.vgg16 import decode_predictions
# convert the probabilities to class labels
label = decode_predictions(yhat)
# retrieve the most likely result, e.g. highest probability
label = label[0][0]
# print the classification
print('%s (%.2f%%)' % (label[1], label[2]*100))
```

And that's it.

Complete Example

```
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.applications.vgg16 import decode_predictions
from keras.applications.vgg16 import VGG16
# load the model
model = VGG16()
# load an image from file
image = load_img('mug.jpg', target_size=(224, 224))
# convert the image pixels to a numpy array
image = img_to_array(image)
# reshape data for the model
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
# prepare the image for the VGG model
image = preprocess_input(image)
# predict the probability across all output classes
yhat = model.predict(image)
# convert the probabilities to class labels
label = decode_predictions(yhat)
# retrieve the most likely result, e.g. highest probability
label = label[0][0]
# print the classification
print('%s (%.2f%%)' % (label[1], label[2]*100))
```


Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Running the example, we can see that the image is correctly classified as a "coffee mug" with a 75% likelihood. `coffee_mug (75.27%)`

22.Google Inception Model

Inception

Inception is a deep convolutional neural network architecture that was introduced in 2014. It won the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC14). It was mostly developed by Google researchers. Inception architecture can be used in computer vision tasks that imply convolutional filters. Inception models remain expensive to train. Transfer learning brings part of the solution when it comes to adapting such algorithms to your specific task.

Inception V1

When multiple deep layers of convolutions were used in a model it resulted in the overfitting of the data. To avoid this from happening the inception V1 model uses the idea of using multiple filters of different sizes on the same level. Thus, in the inception models instead of having deep layers, we have parallel layers thus making our model wider rather than making it deeper.

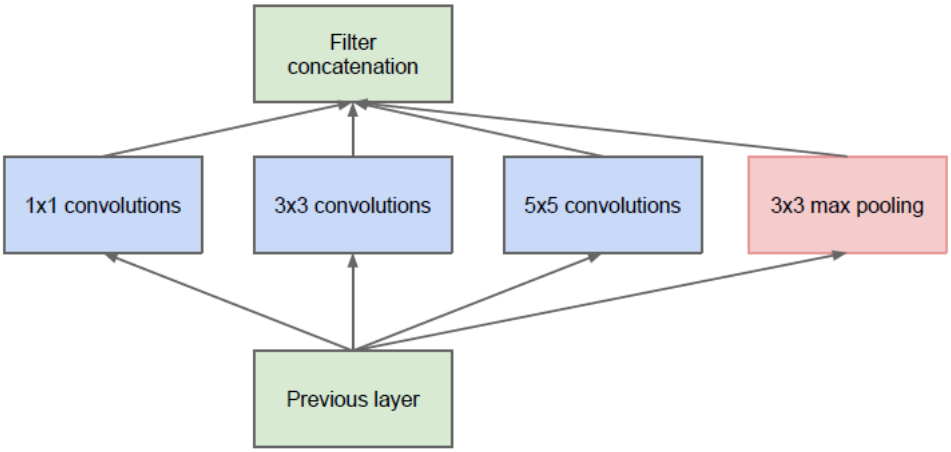
The Inception model is made up of multiple Inception modules. The basic module of the Inception V1 model is made up of four parallel layers.

- 1×1 convolution
- 3×3 convolution
- 5×5 convolution
- 3×3 max pooling

Convolution - The process of transforming an image by applying a kernel over each pixel and its local neighbours across the entire image.

Pooling - Pooling is the process used to reduce the dimensions of the feature map. There are different types of pooling but the most common ones are max pooling and average pooling.

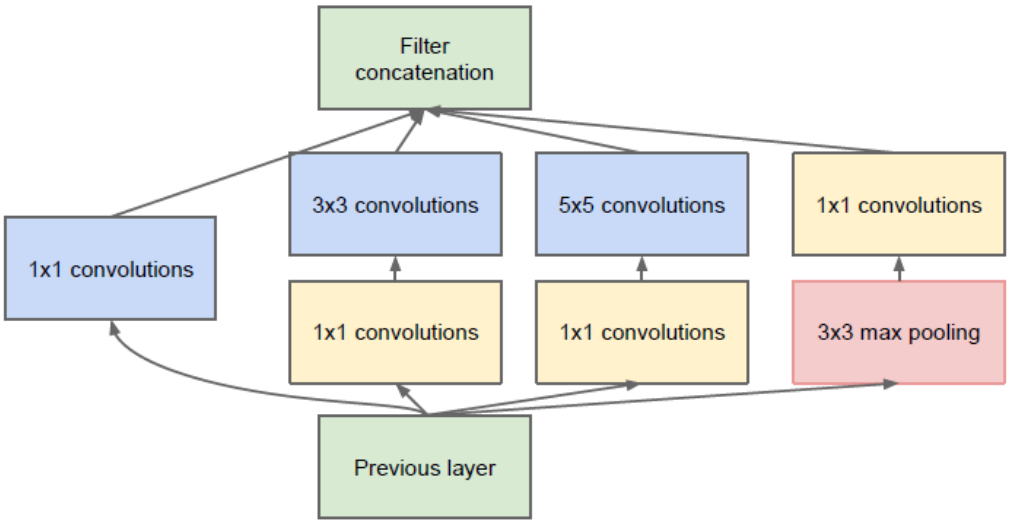
Here, different sizes of convolutions are performed to capture different sizes of information in the Picture.



This document is available free of charge on studocu
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

This module of the Inception V1 is called the Naive form. One of the drawbacks of this naive form is that even the 5×5 convolutional layer is computationally pretty expensive i.e., time-consuming and requires high computational power.

To overcome this the authors added a 1×1 convolutional layer before each convolutional layer, which results in reduced dimensions of the network and faster computations.



This is the building block of the Inception V1 model Architecture. The Inception V1 architecture model was better than most other models at that time with a very minimum error percentage.

What makes the Inception V3 model better?

The inception V3 is just the advanced and optimized version of the inception V1 model. The Inception V3 model used several techniques for optimizing the network for better model adaptation. In total, the inception V3 model is made up of 42 layers which is a bit higher than the previous inception V1 and V2 models. But the efficiency of this model is really impressive.

- ✓ It has higher efficiency
- ✓ It has a deeper network compared to the Inception V1 and V2 models, but its speed isn't compromised.
- ✓ It is computationally less expensive.
- ✓ It uses auxiliary Classifiers as regularizes.

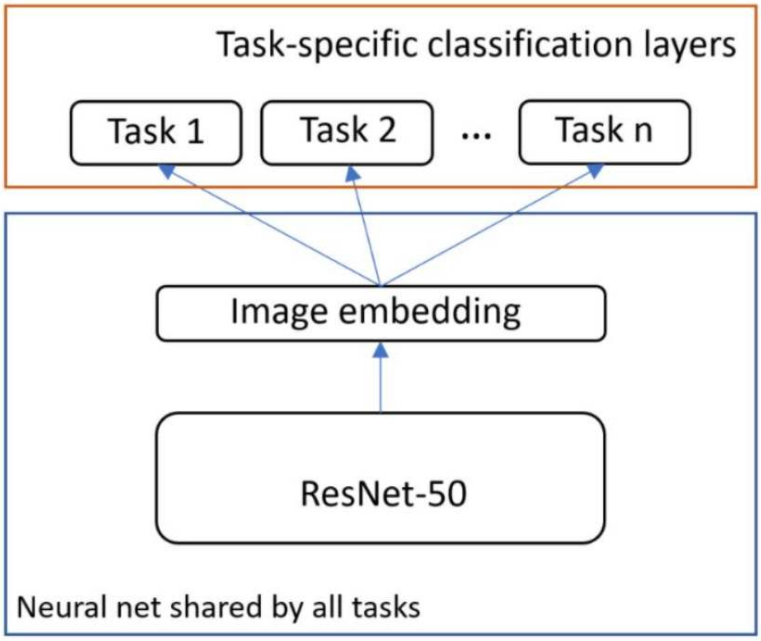
23.Microsoft ResNet Model

Pretrained vision models accelerate deep learning research and bring down the cost of performing computer vision tasks in production. By pretraining one large vision model to learn general visual representation of images, then transferring the learning across multiple downstream tasks, a team achieves competitive performance at a fraction of the cost when compared to collecting new data and training a new model for each task. Further fine-tuning of the pretrained model with task-specific training data often yields even higher performance than training specialized models.

Microsoft Vision Model ResNet-50 is a large pretrained vision model created by the Multimedia Group at Microsoft Bing. The model is built using the search engine's web-scale image data in order to power its Image Search and Visual Search. To achieve the state-of-the-art performance in a cost-sensitive production setting, Microsoft Vision Model ResNet-50 leverages multi-task learning and

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

optimizes separately for four datasets, including ImageNet-22k, Microsoft COCO, and two web-supervised datasets containing 40 million image-label pairs collected from image search engines.



Multi-task learning via hard parameter sharing

Microsoft chose to use multi-task learning with hard parameter sharing (see figure above). Single neural networks will optimize each classification problem at the same time. By using tasks of varied sizes—up to 40 million images with 100,000 different labels from web-supervised sources—Microsoft Vision Model ResNet-50 can achieve high robustness and good transferability to different domains.

During training, images from each dataset are sampled proportionally to the size of the datasets. This approach promotes larger datasets at first, but once optimization flattens, the optimizer needs to look for improvements to smaller datasets without downgrading the performance of the larger ones. As a result, final accuracy scores for each of the datasets are competitive with specialized models trained on each specific dataset.

24.R-CNN

Object Detection

Computer vision is an interdisciplinary field that has been gaining huge amounts of traction in the recent years (since CNN) and self-driving cars have taken centre stage. Another integral part of computer vision is object detection. Object detection aids in pose estimation, vehicle detection, surveillance etc. The difference between object detection algorithms and classification algorithms is that in detection algorithms, we try to draw a bounding box around the object of interest to locate it within the image. Also, you might not necessarily draw just one bounding box in an object detection case, there could be many bounding boxes representing different objects of interest within the image and you would not know how many beforehand.

Need for R-CNN

The major reason why you cannot proceed with this problem by building a standard convolutional network followed by a fully connected layer is that, the length of the output layer is variable — not constant, this is because the number of occurrences of the objects of interest is not fixed. A naive approach to solve this problem would be to take different regions of interest from the image, and use a CNN to classify the presence of the object within that region. The problem with this approach

This document is available free of charge on studocu

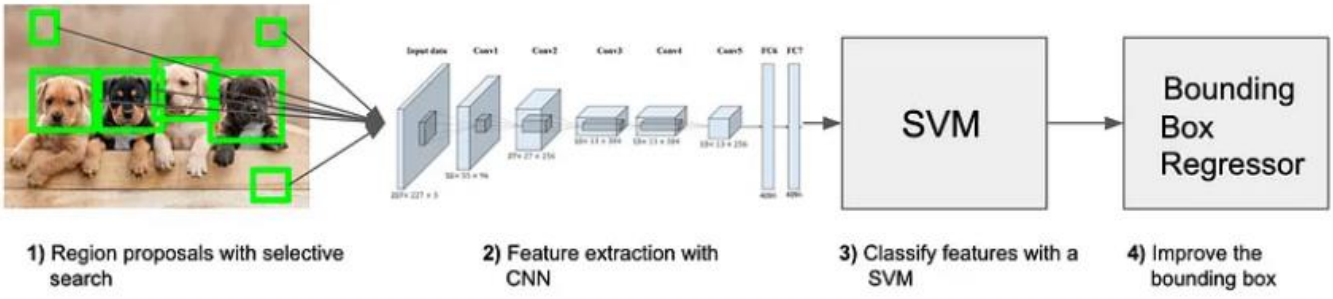
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

is that the objects of interest might have different spatial locations within the image and different aspect ratios. Hence, you would have to select a huge number of regions and this could computationally blow up. Therefore, algorithms like R-CNN, YOLO etc have been developed to find these occurrences and find them fast.

R-CNN

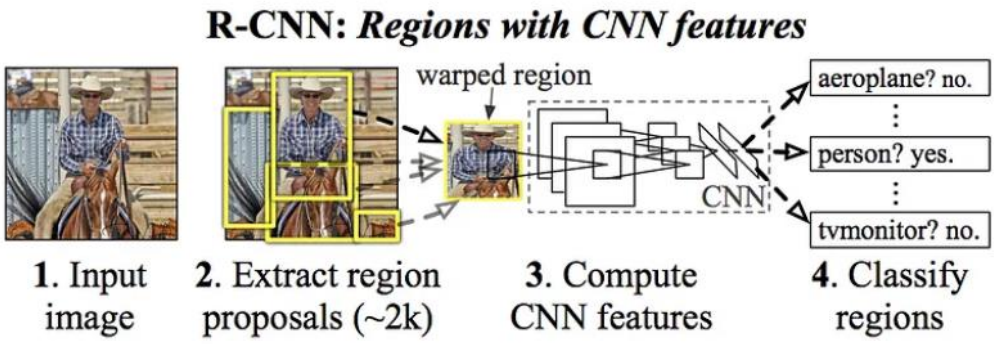
To bypass the problem of selecting a huge number of regions, Ross Girshick et al. proposed a method where we use selective search to extract just 2000 regions from the image and he called them region proposals. Therefore, now, instead of trying to classify a huge number of regions, you can just work with 2000 regions. These 2000 region proposals are generated using the selective search algorithm which is written below.

```
Selective Search:
1. Generate initial sub-segmentation, we generate many candidate
regions
2. Use greedy algorithm to recursively combine similar regions into
larger ones
3. Use the generated regions to produce the final candidate region
proposals
```



The R-CNN system

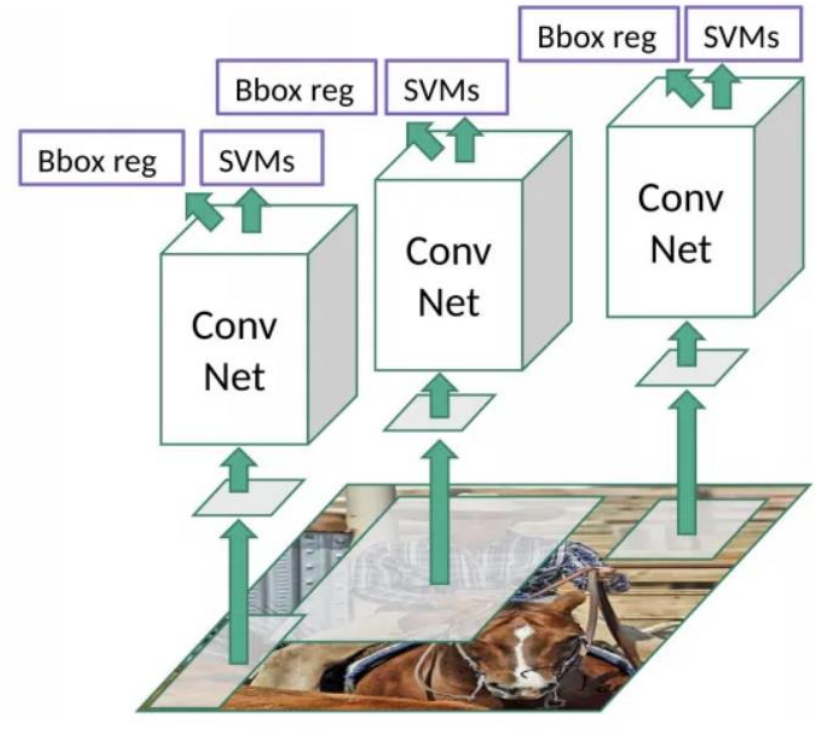
These 2000 candidate region proposals are warped into a square and fed into a convolutional neural network that produces a 4096-dimensional feature vector as output. The CNN acts as a feature extractor and the output dense layer consists of the features extracted from the image and the extracted features are fed into an SVM to classify the presence of the object within that candidate region proposal.



R-CNN

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

In addition to predicting the presence of an object within the region proposals, the algorithm also predicts four values which are offset values to increase the precision of the bounding box. For example, given a region proposal, the algorithm would have predicted the presence of a person but the face of that person within that region proposal could've been cut in half. Therefore, the offset values help in adjusting the bounding box of the region proposal.



R-CNN

Problems with R-CNN

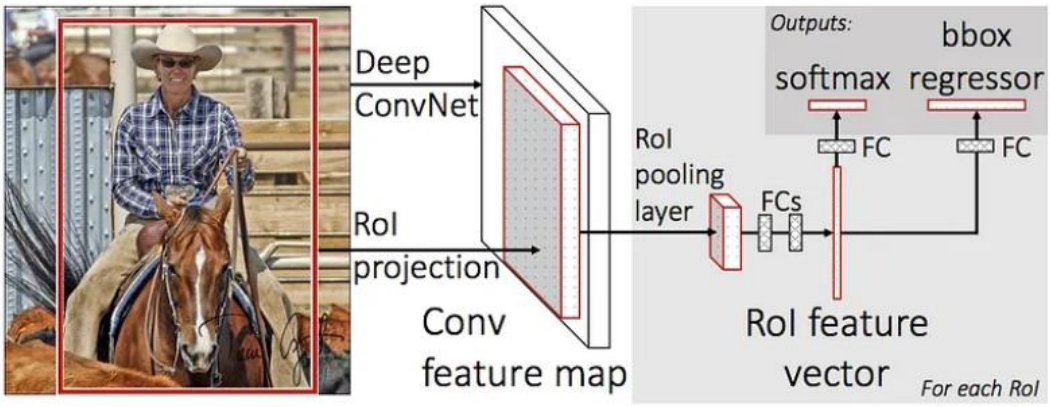
- It still takes a huge amount of time to train the network as you would have to classify 2000 region proposals per image.
- It cannot be implemented real time as it takes around 47 seconds for each test image.
- The selective search algorithm is a fixed algorithm. Therefore, no learning is happening at that stage. This could lead to the generation of bad candidate region proposals.

25.Fast R-CNN

The same author of the previous paper(R-CNN) solved some of the drawbacks of R-CNN to build a faster object detection algorithm and it was called Fast R-CNN. The approach is similar to the R-CNN algorithm. But, instead of feeding the region proposals to the CNN, we feed the input image to the CNN to generate a convolutional feature map. From the convolutional feature map, we identify the region of proposals and warp them into squares and by using a RoI pooling layer we reshape them into a fixed size so that it can be fed into a fully connected layer. From the RoI feature vector, we use a softmax layer to predict the class of the proposed region and also the offset values for the bounding box.

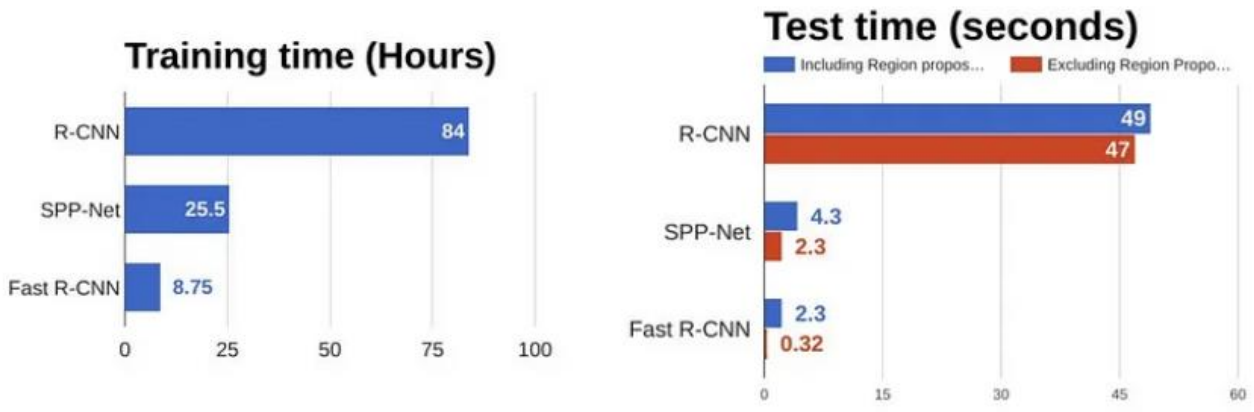
This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)



Fast R-CNN

The reason "Fast R-CNN" is faster than R-CNN is because you don't have to feed 2000 region proposals to the convolutional neural network every time. Instead, the convolution operation is done only once per image and a feature map is generated from it.



Comparison of object detection algorithms

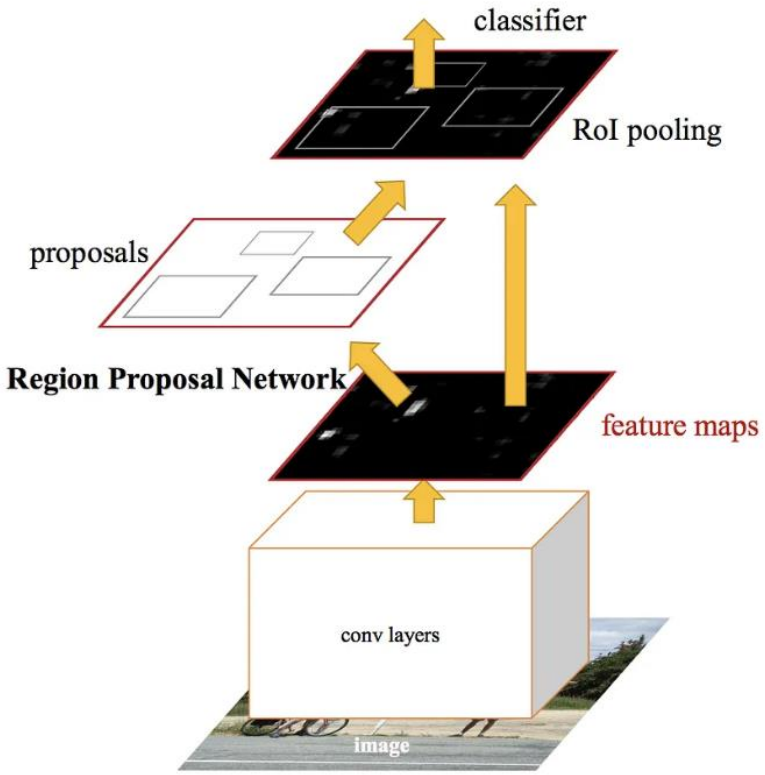
From the above graphs, you can infer that Fast R-CNN is significantly faster in training and testing sessions over R-CNN. When you look at the performance of Fast R-CNN during testing time, including region proposals slows down the algorithm significantly when compared to not using region proposals. Therefore, region proposals become bottlenecks in Fast R-CNN algorithm affecting its performance.

26.Faster R-CNN

Both of the above algorithms (R-CNN & Fast R-CNN) uses selective search to find out the region proposals. Selective search is a slow and time-consuming process affecting the performance of the network. Therefore, Shaoqing Ren et al. came up with an object detection algorithm that eliminates the selective search algorithm and lets the network learn the region proposals.

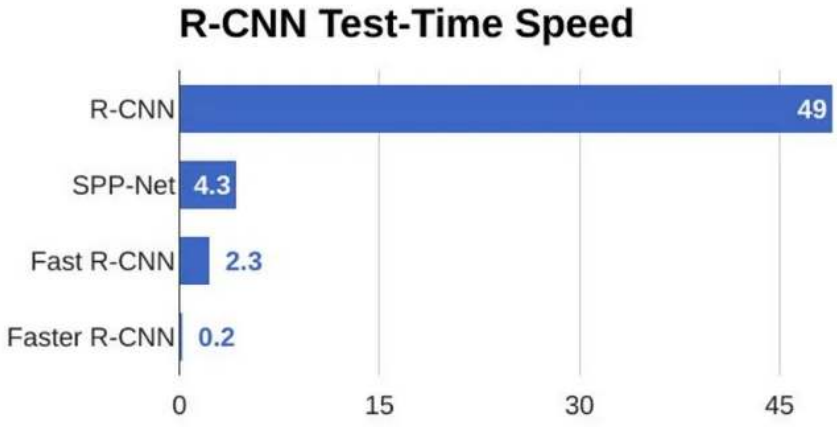
Similar to Fast R-CNN, the image is provided as an input to a convolutional network which provides a convolutional feature map. Instead of using selective search algorithm on the feature map to identify the region proposals, a separate network is used to predict the region proposals. The predicted region proposals are then reshaped using a RoI pooling layer which is then used to classify the image within the proposed region and predict the offset values for the bounding boxes.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)



Faster R-CNN

From the below graph, you can see that Faster R-CNN is much faster than its predecessors. Therefore, it can even be used for real-time object detection.



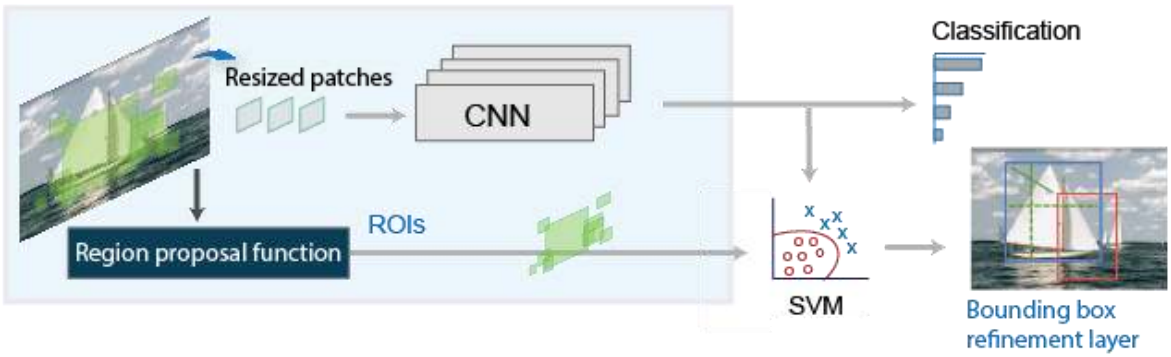
Comparison of test-time speed of object detection algorithms

This document is available free of charge on studocu

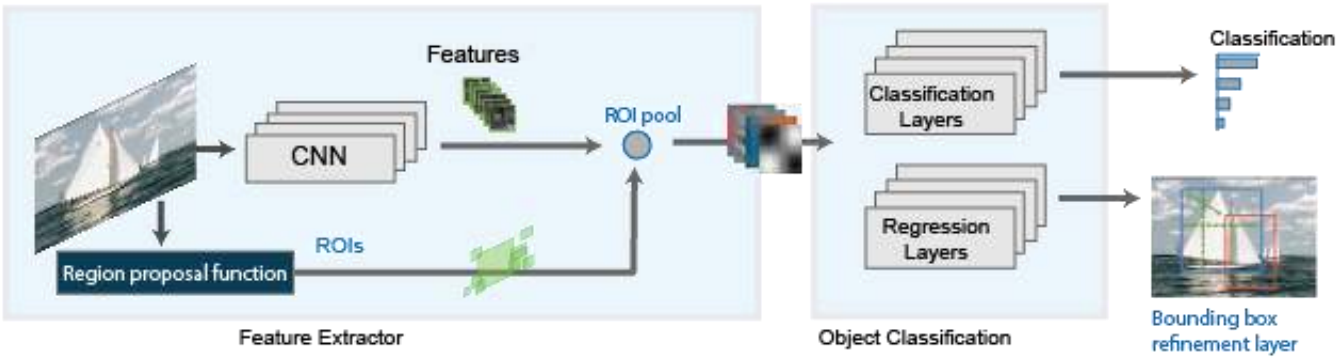
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Comparing R-CNN, Fast R-CNN and Faster R-CNN

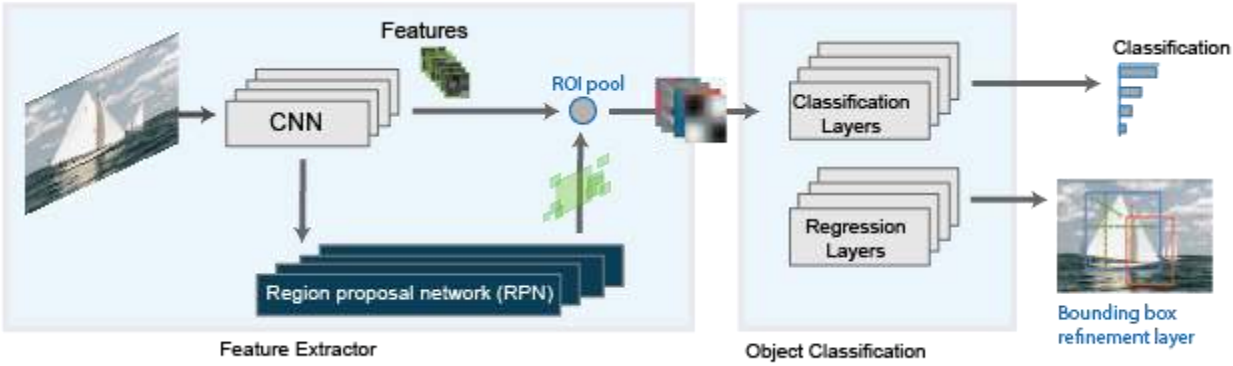
R-CNN



Fast R-CNN



Faster R-CNN



Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

	R-CNN	Fast R-CNN	Faster R-CNN
region proposals method	Selective search	Selective search	Region proposal network
Prediction timing	40-50 sec	2 seconds	0.2 seconds
computation	High computation time	High computation time	Low computation time
The mAP on Pascal VOC 2007 test dataset(%)	58.5	66.9 (when trained with VOC 2007 only) 70.0 (when trained with VOC 2007 and 2012 both)	69.9(when trained with VOC 2007 only)
The mAP on Pascal VOC 2012 test dataset (%)	53.3	65.7 (when trained with VOC 2012 only) 68.4 (when trained with VOC 2007 and 2012 both)	67.0(when trained with VOC 2012 only) 70.4 (when trained with VOC 2007 and 2012 both) 75.9(when trained with VOC 2007 and 2012 and COCO)

27.Mask -RCNN

What is Mask R-CNN?

Mask R-CNN, or Mask RCNN, is a Convolutional Neural Network (CNN) and state-of-the-art in terms of image segmentation and instance segmentation. Mask R-CNN was developed on top of Faster R-CNN, a Region-Based Convolutional Neural Network. The first step to understanding how Mask R-CNN work requires an understanding of the concept of Image Segmentation.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

The computer vision task Image Segmentation is the process of partitioning a digital image into multiple segments (sets of pixels, also known as image objects). This segmentation is used to locate objects and boundaries (lines, curves, etc.). There are 2 main types of image segmentation that fall under Mask R-CNN:

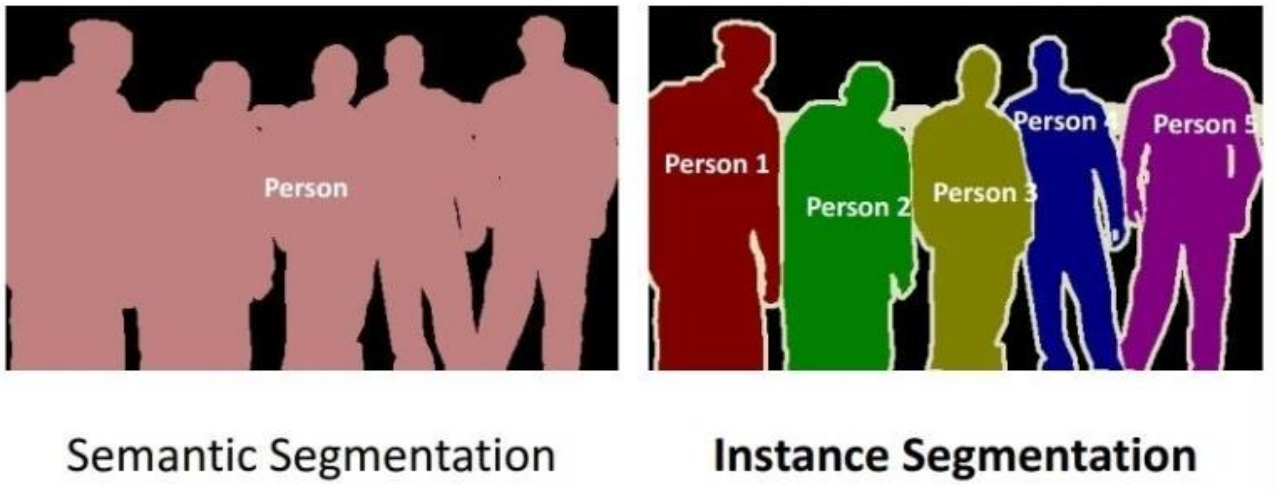
- Semantic Segmentation
- Instance Segmentation

Semantic Segmentation

Semantic segmentation classifies each pixel into a fixed set of categories without differentiating object instances. In other words, semantic segmentation deals with the identification/classification of similar objects as a single class from the pixel level. As shown in the image above, all objects were classified as a single entity (person). Semantic segmentation is otherwise known as background segmentation because it separates the subjects of the image from the background.

Instance Segmentation

Instance Segmentation, or Instance Recognition, deals with the correct detection of all objects in an image while also precisely segmenting each instance. It is, therefore, the combination of object detection, object localization, and object classification. In other words, this type of segmentation goes further to give a clear distinction between each object classified as similar instances.



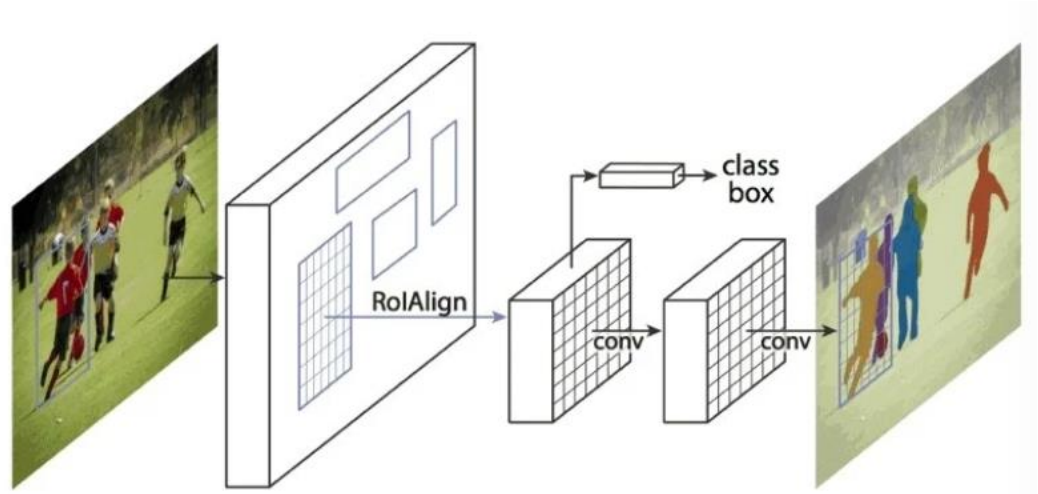
As shown in the example image above, for Instance Segmentation, all objects are persons, but this segmentation process separates each person as a single entity. Semantic segmentation is otherwise known as foreground segmentation because it accentuates the subjects of the image instead of the background.

How does Mask R-CNN work?

Mask R-CNN was built using Faster R-CNN. While Faster R-CNN has 2 outputs for each candidate object, a class label and a bounding-box offset, Mask R-CNN is the addition of a third branch that outputs the object mask. The additional mask output is distinct from the class and box outputs, requiring the extraction of a much finer spatial layout of an object.

Mask R-CNN is an extension of Faster R-CNN and works by adding a branch for predicting an object mask (Region of Interest) in parallel with the existing branch for bounding box recognition.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)



Mask R-CNN – The Mask R-CNN Framework for Instance Segmentation

The key element of Mask R-CNN is the pixel-to-pixel alignment, which is the main missing piece of Fast/Faster R-CNN. Mask R-CNN adopts the same two-stage procedure with an identical first stage (which is RPN). In the second stage, in parallel to predicting the class and box offset, Mask R-CNN also outputs a binary mask for each RoI. This is in contrast to most recent systems, where classification depends on mask predictions.

Furthermore, Mask R-CNN is simple to implement and train given the Faster R-CNN framework, which facilitates a wide range of flexible architecture designs. Additionally, the mask branch only adds a small computational overhead, enabling a fast system and rapid experimentation.

Advantages of Mask R-CNN

- ✓ *Simplicity*: Mask R-CNN is simple to train.
- ✓ *Performance*: Mask R-CNN outperforms all existing, single-model entries on every task.
- ✓ *Efficiency*: The method is very efficient and adds only a small overhead to Faster R-CNN.
- ✓ *Flexibility*: Mask R-CNN is easy to generalize to other tasks. For example, it is possible to use Mask R-CNN for human pose estimation in the same framework.

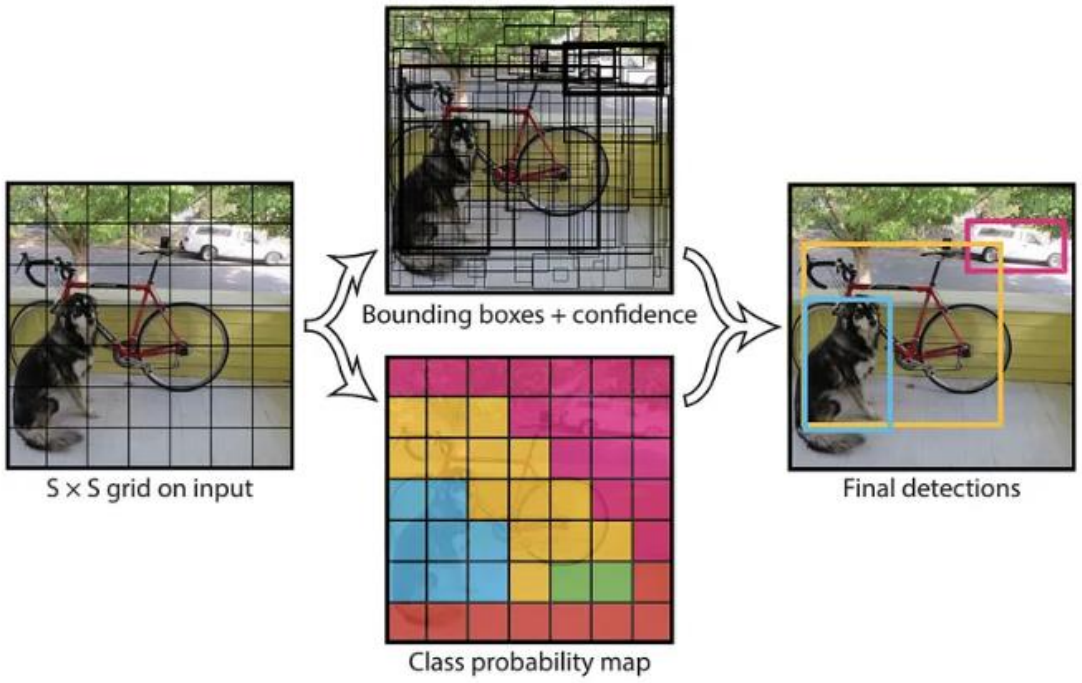
28.YOLO

What is YOLO?

All of the previous object detection algorithms use regions to localize the object within the image. The network does not look at the complete image. Instead, parts of the image which have high probabilities of containing the object. YOLO or You Only Look Once is an object detection algorithm much different from the region-based algorithms seen above. In YOLO a single convolutional network predicts the bounding boxes and the class probabilities for these boxes.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)



YOLO

How YOLO works is that we take an image and split it into an SxS grid, within each of the grid we take m bounding boxes. For each of the bounding box, the network outputs a class probability and offset values for the bounding box. The bounding boxes having the class probability above a threshold value is selected and used to locate the object within the image.

YOLO is orders of magnitude faster (45 frames per second) than other object detection algorithms. The limitation of YOLO algorithm is that it struggles with small objects within the image, for example it might have difficulties in detecting a flock of birds. This is due to the spatial constraints of the algorithm.

Biggest Advantages:

- ✓ Speed (45 frames per second — better than real time)
- ✓ Network understands generalized object representation (This allowed them to train the network on real world images and predictions on artwork was still fairly accurate).
- ✓ Faster version (with smaller architecture) — 155 frames per sec but is less accurate.
- ✓ Open source: https://pjreddie.com/darknet/yolo/

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

UNIT - IV NATURAL LANGUAGE PROCESSING USING RNN (Page No. 117-158)

1.About NLP & its Toolkits Pg. 122

- What is NLP?
- How to Perform NLP?
  - Segmentation
  - Tokenizing
  - Removing Stop Words
  - Stemming
  - Lemmatization
  - Part of Speech Tagging
  - Named Entity Tagging
- Components of NLP
  - Natural Language Understanding (NLU)
  - Natural Language Generation (NLG)
- Why NLP is difficult?
  - Ambiguity
    - Lexical Ambiguity
    - Syntactic Ambiguity
    - Referential Ambiguity
- NLP Toolkits & Libraries
- Phases of NLP
  - Lexical Analysis and Morphological
  - Syntactic Analysis (Parsing)
  - Semantic Analysis
  - Discourse Integration
  - Pragmatic Analysis
- Applications of NLP
- Advantages of NLP
- Disadvantages of NLP

2.Language Modeling Pg. 128

- What is Language Modeling (LM)?
- Why Language Models?
- How Language Modeling works?
  - N-gram
  - Unigram
  - Bidirectional
  - Exponential
  - Continuous space.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

- Uses and Examples of Language Modeling

3.Vector Space Model (VSM) Pg. 131

- What is a Vector Space Model (VSM)?
- Word By Word & Word By Doc
  - Word By Word
  - Word By Doc
- Euclidean Distance
- Cosine Similarity
- Manipulating Words in Vector Space

4.Continuous Bag of Words (CBOW) Pg. 134

- Bag of Words (BOW)
- Continuous Bag of Words (CBOW)
- What is the CBOW Model?
- The Model Architecture
- Points to Remember

5.Skip-Gram Model for Word Embedding Pg. 136

- What is Word Embedding?
- The Skip-Gram Model

CBOW vs Skip-Gram Pg. 138

6.Part of Speech (PoS) Global Co-occurrence Statistics-based Word Vectors Pg. 138

- What is Part-of-Speech (POS) tagging?
- Techniques for POS tagging
- What are Word Vectors?
- What Are Co-occurrence Matrices?
  - Co-occurrence Rules
  - Example of Co-occurrence Matrices

7.Transfer Learning Pg. 141

- Transfer learning in NLP
- What is Model Fine-Tuning?
  - Different Fine-Tuning Techniques

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

15.Gated Recurrent Unit (GRU) Pg. 155

- GRU
- Understanding the GRU Cell
- The Architecture of GRU
- How Does GRU Work?
- Points to Remember

GRU vs LSTM Pg. 158

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

1.About NLP & its Toolkits

What is NLP?

Humans communicate with each other using words and text. The way that humans convey information to each other is called Natural Language. Every day humans share a large quality of information with each other in various languages as speech or text. However, computers cannot interpret this data, which is in natural language, as they communicate in 1s and 0s. The data produced is precious and can offer valuable insights. Hence, you need computers to be able to understand, emulate and respond intelligently to human speech.

Natural Language Processing or NLP refers to the branch of Artificial Intelligence that gives the machines the ability to read, understand and derive meaning from human languages. NLP combines the field of linguistics and computer science to decipher language structure and guidelines and to make models which can comprehend, break down and separate significant details from text and speech.

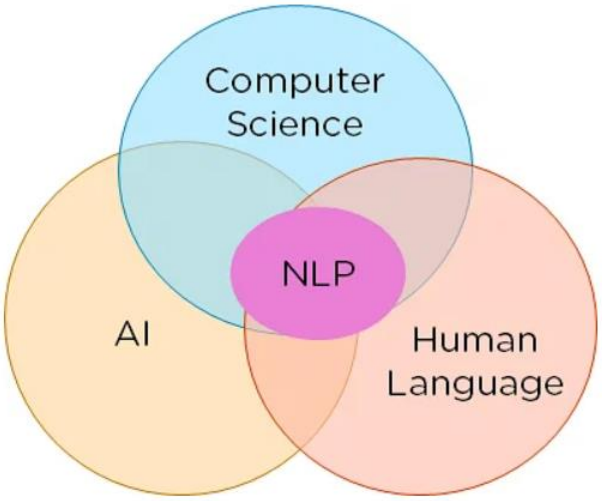


Figure 1: Constituents of NLP

How to Perform NLP?

The steps to perform pre-processing of data in NLP include:

Segmentation:

You first need to break the entire document down into its constituent sentences. You can do this by segmenting the article along with its punctuations like full stops and commas.

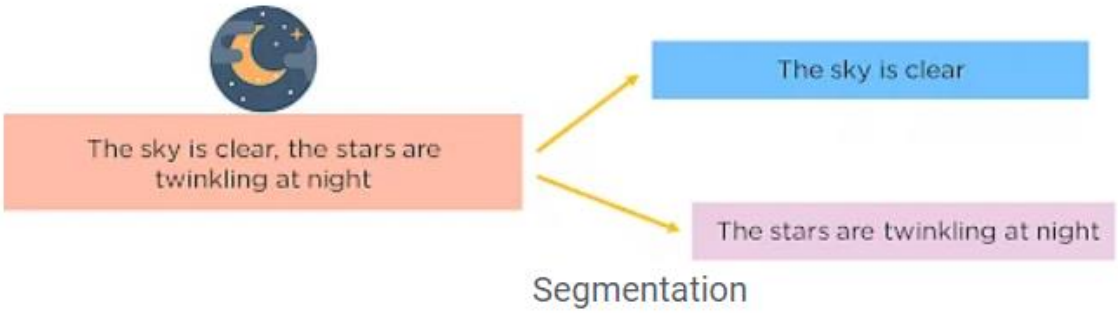


Figure 2:

This document is available free of charge on studocu
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Tokenizing:

For the algorithm to understand these sentences, you need to get the words in a sentence and explain them individually to our algorithm. So, you break down your sentence into its constituent words and store them. This is called tokenizing, and each world is called a token.

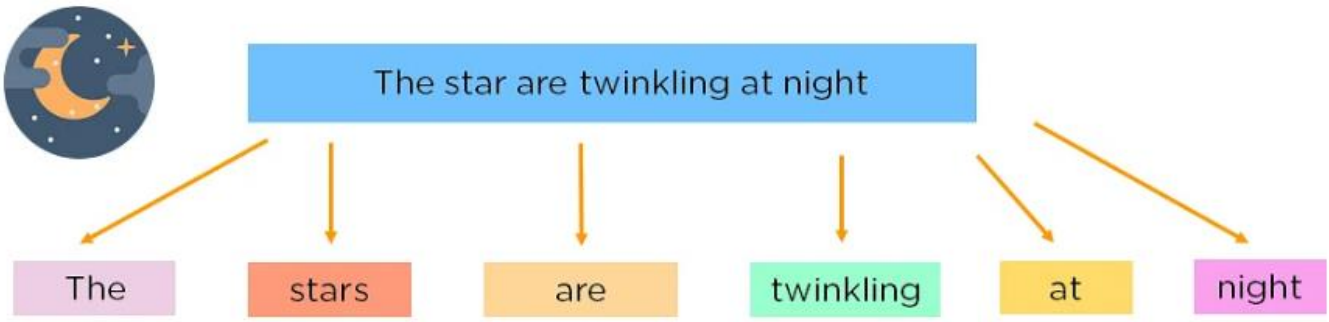


Figure 3: Tokenization

Removing Stop Words:

You can make the learning process faster by getting rid of non-essential words, which add little meaning to our statement and are just there to make our statement sound more cohesive. Words such as was, in, is, and, the, are called stop words and can be removed.

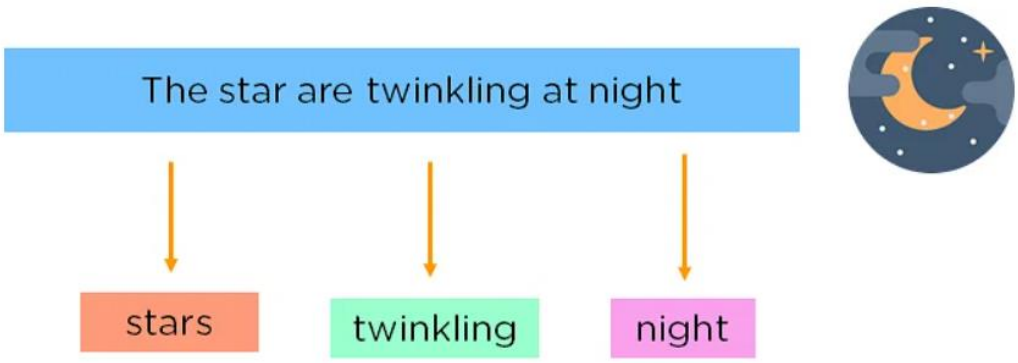


Figure 4: Stop Words

Stemming:

It is the process of obtaining the Word Stem of a word. Word Stem gives new words upon adding affixes to them.

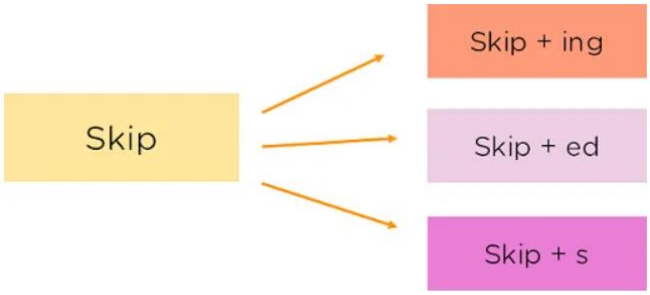


Figure 5: Stemming

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Lemmatization:

The process of obtaining the Root Stem of a word. Root Stem gives the new base form of a word that is present in the dictionary and from which the word is derived. You can also identify the base words for different words based on the tense, mood, gender, etc.

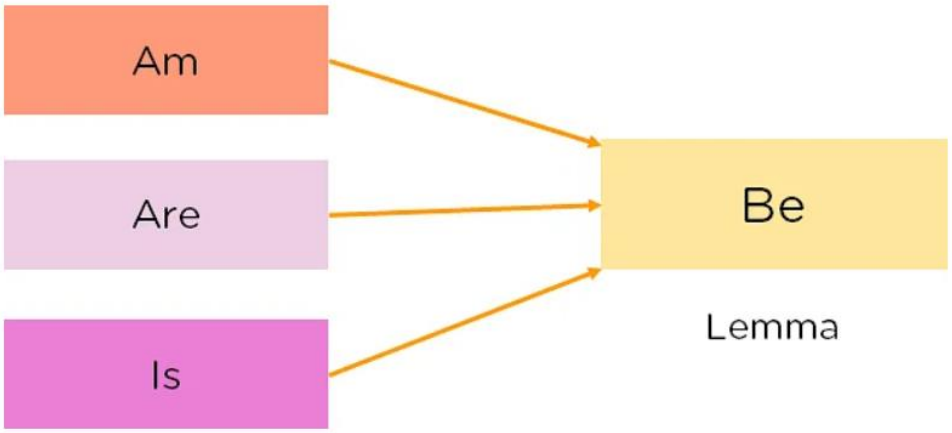


Figure 6: Lemmatization

Part of Speech Tagging:

Now, you must explain the concept of nouns, verbs, articles, and other parts of speech to the machine by adding these tags to our words. This is called 'part of'.

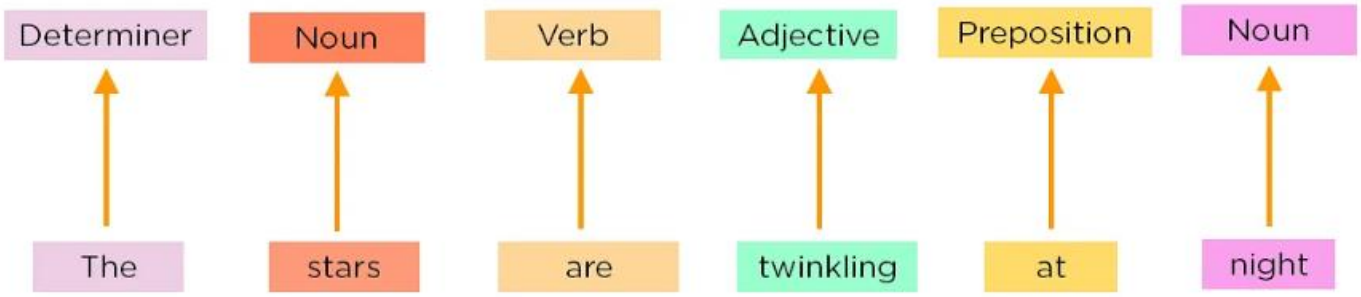


Figure 7: Part of Speech Tagging

Named Entity Tagging:

Next, introduce your machine to pop culture references and everyday names by flagging names of movies, important personalities or locations, etc that may occur in the document. You do this by classifying the words into subcategories. This helps you find any keywords in a sentence. The subcategories are person, location, monetary value, quantity, organization, movie.

After performing the pre-processing steps, you then give your resultant data to a machine learning algorithm like Naive Bayes, etc., to create your NLP application.

Components of NLP

There are the following two components of NLP -

1. Natural Language Understanding (NLU)

Natural Language Understanding (NLU) helps the machine to understand and analyse human language by extracting the metadata from content such as concepts, entities, keywords, emotion,

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

relations, and semantic roles. NLU mainly used in Business applications to understand the customer's problem in both spoken and written language.

NLU involves the following tasks -

- It is used to map the given input into useful representation.
- It is used to analyse different aspects of the language.

2. Natural Language Generation (NLG)

Natural Language Generation (NLG) acts as a translator that converts the computerized data into natural language representation. It mainly involves Text planning, Sentence planning, and Text Realization.

Difference between NLU and NLG

NLU	NLG
NLU is the process of reading and interpreting language.	NLG is the process of writing or generating language.
It produces non-linguistic outputs from natural language inputs.	It produces constructing natural language outputs from non-linguistic inputs.

Why NLP is difficult?

NLP is difficult because Ambiguity and Uncertainty exist in the language.

Ambiguity

There are the following three ambiguity -

- *Lexical Ambiguity* - Lexical Ambiguity exists in the presence of two or more possible meanings of the sentence within a single word. Example: Manya is looking for a match.
  In the above example, the word match refers to that either Manya is looking for a partner or Manya is looking for a match. (Cricket or another match)

- *Syntactic Ambiguity* - Syntactic Ambiguity exists in the presence of two or more possible meanings within the sentence. Example: I saw the girl with the binocular.
  In the above example, did I have the binoculars? Or did the girl have the binoculars?

- *Referential Ambiguity* - Referential Ambiguity exists when you are referring to something using the pronoun. Example: Kiran went to Sunita. She said, "I am hungry." In the above sentence, you do not know that who is hungry, either Kiran or Sunita.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

NLP Toolkits & Libraries

- *Scikit-learn:* It provides a wide range of algorithms for building machine learning models in Python.

- *Natural language Toolkit (NLTK):* NLTK is a complete toolkit for all NLP techniques. It enables users to create Python programs that are compatible with human language data. NLTK has user-friendly interfaces to more than 50 lexical and corpora resources, as well as various text processing libraries and a robust discussion forum. This free, open-source platform is commonly used by educators, students, linguists, engineers, and researchers.

- *Pattern:* It is a web mining module for NLP and machine learning.

- *TextBlob:* A Python library that functions as an extension of NLTK. When using this intuitive interface, beginners can easily perform tasks like part-of-speech tagging, text classification, and sentiment analysis. This library tends to be more accessible to those who are new to NLP than other libraries. It provides an easy interface to learn basic NLP tasks like sentiment analysis, noun phrase extraction, or pos-tagging.

- *Quepy:* Quepy is used to transform natural language questions into queries in a database query language.

- *IBM Watson*: Offers users a range of AI-based services, each of which is stored in the IBM cloud. This versatile suite is well-suited to perform Natural Language Understanding tasks, such as identifying keywords, emotions, and categories. IBM Watson's versatility lends itself to use in a range of industries, such as finance and healthcare.

- *SpaCy:* SpaCy is an open-source NLP library which is used for Data Extraction, Data Analysis, Sentiment Analysis, and Text Summarization. It is one of the newer open-source NLP processing libraries. This Python library performs quickly and is well-documented. It is able to handle large datasets and provides users with a plethora of pre-trained NLP models. SpaCy is geared toward those who are getting text ready for deep learning or extraction.

- *Google Cloud Natural Language API*: Part of Google Cloud. It incorporates question-answering technology, as well as language understanding technology. This interface offers users a variety of pre-trained models that can be used for performing entity extraction, content classification, and sentiment analysis.

- *Gensim:* Gensim is a high-speed, scalable Python library that focuses primarily on topic modelling tasks. It excels at recognizing the similarities between texts, as well as navigating various documents and indexing texts. One of the main benefits of using Gensim is that it can handle huge data volumes. It works with large datasets and processes data streams.

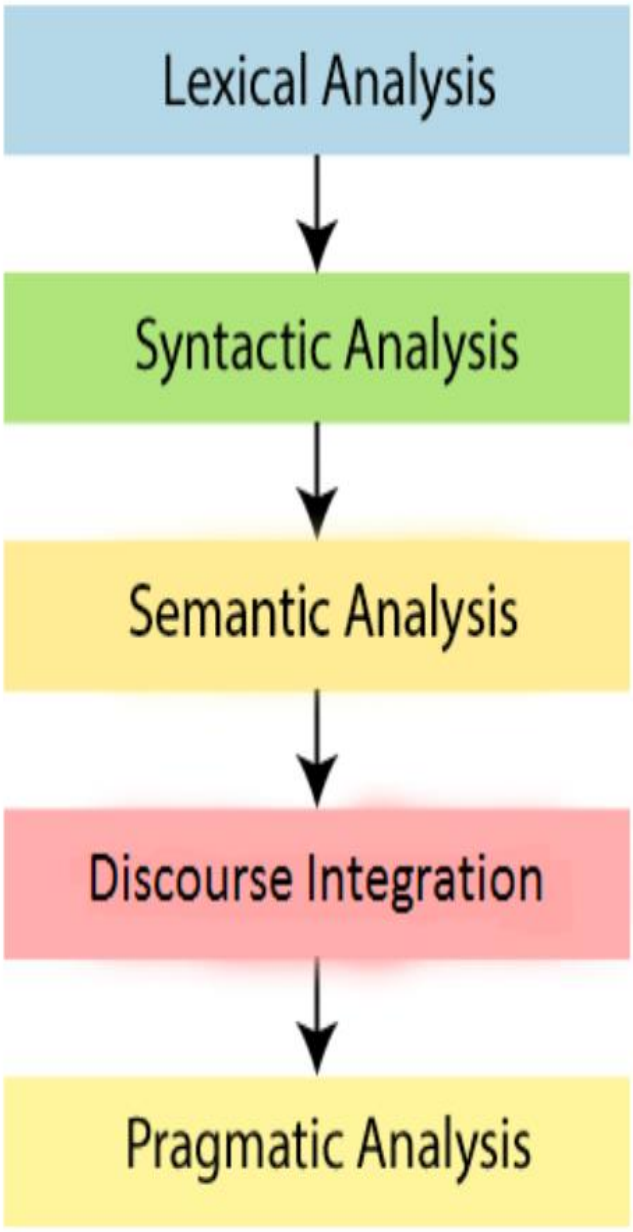
- *MonkeyLearn*: NLP-powered platform that provides users with a means for gathering insights from text data. This user-friendly platform offers pre-trained models that can perform topic classification, keyword extraction, and sentiment analysis, as well as customized machine learning models that can be changed to meet various business needs. MonkeyLearn can also connect to apps like Excel or Google Sheets to perform text analysis.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Phases of NLP

There are the following five phases of NLP:



1. Lexical Analysis and Morphological

The first phase of NLP is the Lexical Analysis. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. It divides the whole text into paragraphs, sentences, and words.

2. Syntactic Analysis (Parsing)

Syntactic Analysis is used to check grammar, word arrangements, and shows the relationship among the words.

Example: Agra goes to the Poonam

In the real world, Agra goes to the Poonam, does not make any sense, so this sentence is rejected by the Syntactic analyser.

3. Semantic Analysis

Semantic analysis is concerned with the meaning representation. It mainly focuses on the literal meaning of words, phrases, and sentences.

4. Discourse Integration

Discourse Integration depends upon the sentences that proceeds it and also invokes the meaning of the sentences that follow it.

5. Pragmatic Analysis

Pragmatic is the fifth and last phase of NLP. It helps you to discover the intended effect by applying a set of rules that characterize cooperative dialogues.

For Example: "Open the door" is interpreted as a request instead of an order.

Applications of NLP

NLP is one of the ways that people have humanized machines and reduced the need for labour. It has led to the automation of speech-related tasks and human interaction. Some applications of NLP include:

- ✓ *Translation Tools:* Tools such as Google Translate, Amazon Translate, etc. translate sentences from one language to another using NLP.
- ✓ *Chatbots:* Chatbots can be found on most websites and are a way for companies to deal with common queries quickly.
- ✓ *Virtual Assistants:* Virtual Assistants like Siri, Cortana, Google Home, Alexa, etc can not only talk to you but understand commands given to them.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

- ✓ *Targeted Advertising:* Have you ever talked about a product or service or just googled something and then started seeing ads for it? This is called targeted advertising, and it helps generate tons of revenue for sellers as they can reach niche audiences at the right time.
- ✓ *Autocorrect:* Autocorrect will automatically correct any spelling mistakes you make, apart from this grammar checkers also come into the picture which helps you write flawlessly.

Advantages of NLP

- NLP helps users to ask questions about any subject and get a direct response within seconds.
- NLP offers exact answers to the question means it does not offer unnecessary and unwanted information.
- NLP helps computers to communicate with humans in their languages.
- It is very time efficient.
- Most of the companies use NLP to improve the efficiency of documentation processes, accuracy of documentation, and identify the information from large databases.

Disadvantages of NLP

- NLP may not show context.
- NLP is unpredictable.
- NLP may require more keystrokes.
- NLP is unable to adapt to the new domain, and it has a limited function that's why NLP is built for a single and specific task only.

2.Language Modeling

What is Language Modeling (LM)?

Language modeling (LM) is the use of various statistical and probabilistic techniques to determine the probability of a given sequence of words occurring in a sentence. Language models analyse bodies of text data to provide a basis for their word predictions. They are used in natural language processing (NLP) applications, particularly ones that generate text as an output. Some of these applications include, machine translation and question answering.

A language model in NLP is a probabilistic statistical model that determines the probability of a given sequence of words occurring in a sentence based on the previous words. It helps to predict which word is more likely to appear next in the sentence. Hence it is widely used in predictive text input systems, speech recognition, machine translation, spelling correction etc. The input to a language model is usually a training set of example sentences. The output is a probability distribution over sequences of words. We can use the last one word (unigram), last two words (bigram), last three words (trigram) or last n words (n-gram) to predict the next word as per our requirements.

Why Language Models?

Language models form the backbone of Natural Language Processing. They are a way of transforming qualitative information about text into quantitative information that machines can understand. They have applications in a wide range of industries like tech, finance, healthcare, military etc. All of us encounter language models daily, be it the predictive text input on our mobile phones or a simple Google search. Hence language models form an integral part of any natural language processing application.

How Language Modeling works?

Language models determine word probability by analysing text data. They interpret this data by feeding it through an algorithm that establishes rules for context in natural language. Then, the model

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

applies these rules in language tasks to accurately predict or produce new sentences. The model essentially learns the features and characteristics of basic language and uses those features to understand new phrases.

There are several different probabilistic approaches to modeling language, which vary depending on the purpose of the language model. From a technical perspective, the various types differ by the amount of text data they analyse and the math they use to analyse it. For example, a language model designed to generate sentences for an automated Twitter bot may use different math and analyse text data in a different way than a language model designed for determining the likelihood of a search query.

Some common statistical language modeling types are:

N-gram. N-grams are a relatively simple approach to language models. They create a probability distribution for a sequence of n The n can be any number, and defines the size of the "gram", or sequence of words being assigned a probability. For example, if n = 5, a gram might look like this: "can you please call me." The model then assigns probabilities using sequences of n size. Basically, n can be thought of as the amount of context the model is told to consider. Some types of n-grams are unigrams, bigrams, trigrams and so on.

Unigram. The unigram is the simplest type of language model. It doesn't look at any conditioning context in its calculations. It evaluates each word or term independently. Unigram models commonly handle language processing tasks such as information retrieval. The unigram is the foundation of a more specific model variant called the query likelihood model, which uses information retrieval to examine a pool of documents and match the most relevant one to a specific query.

Bidirectional. Unlike n-gram models, which analyse text in one direction (backwards), bidirectional models analyse text in both directions, backwards and forwards. These models can predict any word in a sentence or body of text by using every other word in the text. Examining text bidirectionally increases result accuracy. This type is often utilized in machine learning and speech generation applications. For example, Google uses a bidirectional model to process search queries.

Exponential. Also known as maximum entropy models, this type is more complex than n-grams. Simply put, the model evaluates text using an equation that combines feature functions and n-grams. Basically, this type specifies features and parameters of the desired results, and unlike n-grams, leaves analysis parameters more ambiguous -- it doesn't specify individual gram sizes, for example. The model is based on the principle of entropy, which states that the probability distribution with the most entropy is the best choice. In other words, the model with the most chaos, and least room for assumptions, is the most accurate. Exponential models are designed maximize cross entropy, which minimizes the amount statistical assumptions that can be made. This enables users to better trust the results they get from these models.

Continuous space. This type of model represents words as a non-linear combination of weights in a neural network. The process of assigning a weight to a word is also known as word embedding. This type becomes especially useful as data sets get increasingly large, because larger datasets often include more unique words. The presence of a lot of unique or rarely used words can cause problems for linear model like an n-gram. This is because the amount of possible word sequences increases, and the patterns that inform results become weaker. By weighting words in a non-linear, distributed way, this model can "learn" to approximate words and therefore not be misled by any unknown values. Its

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

"understanding" of a given word is not as tightly tethered to the immediate surrounding words as it is in n-gram models.

The models listed above are more general statistical approaches from which more specific variant language models are derived. For example, as mentioned in the n-gram description, the query likelihood model is a more specific or specialized model that uses the n-gram approach. Model types may be used in conjunction with one another. The models listed also vary significantly in complexity. Broadly speaking, more complex language models are better at NLP tasks, because language itself is extremely complex and always evolving. Therefore, an exponential model or continuous space model might be better than an n-gram for NLP tasks, because they are designed to account for ambiguity and variation in language.

A good language model should also be able to process long-term dependencies, handling words that may derive their meaning from other words that occur in far-away, disparate parts of the text. An LM should be able to understand when a word is referencing another word from a long distance, as opposed to always relying on proximal words within a certain fixed history. This requires a more complex model.

Uses and Examples of Language Modeling

Language models are the backbone of natural language processing (NLP). Below are some NLP tasks that use language modeling, what they mean, and some applications of those tasks:

- ✓ *Speech recognition* -- involves a machine being able to process speech audio. This is commonly used by voice assistants like Siri and Alexa.
- ✓ *Machine translation* -- involves the translation of one language to another by a machine. Google Translate and Microsoft Translator are two programs that do this. SDL Government is another, which is used to translate foreign social media feeds in real time for the U.S. government.
- ✓ *Parts-of-speech tagging* -- involves the markup and categorization of words by certain grammatical characteristics. This is utilized in the study of linguistics, first and perhaps most famously in the study of the Brown Corpus, a body of composed of random English prose that was designed to be studied by computers. This corpus has been used to train several important language models, including one used by Google to improve search quality.
- ✓ *Parsing* -- involves analysis of any string of data or sentence that conforms to formal grammar and syntax rules. In language modeling, this may take the form of sentence diagrams that depict each word's relationship to the others. Spell checking applications use language modeling and parsing.
- ✓ *Sentiment analysis* -- involves determining the sentiment behind a given phrase. Specifically, it can be used to understand opinions and attitudes expressed in a text. Businesses can use this to analyse product reviews or general posts about their product, as well as analyse internal data like employee surveys and customer support chats. Some services that provide sentiment analysis tools are Repustate and Hubspot's ServiceHub. Google's NLP tool -- called Bidirectional Encoder Representations from Transformers (BERT) -- is also used for sentiment analysis.
- ✓ *Optical character recognition* -- involves the use of a machine to convert images of text into machine encoded text. The image may be a scanned document or document photo, or a photo with text somewhere in it -- on a sign, for example. It is often used in data entry when processing old paper records that need to be digitized. In can also be used to analyse and identify handwriting samples.
- ✓ *Information retrieval* -- involves searching in a document for information, searching for documents in general, and searching for metadata that corresponds to a document. Web browsers are the most common information retrieval applications.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

3.Vector Space Model (VSM)

What is a Vector Space Model (VSM)?

Vector space models are algebraic models that are often used to represent text (although they can represent any object) as a vector of identifiers. With these models, we are able to identify whether various texts are similar in meaning, regardless of whether they share the same words.

There are numerous instances we may decide to employ a vector spaced model, for instance:

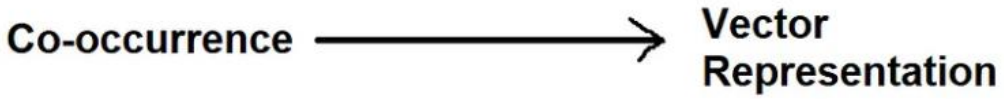
- ✓ Information Filtering
- ✓ Information Retrieval
- ✓ Machine Translation
- ✓ Chatbots

And many more!

In general, Vector space models allow us to represent words and documents as vectors. Vector space models are to consider the relationship between data that are represented by vectors. It is popular in information retrieval systems but also useful for other purposes. Generally, this allows us to compare the similarity of two vectors from a geometric perspective.

Word By Word & Word By Doc

For us to represent our text as vectors we may decide to use a word-by-word or word-by-doc design. Performing this task involves first creating a co-occurrence matrix. Although how we perform each task is quite similar, we will discuss each design one at a time, nonetheless, the objective is the same. We want to go from our co-occurrence matrix to a vector representation.



Word By Word: This design counts the number of times words occur within a certain distance k.

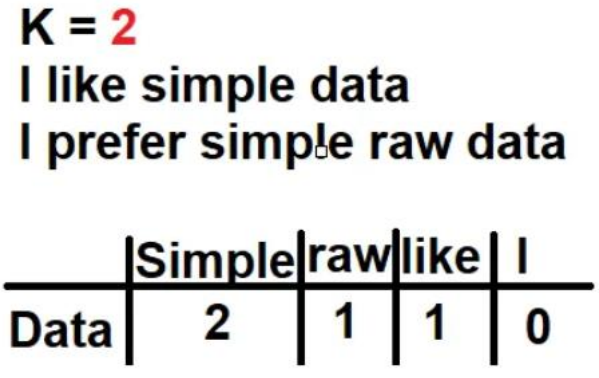


Figure 3: Example of word by word co-occurrence matrix where k=2 (Image By Author)

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

In the word-by-word design, the co-occurrence matrix is between 1 and N entries.

Word By Doc: The number of times words from the vocabulary appear in documents that belong to certain categories.

	Entertainment	Economy	Machine Learning
Data	500	6620	9320
Film	7000	4000	1000

Figure 4: Example of word by doc co-occurrence matrix (Image By Author)

Using these vector representations, we can now represent our text or documents in vector space. This is perfect because in vector space we can determine the relationships between types of documents, such as their similarity.

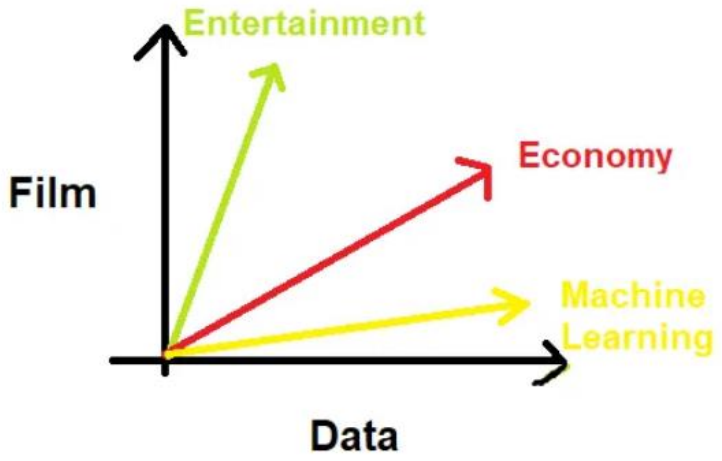


Figure 5: Representing word by doc in vector space (Image By Author)

Euclidean Distance

A similarity metric we may use to determine how far apart 2 vectors are from one another is the Euclidean distances, which is merely the length of a straight line that connects 2 vectors.

$$Euc(A,B) = \sqrt{\sum_{i=1}^{n}(A_i - B_i)^2}$$

Figure 6: Formula for Euclidean Distance (Image By Author)

Let's use the formula in Figure 6 to calculate the more similar documents using our vector representations from Figure 5.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

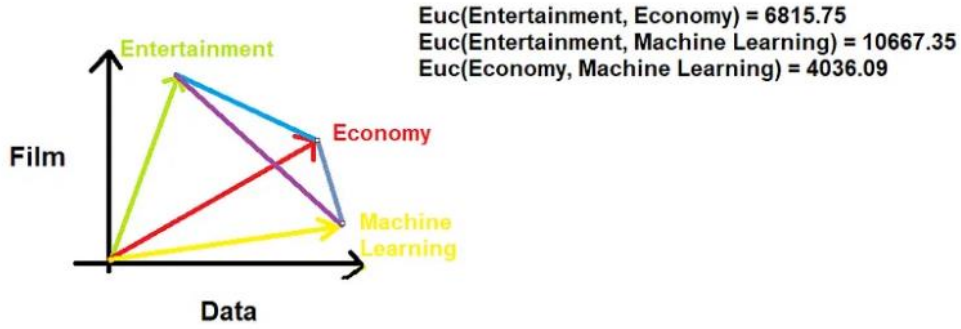


Figure 7: Calculating the Euclidean distances (Image By Author)

The results tell us that the economy and Machine learning documents are more similar since distance-based metrics prioritize objects with lower values to detect similarity. With that being said, it is important to note that the Euclidean distance is not scale-invariant and it's often recommended to scale your data.

Cosine Similarity

The problem with the Euclidean distance is that it is biased by size difference in representations. Therefore, we may decide to use the cosine similarity which would determine how similar text is using the inner angle.

Vector Norm = $||\vec{V}|| = \sqrt{\sum_{i=1}^{n} v_i^2}$

Dot Product = $\vec{V}.\vec{W} = ||\vec{V}||||\vec{W}||cos(\beta)$

Therefore,

$$Cos(\beta) = \frac{\vec{V}.\vec{W}}{||\vec{V}||||\vec{W}||}$$

Figure 8: Formula for Cosine similarity (Image By Author)

Cosine similarity is one of the most popular similarity metrics used in NLP. To calculate similarity, we take the cosine similarity of an angle between two vectors.

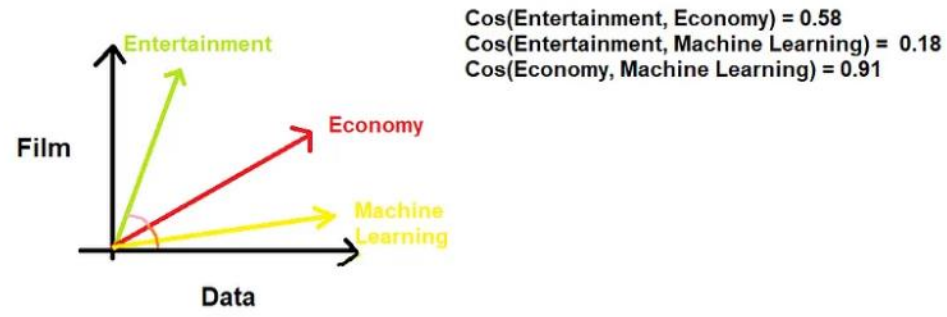


Figure 9: Calculating cosine similarities (Image By Author)

When the cosine value is equal to 0 this means the two vectors are orthogonal to one another and have no match. Whereas, a cosine value closer to 1 would imply that there is a greater match between the

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

two values (since the angles are smaller). Therefore, from our results, Economy and Machine Learning are the most similar.

Manipulating Words in Vector Space

By performing some simple vector arithmetic, we are able to infer unknown representations among words. For instance, if we know the relationship between two similar such as King and Man. In order to find the vector representation of the word "Queen", we can add the vector representation we retrieved from determining the relationship between King and Man (we retrieve this vector by subtracting the vectors i.e., King - Man) to the vector representation of Woman and inferring that the most similar vector representation (which would be Queen in this instance) is the vector we wanted to find.

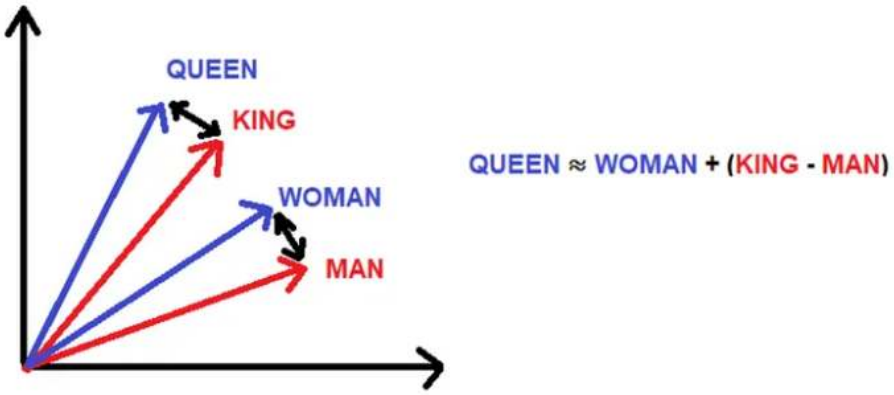


Figure 10: Visual Representation (Image by Author)

In conclusion, we may use vector space models to represent our text or documents in vector space, and when our data is in vector space, we can use the vectors to determine the relationships between text (or documents).

4.Continuous Bag of Words (CBOW)

Bag of Words (BOW)

Bag of Words (BOW) is an approach for dealing with words and context in text processing or information retrieval. It refers to a way in which a group of words are represented without retaining order. Consider the below sentence.

"I had a great time today."

The BOW representations of the sentence would be the collection of the six words without order.

Continuous Bag of Words (CBOW)

CBOW is always with respect to a word in the sequence. Let us consider the word 'great' as the word of focus from the above line. In CBOW you consider other words in the line based on their position w.r.t to the focus word.

Words before i.e. "I", "had" and "a" are considered history words.

Words after i.e., "time" and "today" are considered future words.

So, history Bag of Words for 'great' is ("I", "had", "a") without order and future Bag of Words for is ("time", "today") also without order.

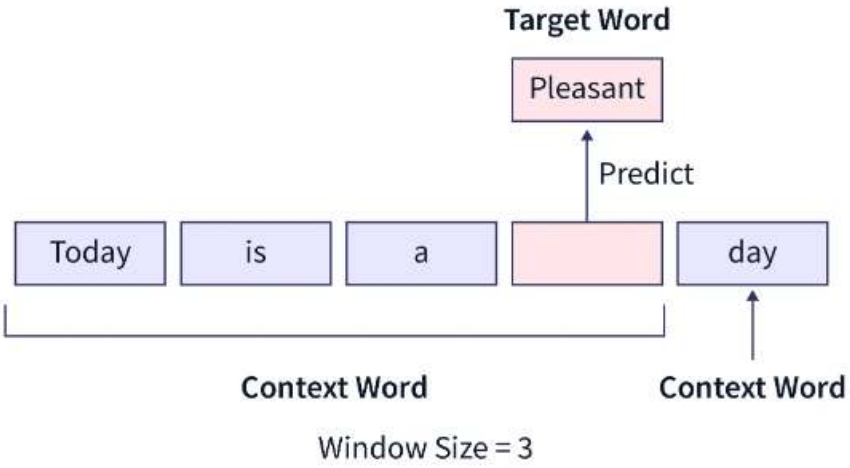
CBOW is an architecture to train Word Embedding (vector representations of words, specifically Word2Vec).

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

What is the CBOW Model?

The CBOW model tries to understand the context of the words and takes this as input. It then tries to predict words that are contextually accurate. Let us consider an example for understanding this. Consider the sentence: 'It is a pleasant day' and the word 'pleasant' goes as input to the neural network. We are trying to predict the word 'day' here. We will use the one-hot encoding for the input words and measure the error rates with the one-hot encoded target word. Doing this will help us predict the output based on the word with least error.



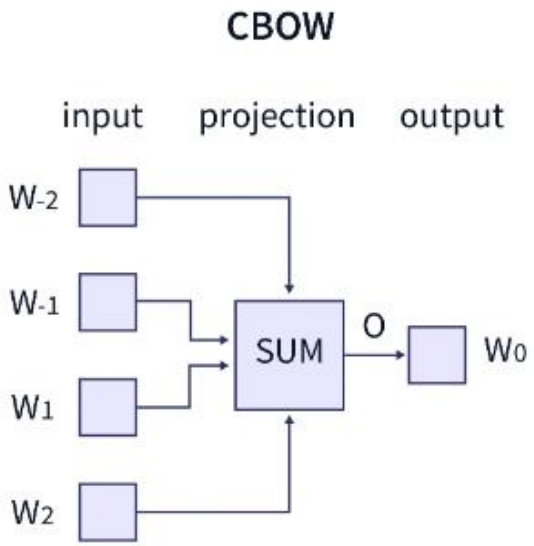
The continuous bag-of-words (CBOW) model is a neural network for natural language processing tasks such as language translation and text classification. It is based on predicting a target word given the context of the surrounding words. The CBOW model takes a window of surrounding words as input and tries to predict the target word in the centre of the window. The model is trained on a large text dataset and learns to predict the target word based on the patterns it observes in the input data. The CBOW model is often combined with other natural language processing techniques and models, such as the skip-gram model, to improve the performance of natural language processing tasks.

The Model Architecture

CBOW (Continuous Bag of Words) model predicts a target word based on the context of the surrounding words in a sentence or text. It is trained using a feedforward neural network where the input is a set of context words, and the output is the target word. The model learns to adjust the weights of the neurons in the hidden layer to produce an output that is most likely to be the target word.

The embeddings learned by the model can be used for a variety of NLP tasks such as text classification, language translation, and sentiment analysis. It's fast and efficient in learning the representation of context words and predict the target word in one go, which is also known as one pass learning. Its method is widely used in learning word embeddings, where the goal is to learn a dense representation of words, where semantically similar words are close to each other in the embedding space.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)



The CBOW model architecture is as shown above. The model tries to predict the target word by trying to understand the context of the surrounding words. Consider the same sentence as above, 'It is a pleasant day'. The model converts this sentence into word pairs in the form (contextword, targetword). The user will have to set the window size. If the window for the context word is 2 then the word pairs would look like this: ([it, a], is), ([is, pleasant], a), ([a, day], pleasant). With these word pairs, the model tries to predict the target word considered the context words.

If we have 4 context words used for predicting one target word the input layer will be in the form of four 1XW input vectors. These input vectors will be passed to the hidden layer where it is multiplied by a WXN matrix. Finally, the 1XN output from the hidden layer enters the sum layer where an element-wise summation is performed on the vectors before a final activation is performed and the output is obtained.

Points to Remember

- The continuous bag-of-words (CBOW) model is a neural network for natural language processing tasks such as translation and text classification.
- It is based on predicting a target word given the context of the surrounding words.
- CBOW models are trained on a large dataset of text and adjust the weights and biases of the neurons during training to minimize the error between the predicted target word embedding and the true target word embedding.
- In conclusion, the CBOW model is a powerful tool for natural language processing tasks and can be easily implemented using the Gensim library in python.
- By understanding the principles of the CBOW model and how to use it with Gensim, you can improve your knowledge of this topic and apply it to a wide range of applications.

5.Skip-Gram Model for Word Embedding

In Natural Language Processing, we want computers to understand the text as we humans do. However, for this to happen, we need them to translate the words to a language computer can work with and understand.

What is Word Embedding?

Word Embedding is a numerical representation of words, such as how colours can be represented using the RGB system. In natural language processing, word embedding is a term used to

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

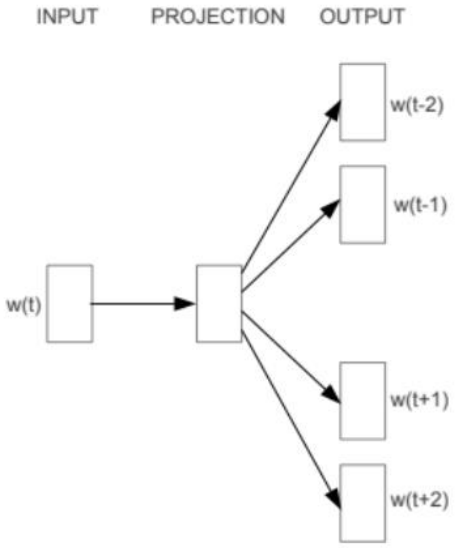
represent words for text analysis, typically in real-valued vectors that encode the word's meaning. The words that are closed in vector space are expected to have a similar meaning. Word embedding uses language modeling and feature learning techniques where words from the vocabulary are mapped to vectors of real numbers.

Let's take an example, text = "The match between India and New-Zealand delayed due to rain"
From the above text, we can form the dictionary of unique words as following.
['The','match','between','India','and','New-Zealand','delayed', 'due', 'to', 'rain'] and the vector representation of the word can be one-hot encoded vector. The vector representation of the word 'India' according to the above vocabulary can be seen as [0,0,0,1,0,0,0,0,0,0].

If you try to visualize these vectors, we can think of a 10-dimensional space where each word occupies one dimension and has no relation with other vectors. Here comes the idea of distributed representation, introducing some dependence of one word to another word. In one hot encoding representation, all the words are independent of each other. There are different word embedding techniques such as Count-Vectorizer, TFIDF-Vectorizer, Continuous Bag Of Word and Skip-Gram.

The Skip-Gram Model

The Skip-Gram model architecture usually tries to achieve the reverse of what the CBOW model does. It tries to predict the source context words (surrounding words) given a target word (the centre word). Considering our simple sentence, "the quick brown fox jumps over the lazy dog". If we used the CBOW model, we get pairs of (context_window, target_word) where if we consider a context window of size 2, we have examples like ([quick, fox], brown), ([the, brown], quick), ([the, dog], lazy) and so on. Now considering that the skip-gram model's aim is to predict the context from the target word, the model typically inverts the contexts and targets, and tries to predict each context word from its target word. Hence the task becomes to predict the context [quick, fox] given target word 'brown' or [the, brown] given target word 'quick' and so on. Thus, the model tries to predict the context_window words based on the target_word.



Skip-gram

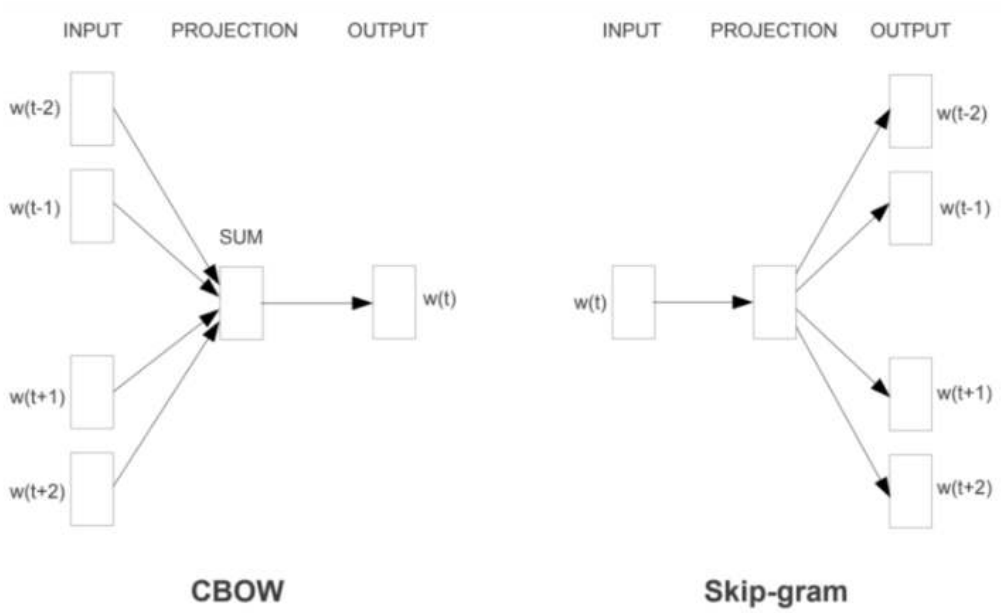
Just like we discussed in the CBOW model, we need to model this Skip-gram architecture now as a deep learning classification model such that we take in the target word as our input and try to predict the context words. This becomes slightly complex since we have multiple words in our context. We simplify this further by breaking down each (target, context_words) pair into (target, context) pairs

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

such that each context consists of only one word. Hence our dataset from earlier gets transformed into pairs like (brown, quick), (brown, fox), (quick, the), (quick, brown) and so on.

To supervise or train the model to know what is contextual and what is not, we feed our skip-gram model pairs of (X, Y) where X is our input and Y is our label. We do this by using [(target, context), 1] pairs as positive input samples where target is our word of interest and context is a context word occurring near the target word and the positive label 1 indicates this is a contextually relevant pair. We also feed in [(target, random), 0] pairs as negative input samples where target is again our word of interest but random is just a randomly selected word from our vocabulary which has no context or association with our target word. Hence the negative label 0indicates this is a contextually irrelevant pair. We do this so that the model can then learn which pairs of words are contextually relevant and which are not and generate similar embeddings for semantically similar words.

CBOW vs Skip-Gram



According to the original paper, Mikolov et al., it is found that Skip-Gram works well with small datasets, and can better represent less frequent words. However, CBOW is found to train faster than Skip-Gram, and can better represent more frequent words. Of course, which model we choose largely depends on the problem we're trying to solve. Is it important for our model to represent rare words? If so, we should choose Skip-Gram. We don't have much time to train and rare words are not that important for our solution? Then we should choose CBOW.

6.Part of Speech (PoS) Global Co-occurrence Statistics–based Word Vectors

What is Part-of-Speech (POS) tagging?

POS Tagging (Parts of Speech Tagging) is a process to mark up the words in text format for a particular part of a speech based on its definition and context. This can include nouns, verbs, adjectives, and other grammatical categories. It is responsible for text reading in a language and assigning some specific token (Parts of Speech) to each word. It is also called grammatical tagging.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Let's take an example,

Text: "The cat sat on the mat."

POS tags:

- The: determiner
- cat: noun
- sat: verb
- on: preposition
- the: determiner
- mat: noun

POS tagging is useful for a variety of NLP tasks, such as information extraction, named entity recognition, and machine translation. It can also be used to identify the grammatical structure of a sentence and to disambiguate words that have multiple meanings. POS tagging is typically performed using machine learning algorithms, which are trained on a large annotated corpus of text. The algorithm learns to predict the correct POS tag for a given word based on the context in which it appears. There are various POS tagging schemes that have been developed, each with its own set of tags and rules.

In the above example, each word in the sentence has been labelled with its corresponding part of speech. The determiner "the" is used to identify specific nouns, while the noun "cat" refers to a specific animal. The verb "sat" describes an action, and the preposition "on" describes the relationship between the cat and the mat.

Techniques for POS tagging

There are various techniques that can be used for POS tagging such as

- *Rule-based POS tagging*: The rule-based POS tagging models apply a set of handwritten rules and use contextual information to assign POS tags to words. These rules are often known as context frame rules. One such rule might be: "If an ambiguous/unknown word ends with the suffix 'ing' and is preceded by a Verb, label it as a Verb".

- *Transformation Based Tagging*: The transformation-based approaches use a pre-defined set of handcrafted rules as well as automatically induced rules that are generated during training.

- *Deep learning models*: Various Deep learning models have been used for POS tagging such as Meta-BiLSTM which have shown an impressive accuracy of around 97 percent.

- *Stochastic (Probabilistic) tagging*: A stochastic approach includes frequency, probability or statistics. The simplest stochastic approach finds out the most frequently used tag for a specific word in the annotated training data and uses this information to tag that word in the unannotated text. But sometimes this approach comes up with sequences of tags for sentences that are not acceptable according to the grammar rules of a language. One such approach is to calculate the probabilities of various tag sequences that are possible for a sentence and assign the POS tags from the sequence with the highest probability. Hidden Markov Models (HMMs) are probabilistic approaches to assign a POS Tag.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

What are Word Vectors?

Word Vectors (also called Word Embeddings) are a type of word representation that allows words with similar meanings to have an equal representation. In simple terms: A word vector is a vector representation of a particular word. Words that appear in similar contexts will have similar vectors. For example, vectors for "leopard", "lion", and "tiger" will be close together, while they'll be far away from "planet" and "castle".

There are two main classes of methods to find word embeddings. The first set are count-based and rely on matrix factorization (e.g., LSA, HAL). While these methods effectively leverage global statistical information, they are primarily used to capture word similarities and do poorly on tasks such as word analogy, indicating a sub-optimal vector space structure. The other set of methods are shallow window-based (e.g., the skip-gram and the CBOW models), which learn word embeddings by making predictions in local context windows. These models demonstrate the capacity to capture complex linguistic patterns beyond word similarity, but fail to make use of the global co-occurrence statistics.

In comparison, GloVe consists of a weighted least squares model that trains on global word-word co-occurrence counts and thus makes efficient use of statistics. The model produces a word vector space with meaningful sub-structure. It shows state-of-the-art performance on the word analogy task, and outperforms other current methods on several word similarity tasks.

In conclusion, the GloVe model efficiently leverages global statistical information by training only on the nonzero elements in a word-word co-occurrence matrix, and produces a vector space with meaningful sub-structure. It consistently outperforms word2vec on the word analogy task, given the same corpus, vocabulary, window size, and training time. It achieves better results faster, and also obtains the best results irrespective of speed.

What Are Co-occurrence Matrices?

Co-occurrence matrices are a fundamental concept in NLP, and we can use them to represent the relationship between elements in a text corpus. Usually, in NLP, we work with a collection of text or text corpus. Elements of text corpus can refer to sentences, words, phrases, or any other linguistic unit of interest. With co-occurrence matrices, it is possible to represent these elements using rows and columns of a matrix. More precisely, each row and column of a matrix represents a unique element of a text corpus. Cells of the matrix represent the number of times two elements appear together in a predefined context. The context can be a document, sentence, word window, or any other relevant unit.

Co-occurrence Rules

Co-occurrence rules enable you to discover and group concepts that are strongly related within the set of documents or records. The idea is that when concepts are often found together in documents and records, that co-occurrence reflects an underlying relationship that is probably of value in your category definitions. This technique creates co-occurrence rules that can be used to create a new category, extend a category, or as input to another category technique. Two concepts strongly co-occur if they frequently appear together in a set of records and rarely separately in any of the other records. This technique can produce good results with larger datasets with at least several hundred documents or records.

Example of Co-occurrence Matrices

As an example, let's use three sentences below:

Apples are green and red.
Red apples are sweet.
Green oranges are sour.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Let's assume that these three sentences are our text corpus, elements are words and their context is one sentence. It means that for each pair of two words from the sentences above, we need to count how many times they appear together in one sentence. For example, the words "apples" and "red" appear two times, in the first and second sentences, while the words "red" and "sour" don't appear in the same sentence.
Following that logic, our co-occurrence matrix will look like this:

-	apples	are	green	and	red	sweet	oranges	sour
apples	2	2	1	1	2	1	0	0
are	2	3	1	1	2	1	1	1
green	1	1	2	1	1	0	1	1
and	1	1	1	1	1	0	0	0
red	2	2	1	1	2	1	0	0
sweet	1	1	0	0	1	1	0	0
oranges	0	1	1	0	0	0	1	1
sour	0	1	1	0	0	0	1	1

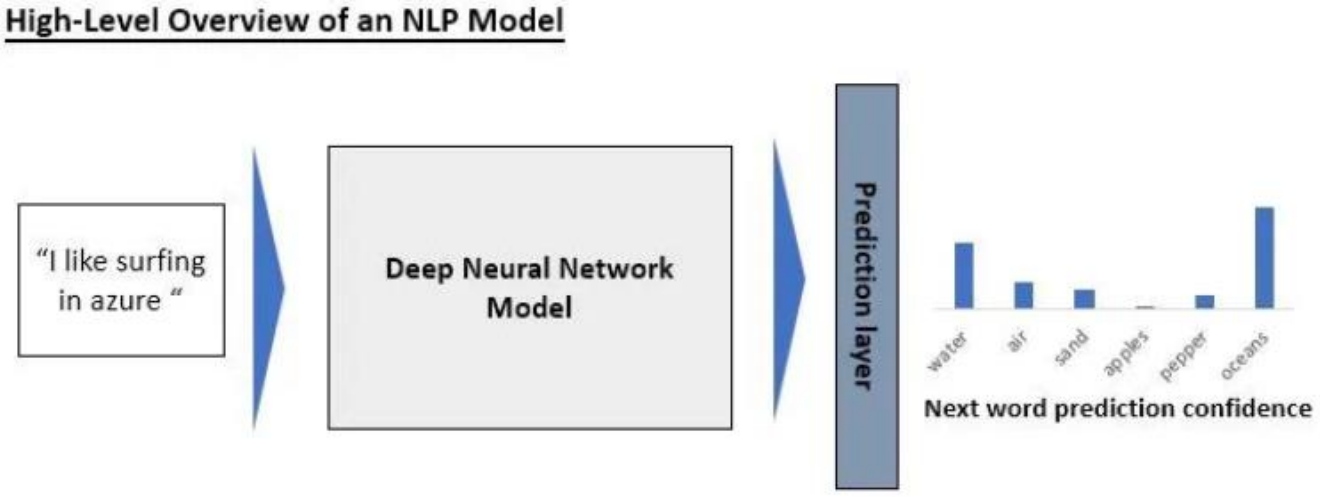
From the table above, we can notice that the co-occurrence matrix is symmetric. It means that the value with row X and column Y will be the same as the value with row Y and column X. In general, we don't need to keep all elements from the text corpus in the co-occurrence matrix but only those of interest. For instance, before creating a matrix, it would be useful to clean text, remove stop words, implement stemming and lemmatization, and similar.

7.Transfer Learning

Transfer learning in NLP

Transfer learning is a technique where a deep learning model trained on a large dataset is used to perform similar tasks on another dataset. We call such a deep learning model a pre-trained model. The most renowned examples of pre-trained models are the computer vision deep learning models trained on the ImageNet dataset. So, it is better to use a pre-trained model as a starting point to solve a problem rather than building a model from scratch.

The process of taking a model that's been trained to do one task — as these pre-trained models have been trained to do — and then fine-tuning it to work on a related but different task is at the essence of what's called Transfer Learning. It's been used in computer vision for just over half a decade and has been making waves in NLP for about three years.



Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

After converting words into a numerical form machine learning models can understand, these are fed into the main part of the model which is (most often) a deep, multi-layered neural network. The most popular language models at the moment, the Transformer, have a structure where they build a very deep set of relationships between every word in the sentence, creating an incredibly rich (numerical) representation of a sentence. This rich representation is then fed into the last layer of the model which, given a part of the sentence, is trained to predict the next word. It does this by giving its confidence of predicting what the next word is over all of the words in its vocabulary. In the example in the image above, the model is most confident that the next word is 'oceans'.

One of the fascinating things with Transfer Learning in NLP was that researchers discovered that when you train a model to predict the next word, you can take the trained model, chop off the layer that predicts the next word and put on a new layer and train just that last layer — very quickly — to predict the sentiment of a sentence.

The team over at *HuggingFace* have done an incredible job of making these models incredibly easy to use. You can basically load up these pre-trained models with just a couple lines of code and start experimenting with the possibilities.

What is Model Fine-Tuning?

BERT (Bidirectional Encoder Representations from Transformers) is a big neural network architecture, with a huge number of parameters, that can range from 100 million to over 300 million. It is designed to pre-train deep bidirectional representations from unlabelled text by jointly conditioning on both left and right context. So, training a BERT model from scratch on a small dataset would result in overfitting. So, it is better to use a pre-trained BERT model that was trained on a huge dataset, as a starting point. We can then further train the model on our relatively smaller dataset and this process is known as model fine-tuning. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of NLP tasks."

Different Fine-Tuning Techniques

- *Train the entire architecture* – We can further train the entire pre-trained model on our dataset and feed the output to a softmax layer. In this case, the error is back-propagated through the entire architecture and the pre-trained weights of the model are updated based on the new dataset.
- *Train some layers while freezing others* – Another way to use a pre-trained model is to train it partially. What we can do is keep the weights of initial layers of the model frozen while we retrain only the higher layers. We can try and test as to how many layers to be frozen and how many to be trained.
- *Freeze the entire architecture* – We can even freeze all the layers of the model and attach a few neural network layers of our own and train this new model. Note that the weights of only the attached layers will be updated during model training.

8.Word2Vec

Word Embedding is a word representation type that allows machine learning algorithms to understand words with similar meanings. It is a language modeling and feature learning technique to map words into vectors of real numbers using neural networks, probabilistic models, or dimension reduction on the word co-occurrence matrix. Some word embedding models are Word2vec (Google), Glove (Stanford), and fastest (Facebook). Word Embedding is also called as distributed semantic model or distributed represented or semantic vector space or vector space model.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

What is Word2Vec?

Word2Vec is a technique/model to produce word embedding for better word representation. It is a natural language processing method that captures a large number of precise syntactic and semantic word relationships. It is a shallow two-layered neural network that can detect synonymous words and suggest additional words for partial sentences once it is trained.

The shallow neural network consists of the only a hidden layer between input and output whereas deep neural network contains multiple hidden layers between input and output. Input is subjected to nodes whereas the hidden layer, as well as the output layer, contains neurons.

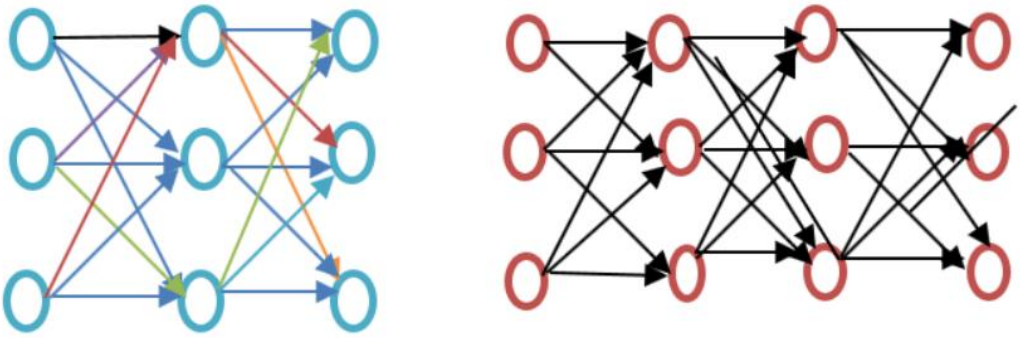


Figure: Shallow vs. Deep learning

Word2Vec is a two-layer network where there is input one hidden layer and output. Word2vec was developed by a group of researchers headed by Tomas Mikolov at Google. Word2vec is better and more efficient that latent semantic analysis model.

Why Word2Vec?

Word2Vec represents words in vector space representation. Words are represented in the form of vectors and placement is done in such a way that similar meaning words appear together and dissimilar words are located far away. This is also termed as a semantic relationship. Neural networks do not understand text instead they understand only numbers. Word Embedding provides a way to convert text to a numeric vector. Word2Vec reconstructs the linguistic context of words.

What is Linguistic Context?

In general life scenario when we speak or write to communicate, other people try to figure out what is objective of the sentence. For example, “What is the temperature of India”, here the context is the user wants to know “temperature of India” which is context. In short, the main objective of a sentence is context. Word or sentence surrounding spoken or written language (disclosure) helps in determining the meaning of context. Word2Vec learns vector representation of words through the contexts.

How Word2Vec works?

Word2Vec learns word by predicting its surrounding context. For example, let us take the word “He loves Football.” We want to calculate the Word2vec for the word: loves. Suppose

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

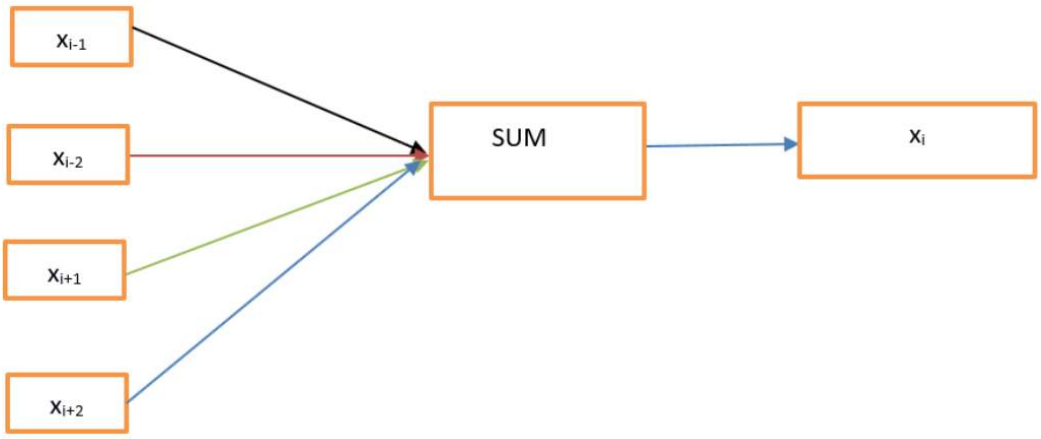
loves = V_{in}. $P(V_{out} / V_{in})$ is calculated
where,
V_{in} is the input word.
P is the probability of likelihood.
V_{out} is the output word.

Word *loves* moves over each word in the corpus. Syntactic as well as the Semantic relationship between words is encoded. This helps in finding similar and analogies words. All random features of the word *loves* are calculated. These features are changed or update concerning neighbour or context words with the help of a Back Propagation method. Another way of learning is that if the context of two words is similar or two words have similar features, then such words are related.

Word2Vec Architecture

There are two architectures used by Word2Vec:

- **Continuous Bag of Words (CBOW)**
  Let us draw a simple Word2vec example diagram to understand the continuous bag of word architecture.



Let us calculate the equations mathematically. Suppose V is the vocabulary size and N is the hidden layer size. Input is defined as { xi-1, xi-2, xi+1, xi+2}. We obtain the weight matrix by multiplying V * N. Another matrix is obtained by multiplying input vector with the weight matrix. This can also be understood by the following equation.

$$h=xitW$$

where xit and W are the input vector and weight matrix respectively,
To calculate the match between context and the next word, please refer to the below equation

$$u=predictedrepresentation*h$$

where predictedrepresentation is obtained model h in the above equation.

- **Skip-Gram**
  Skip-Gram approach is used to predict a sentence given an input word. To understand it better let us draw the diagram as shown in the below Word2vec example.

This document is available free of charge on studocu
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

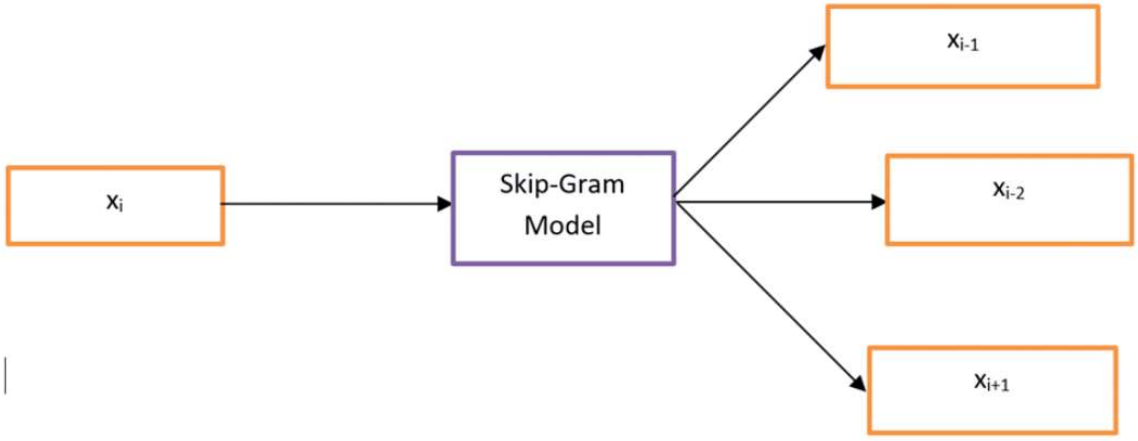


Figure Skip-Gram Model

One can treat it as the reverse of the Continuous bag of word model where the input is the word and model provide the context or the sequence. We can also conclude that the target is fed to the input and output layer is replicated multiple times to accommodate the chosen number of context words. Error vector from all the output layer is summed up to adjust weights via a backpropagation method.

Learning word representation is essentially unsupervised, but targets/labels are needed to train the model. Skip-gram and CBOW convert unsupervised representation to supervised form for model training. In CBOW, the current word is predicted using the window of surrounding context windows. For example, if wi-1, wi-2, wi+1, wi+2are given words or context, this model will provide wi. Skip-Gram performs opposite of CBOW which implies that it predicts the given sequence or context from the word. You can reverse the example to understand it. If wi is given, this will predict the context or wi-1, wi-2, wi+1, wi+2.

Word2vec provides an option to choose between CBOW (continuous Bag of words) and skim-gram. Such parameters are provided during training of the model. One can have the option of using negative sampling or hierarchical softmax layer.

Which model to choose?

CBOW is several times faster than skip gram and provides a better frequency for frequent words whereas skip gram needs a small amount of training data and represents even rare words or phrases.

The General Flow of the Algorithm

- *Step-1*: Initially, we will assign a vector of random numbers to each word in the corpus.
- *Step-2*: Then, we will iterate through each word of the document and grab the vectors of the nearest n-words on either side of our target word, and concatenate all these vectors, and then forward propagate these concatenated vectors through a linear layer + softmax function, and try to predict what our target word was.
- *Step-3*: In this step, we will compute the error between our estimate and the actual target word and then backpropagated the error and then modifies not only the weights of the linear layer but also the vectors or embeddings of our neighbour's words.
- *Step-4*: Finally, we will extract the weights from the hidden layer and by using these weights encode the meaning of words in the vocabulary.

Word2Vec model is not a single algorithm but is composed of the following two pre-processing modules or techniques:

- Continuous Bag of Words (CBOW)
- Skip-Gram.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

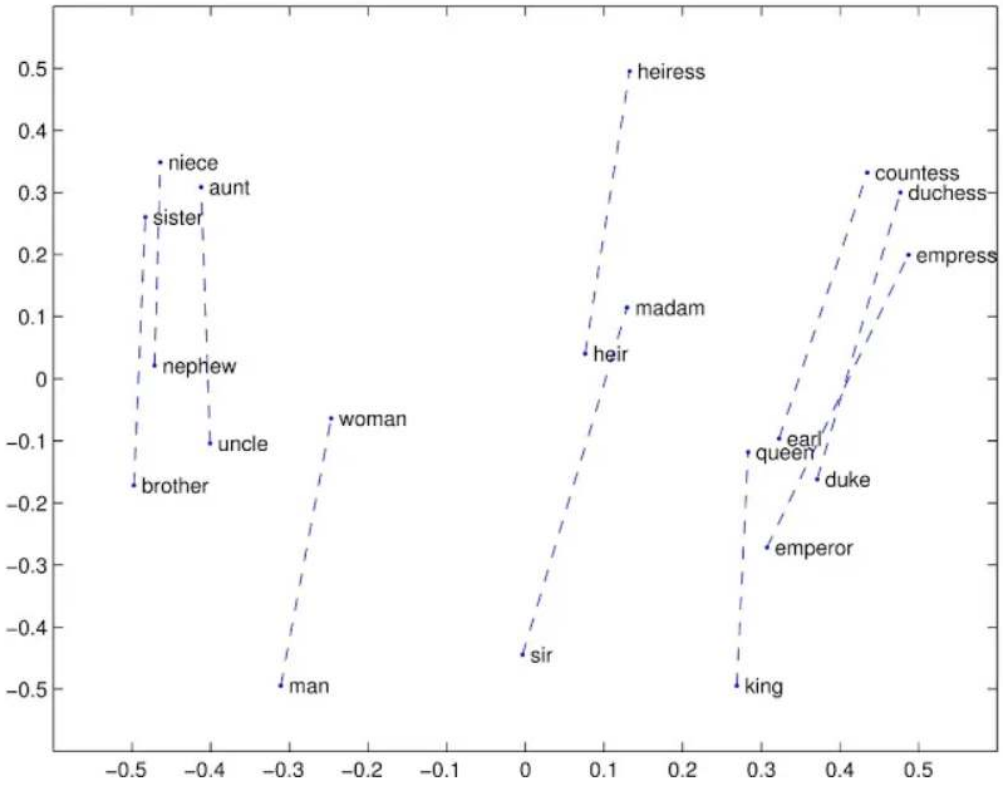
Both of the mentioned models are basically shallow neural networks that map word(s) to the target variable which is also a word(s). These techniques learn the weights that act as word vector representations. Both these techniques can be used to implementing word embedding using word2vec.

9.Global Vectors for Word Representation GloVe

GloVe is a word vector technique that rode the wave of word vectors after a brief silence. Just to refresh, word vectors put words to a nice vector space, where similar words cluster together and different words repel. The advantage of GloVe is that, unlike Word2vec, GloVe does not rely just on local statistics (local context information of words), but incorporates global statistics (word co-occurrence) to obtain word vectors.

GloVe stands for Global Vectors for word representation. It is an unsupervised learning algorithm developed by researchers at Stanford University aiming to generate word embeddings by aggregating global word co-occurrence matrices from a given corpus. The resulting embeddings show interesting linear substructures of the word in vector space.

Examples for linear substructures are:



These results are pretty impressive. This type of representation can be very useful in many machine learning algorithms. The basic idea behind the GloVe word embedding is to derive the relationship between the words from statistics. Unlike the occurrence matrix, the co-occurrence matrix tells you how often a particular word pair occurs together. Each value in the co-occurrence matrix represents a pair of words occurring together.

Given a corpus having V words, the co-occurrence matrix X will be a V x V matrix, where the i^{th} row and j^{th} column of X, X_ij denotes how many times word i has co-occurred with word j. An example co-occurrence matrix might look as follows.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

	the	cat	sat	on	mat
the	0	1	0	1	1
cat	1	0	1	0	0
sat	0	1	0	1	0
on	1	0	1	0	0
mat	1	0	0	0	0

The co-occurrence matrix for the sentence "the cat sat on the mat" with a window size of 1. As you probably noticed it is a symmetric matrix.

Word2vec relies only on local information of language. That is, the semantics learnt for a given word, is only affected by the surrounding words. GloVe captures both global statistics and local statistics of a corpus, in order to come up with word vectors.

10.Backpropagation Through Time

What is Backpropagation?

Backpropagation is a training algorithm that is used for training neural networks. When training a neural network, we are actually tuning the weights of the network to minimize the error with respect to the already available true values(labels) by using the Backpropagation algorithm. It is a supervised learning algorithm as we find errors with respect to already given labels. The general algorithm is as follows:

1. Present a training input pattern and propagate it through the network to get an output.
2. Compare the predicted outputs to the expected outputs and calculate the error.
3. Calculate the derivatives of the error with respect to the network weights.
4. Use these calculated derivatives to adjust the weights to minimize the error.
5. Repeat.

Backpropagation Through Time

Backpropagation Through Time, or BPTT, is the application of the Backpropagation training algorithm to recurrent neural network applied to sequence data like a time series. A recurrent neural network is shown one input each timestep and predicts one output. Conceptually, BPTT works by unrolling all input timesteps. Each timestep has one input timestep, one copy of the network, and one output. Errors are then calculated and accumulated for each timestep. The network is rolled back up and the weights are updated.

Spatially, each timestep of the unrolled recurrent neural network may be seen as an additional layer given the order dependence of the problem and the internal state from the previous timestep is taken as an input on the subsequent timestep. We can summarize the algorithm as follows:

1. Present a sequence of timesteps of input and output pairs to the network.
2. Unroll the network then calculate and accumulate errors across each timestep.
3. Roll-up the network and update weights.
4. Repeat.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

BPTT can be computationally expensive as the number of timesteps increases. If input sequences are comprised of thousands of timesteps, then this will be the number of derivatives required for a single update weight update. This can cause weights to vanish or explode (go to zero or overflow) and make slow learning and model skill noisy.

11.Bidirectional RNNs (BRNN)

What are Bidirectional Recurrent Neural Networks?

Bidirectional recurrent neural networks (BRNN) connect two hidden layers running in opposite directions to a single output, allowing them to receive information from both past and future states. This generative deep learning technique is more common in supervised learning approaches, rather than unsupervised or semi-supervised because how difficult it is to calculate a reliable probabilistic model.

A regular recurrent neural network (RNN) is extended to a bidirectional recurrent neural network (BRNN), by training it simultaneously in positive and negative time direction.

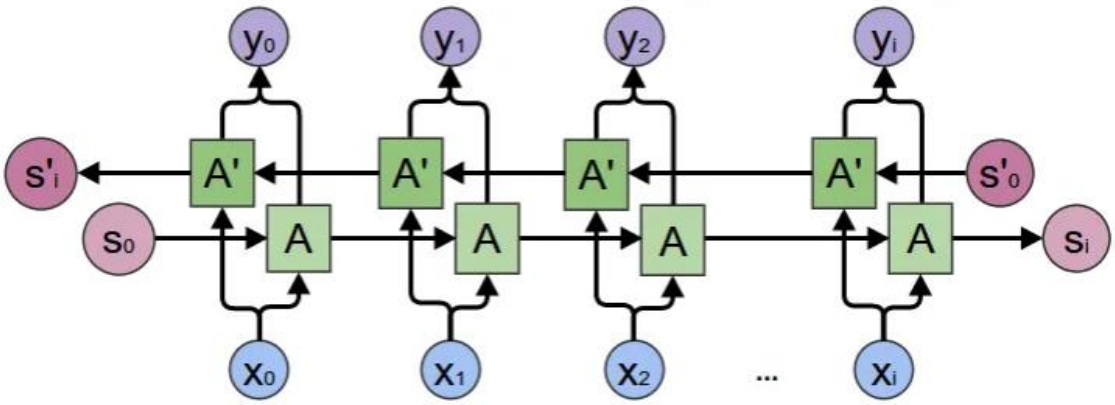


Fig 1: General Structure of Bidirectional Recurrent Neural Networks. Source: colah's blog

Bidirectional recurrent neural networks (RNN) are really just putting two independent RNNs together. The input sequence is fed in normal time order for one network, and in reverse time order for another. The outputs of the two networks are usually concatenated at each time step, though there are other options, e.g., summation. This structure allows the networks to have both backward and forward information about the sequence at every time step.

How are BRNNs Trained?

BRNNs are trained with similar algorithms as RNNs, since the two directional neurons do not interact with one another. If back-propagation is necessary, some additional process is needed, since input and output layers cannot both be updated at once.

In general training, forward and backward states are processed first in the "forward" pass, before output neurons are passed. For the backward pass, the opposite takes place; output neurons are processed first, then forward and backward states are passed next. Weights are updated only after the forward and backward passes are complete.

What's the Difference Between BRNN's and Recurrent Neural Networks?

Unlike standard recurrent neural networks, BRNN's are trained to predict both the positive and negative directions of time simultaneously. BRNN's split the neurons of a regular RNN into two directions, one for the forward states (positive time direction), and another for the backward states (negative time direction) Neither of these output states are connected to inputs of the opposite

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

directions. By employing two-time directions simultaneously, input data from the past and future of the current time frame can be used to calculate the same output. Which is the opposite of standard recurrent networks that requires an extra layer for including future information.

12.Long Short-Term Model (LSTM)

What is Long Short-Term Model (LSTM)?

LSTM stands for long short-term memory networks, used in the field of Deep Learning. It is a variety of recurrent neural networks (RNNs) that are capable of learning long-term dependencies, especially in sequence prediction problems. LSTM has feedback connections, i.e., it is capable of processing the entire sequence of data, apart from single data points such as images. This finds application in speech recognition, machine translation, etc. LSTM is a special kind of RNN, which shows outstanding performance on a large variety of problems.

LSTM can be implemented in Python using the Keras library. Let's say while watching a video, you remember the previous scene, or while reading a book, you know what happened in the earlier chapter. RNNs work similarly; they remember the previous information and use it for processing the current input. The shortcoming of RNN is they cannot remember long-term dependencies due to vanishing gradient. LSTMs are explicitly designed to avoid long-term dependency problems.

Difference between LSTM and Gated Recurrent Unit (GRU)

LSTM and GRU are both variants of RNN that are used to resolve the vanishing gradient issue of the RNN, but they have some differences, which are:

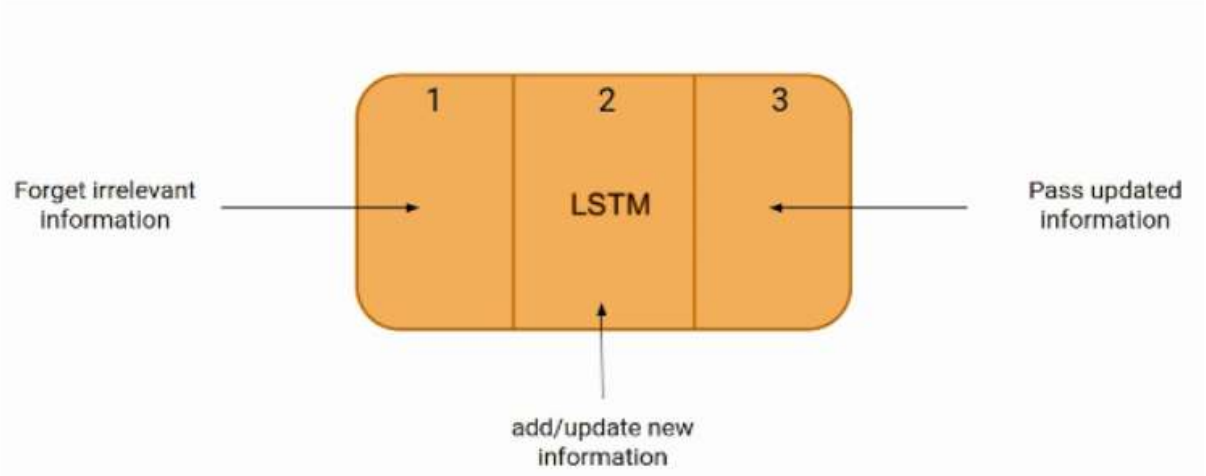
→LSTM uses three gates to compute the input of sequence data where, whereas GRU uses only two gates.

→ GRUs are generally simpler and faster than LSTM.

→LSTMs are preferred for large datasets, whereas for small datasets GRUs are preferred.

LSTM Architecture

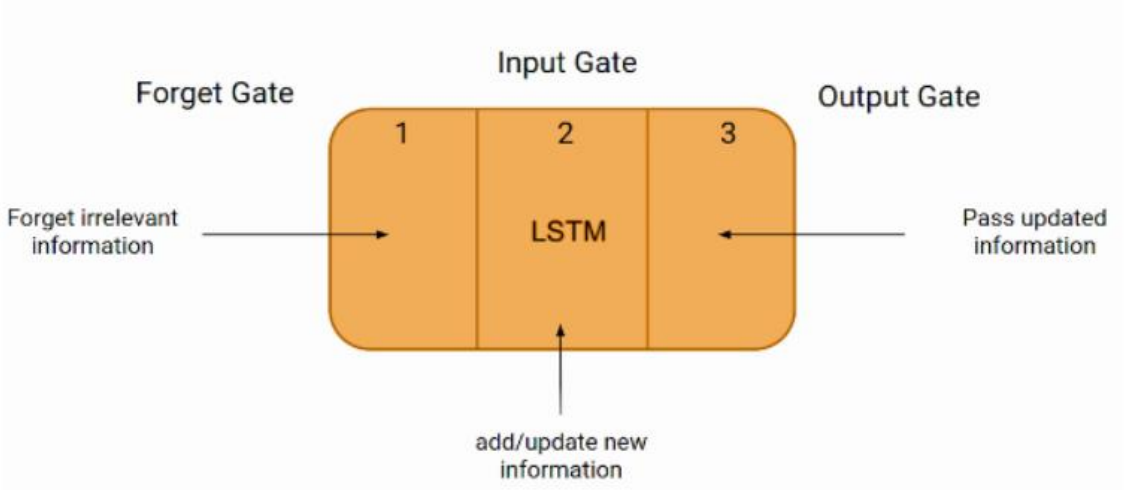
At a high level, LSTM works very much like an RNN cell. Here is the internal functioning of the LSTM network. The LSTM network architecture consists of three parts, as shown in the image below, and each part performs an individual function.



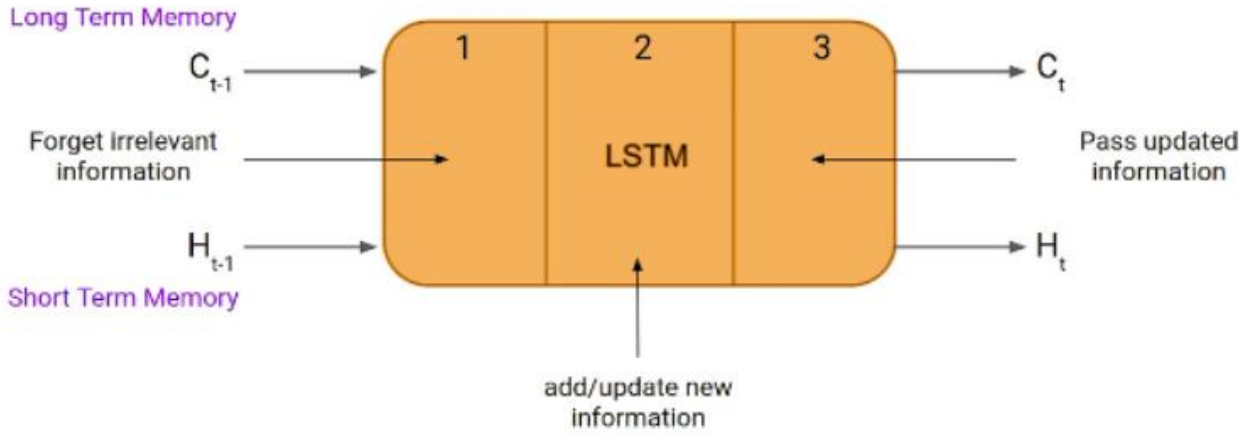
The first part chooses whether the information coming from the previous timestamp is to be remembered or is irrelevant and can be forgotten. In the second part, the cell tries to learn new

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

information from the input to this cell. At last, in the third part, the cell passes the updated information from the current timestamp to the next timestamp. This one cycle of LSTM is considered a single-time step.



These three parts of an LSTM unit are known as gates. They control the flow of information in and out of the memory cell or LSTM cell. The first gate is called Forget gate, the second gate is known as the Input gate, and the last one is the Output gate. An LSTM unit that consists of these three gates and a memory cell or LSTM cell can be considered as a layer of neurons in traditional feedforward neural network, with each neuron having a hidden layer and a current state. These gates can be thought of as filters and are each their own neural network.



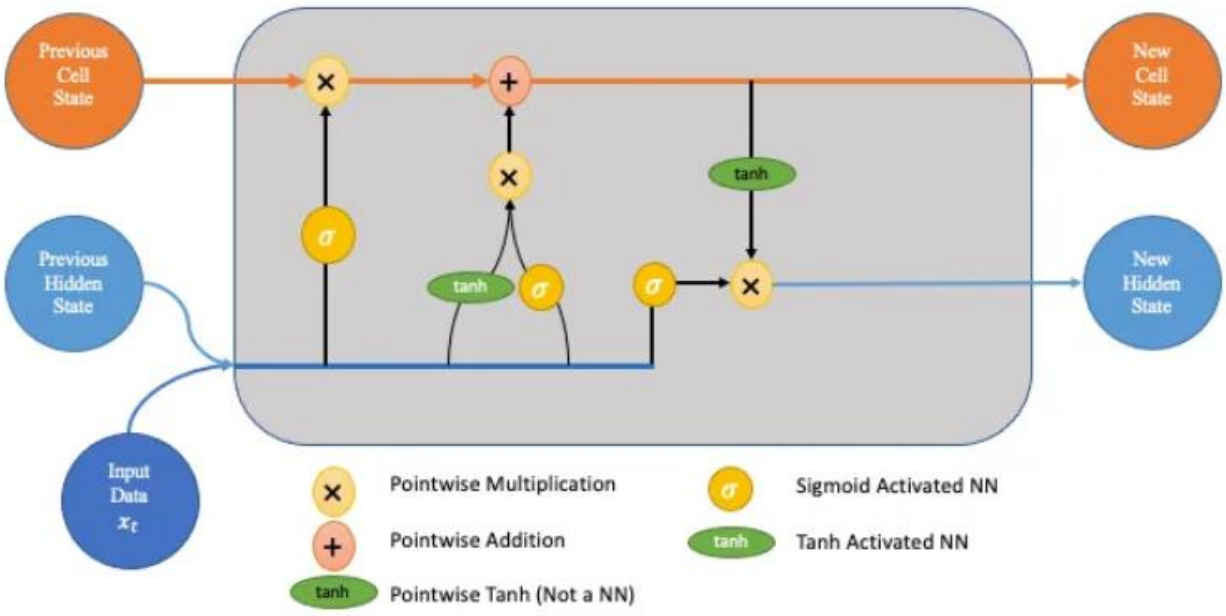
How do LSTM Networks Work?

Firstly, at a basic level, the output of an LSTM at a particular point in time is dependent on three things:

- The current long-term memory of the network — known as the cell state
- The output at the previous point in time — known as the previous hidden state
- The input data at the current time step

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)



LSTM Diagram

Step 1
The first step in the process is the forget gate. Here we will decide which bits of the cell state (long term memory of the network) are useful given both the previous hidden state and new input data.

Step 2
The next step involves the new memory network and the input gate. The goal of this step is to determine what new information should be added to the networks long-term memory (cell state), given the previous hidden state and new input data.

The new memory network is a tanh activated neural network which has learned how to combine the previous hidden state and new input data to generate a 'new memory update vector'. The input gate is a sigmoid activated network which acts as a filter, identifying which components of the 'new memory vector' are worth retaining.

The output of parts 1 and 2 are pointwise multiplied. The resulting combined vector is then added to the cell state, resulting in the long-term memory of the network being updated.

Step 3
Now that our updates to the long-term memory of the network are complete, we can move to the final step, the output gate, deciding the new hidden state. To decide this, we will use three things; the newly updated cell state, the previous hidden state and the new input data.

The step-by-step process for this final step is as follows:
- Apply the tanh function to the current cell state pointwise to obtain the squished cell state, which now lies in [-1,1].
- Pass the previous hidden state and current input data through the sigmoid activated neural network to obtain the filter vector.
- Apply this filter vector to the squished cell state by pointwise multiplication.
- Output the new hidden state!

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Firstly, the steps above are repeated many times. For example, if you are trying to predict the following days stock price based on the previous 30 days pricing data, then the steps will be repeated 30 times. In other words, your model will have iteratively produced 30 hidden states to predict tomorrow's price.

LSTM Applications

LSTM networks find useful applications in the following areas:

- ✓ Language modeling
- ✓ Machine translation
- ✓ Handwriting recognition
- ✓ Image captioning
- ✓ Image generation using attention models
- ✓ Question answering
- ✓ Video-to-text conversion
- ✓ Polymorphic music modeling
- ✓ Speech synthesis
- ✓ Protein secondary structure prediction

Major Difference between LSTM and Bidirectional LSTM

The major difference between the two is that LSTM can process the input sequence in a forward or backward direction at a time, whereas Bidirectional LSTM can process the input sequence in a forward or backward direction simultaneously.

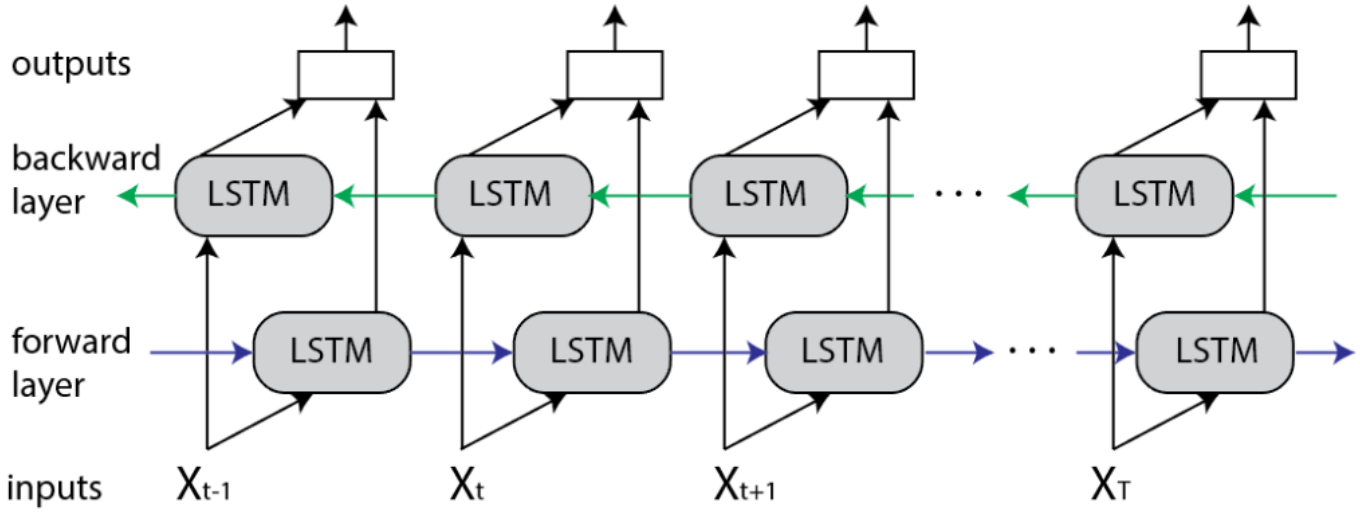
13.Bi-directional LSTM

What is Bidirectional LSTM?

Bidirectional long-short term memory(bi-lstm) is the process of making any neural network o have the sequence information in both directions backwards (future to past) or forward (past to future). In bidirectional, our input flows in two directions, making a bi-lstm different from the regular LSTM. With the regular LSTM, we can make input flow in one direction, either backwards or forward. However, in bi-directional, we can make the input flow in both directions to preserve the future and the past information. These are like an upgrade over LSTMs. In bidirectional LSTMs, each training sequence is presented forward and backward so as to separate recurrent nets. Both sequences are connected to the same output layer. Bidirectional LSTMs have complete information about every point in a given sequence, everything before and after it.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)



But, how do you rely on the information that hasn't happened yet? The human brain uses its senses to pick up information from words, sounds, or from whole sentences that might, at first, make no sense but mean something in a future context. Conventional recurrent neural networks are only capable of using the previous context to get information. Whereas, in bidirectional LSTMs, the information is obtained by processing the data in both directions within two hidden layers, pushed toward the same output layer. This helps bidirectional LSTMs access long-range context in both directions.

Advantages and Disadvantages

This type of architecture has many advantages in real-world problems, especially in NLP. The main reason is that every component of an input sequence has information from both the past and present. For this reason, BiLSTM can produce a more meaningful output, combining LSTM layers from both directions. The BiLSTM model is beneficial in some NLP tasks, such as sentence classification, translation, and entity recognition. In addition, it finds its applications in speech recognition, protein structure prediction, handwritten recognition, and similar fields.

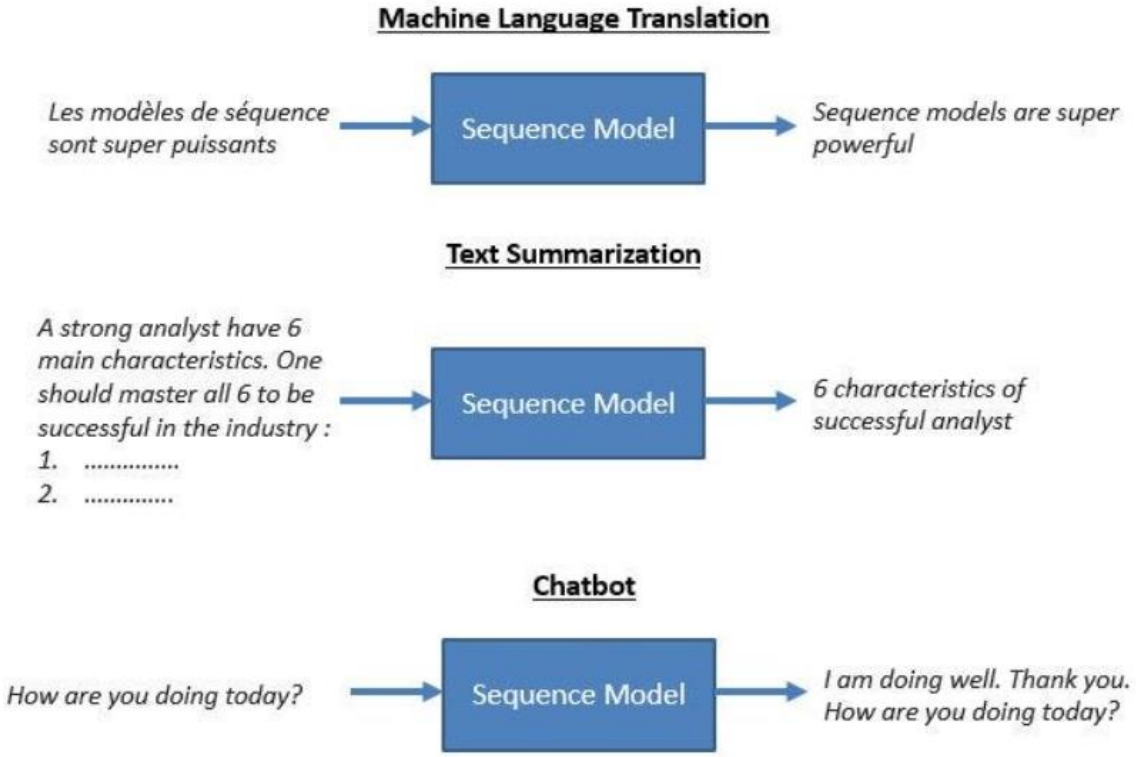
Finally, regarding the disadvantages of BiLSTM compared to LSTM, it's worth mentioning that BiLSTM is a much slower model and requires more time for training. Thus, it is recommended as using it only if there's a real necessity.

14.Sequence-to-Sequence Models (Seq2Seq)

Seq2Seq

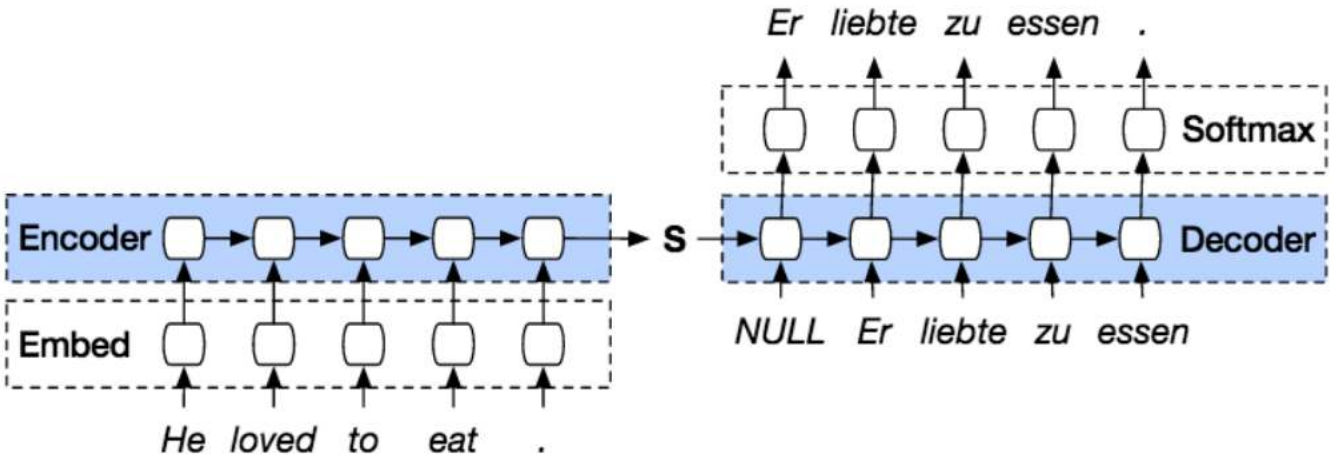
Sequence to Sequence (often abbreviated to seq2seq) models is a special class of Recurrent Neural Network architectures that we typically use (but not restricted) to solve complex Language problems like Machine Translation, Question Answering, creating Chatbots, Text Summarization, etc. Seq2Seq is a method of encoder-decoder based machine translation and language processing that maps an input of sequence to an output of sequence with a tag and attention value. The idea is to use 2 RNNs that will work together with a special token and try to predict the next state sequence from the previous sequence.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)



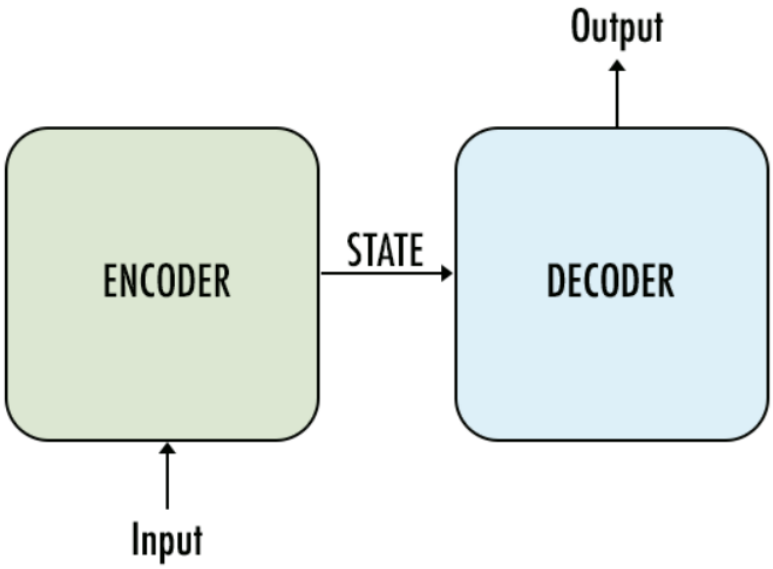
How to predict sequence from the previous sequence?

The Encoder will encode the sentence word by words into an indexed of vocabulary or known words with index, and the decoder will predict the output of the coded input by decoding the input in sequence and will try to use the last input as the next input if it's possible. With this method, it is also possible to predict the next input to create a sentence. Each sentence will be assigned a token to mark the end of the sequence. At the end of prediction, there will also be a token to mark the end of the output. So, from the encoder, it will pass a state to the decoder to predict the output.

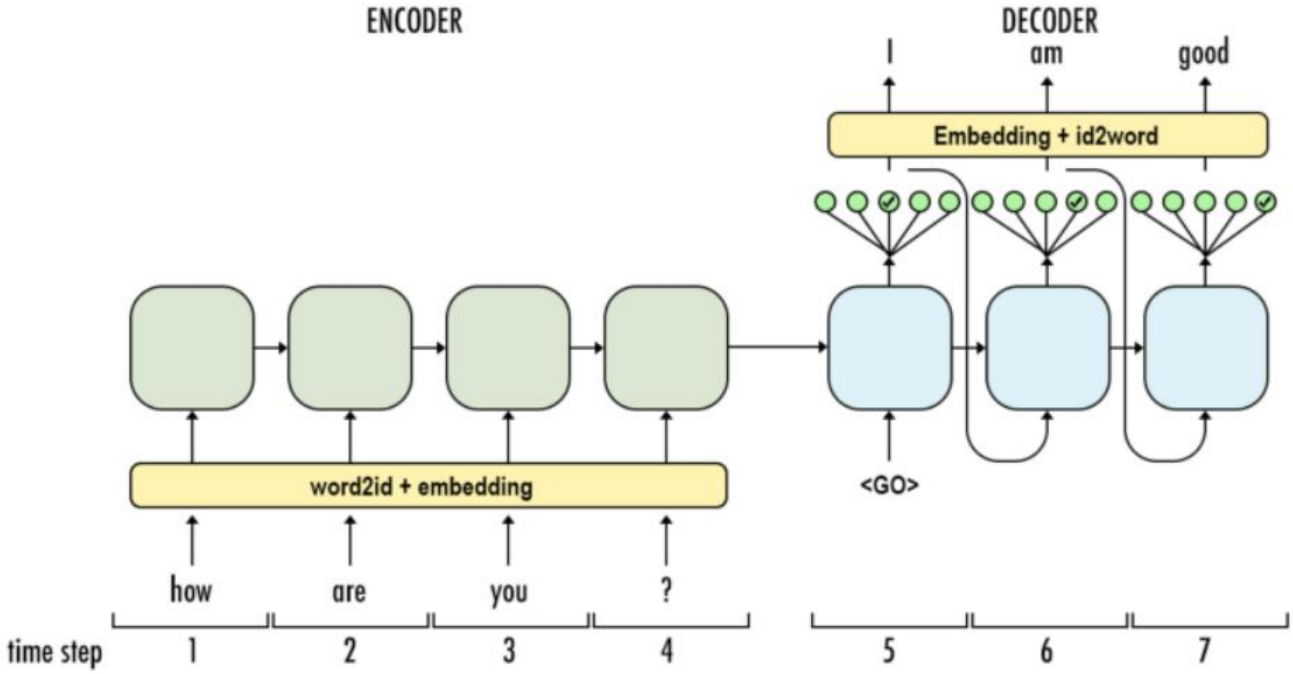


The Encoder will encode our input sentence word by word in sequence and in the end, there will be a token to mark the end of a sentence. The encoder consists of an Embedding layer and a GRU layers. The Embedding layer is a lookup table that stores the embedding of our input into a fixed sized dictionary of words. It will be passed to a GRU layer. GRU layer is a Gated Recurrent Unit that consists of multiple layer type of RNN that will calculate the sequenced input. This layer will calculate the hidden state from the previous one and update the reset, update, and new gates.

This document is available free of charge on studocu
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)



The Decoder will decode the input from the encoder output. It will try to predict the next output and try to use it as the next input if it's possible. The Decoder consists of an Embedding layer, GRU layer, and a Linear layer. The embedding layer will make a lookup table for the output and pass it into a GRU layer to calculate the predicted output state. After that, a Linear layer will help to calculate the activation function to determine the true value of the predicted output.



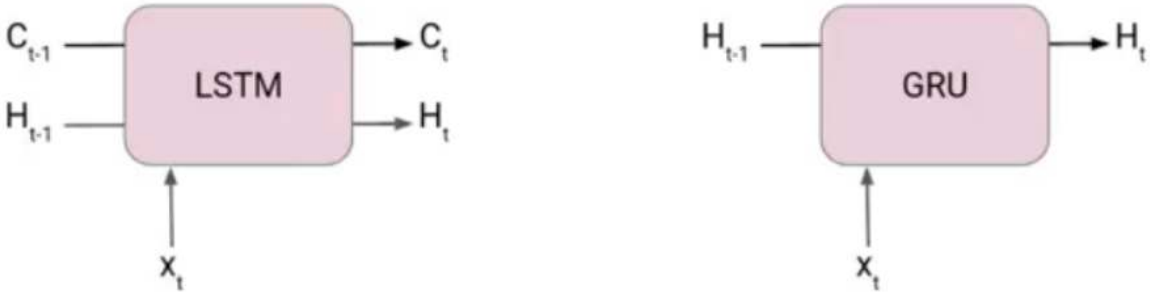
15.Gated Recurrent Unit (GRU)

GRU

Gated Recurrent Unit (GRU) is an advancement of the standard RNN i.e., recurrent neural network. It was introduced by Kyunghyun Cho et al in the year 2014. In sequence modeling techniques, the Gated Recurrent Unit is the newest entrant after RNN and LSTM, hence it offers an improvement over the other two. GRUs are very similar to Long Short-Term Memory (LSTM). Just like LSTM, GRU uses

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

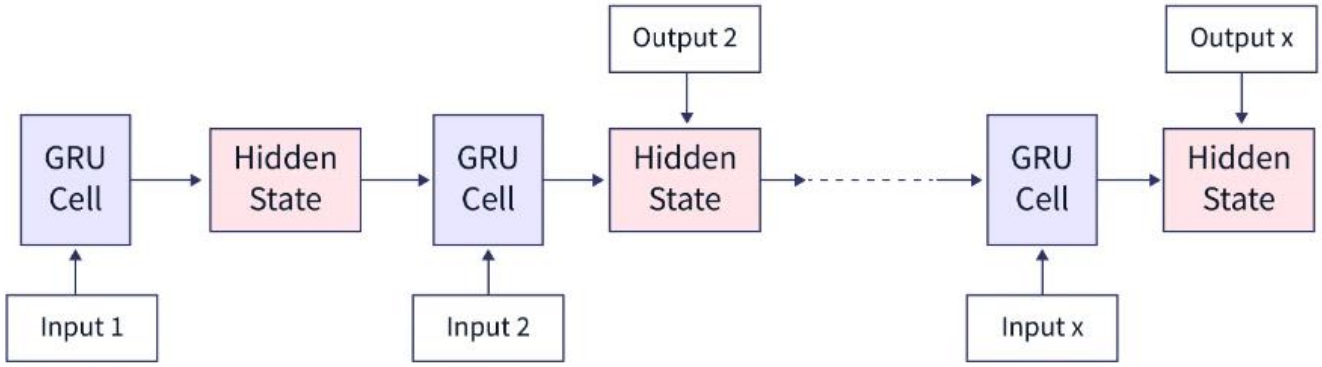
gates to control the flow of information. They are relatively new as compared to LSTM. This is the reason they offer some improvement over LSTM and have simpler architecture.



Another Interesting thing about GRU is that, unlike LSTM, it does not have a separate cell state (Ct). It only has a hidden state (Ht). Due to the simpler architecture, GRUs are faster to train. A Gated Recurrent Unit (GRU) is a Recurrent Neural Network (RNN) architecture type. Like other RNNs, a GRU can process sequential data such as time series, natural language, and speech. The main difference between a GRU and other RNN architectures, such as the Long Short-Term Memory (LSTM) network, is how the network handles information flow through time.

Understanding the GRU Cell

The Gated Recurrent Unit (GRU) cell is the basic building block of a GRU network. It comprises three main components: an update gate, a reset gate, and a candidate hidden state.



One of the key advantages of the GRU cell is its simplicity. Since it has fewer parameters than a long short-term memory (LSTM) cell, it is faster to train and run and less prone to overfitting.

Additionally, one thing to remember is that the GRU cell architecture is simple, the cell itself is a black box, and the final decision on how much we should consider the past state and how much should be forgotten is taken by this GRU cell. We need to look inside and understand what the cell is thinking.

The Architecture of GRU

A GRU cell keeps track of the important information maintained throughout the network. A GRU network achieves this with the following two gates:

- *Reset Gate* - A reset gate identifies the unnecessary information and decides what information to be laid off from the GRU network. Simply put, it decides what information to delete at the specific timestamp.
- *Update Gate* - An update gate determines what current GRU cell will pass information to the next GRU cell. It helps in keeping track of the most important information.

Given below is the simplest architectural form of a GRU cell.

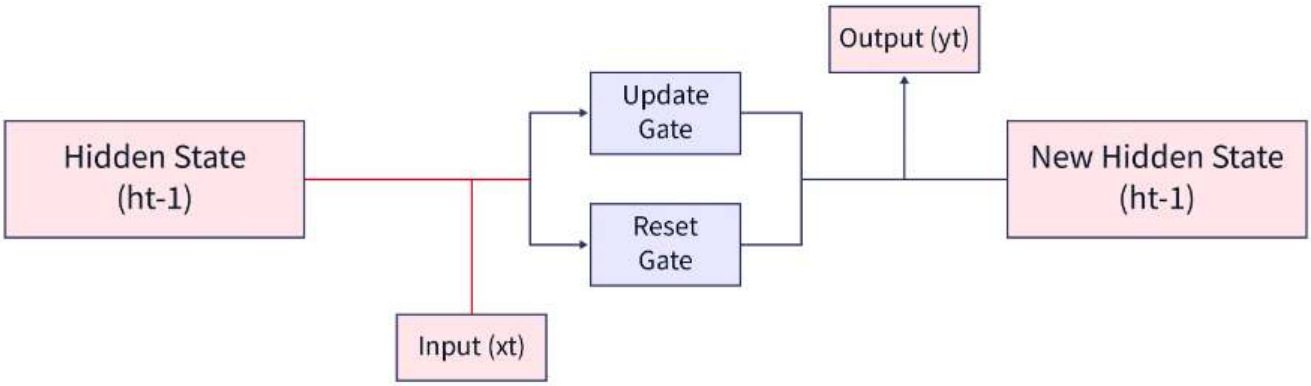
This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

As shown below, a GRU cell takes two inputs:

1. The previous hidden state
2. The input in the current timestamp.

The cell combines these and passes them through the update and reset gates. To get the output in the current timestep, we must pass this hidden state through a dense layer with softmax activation to predict the output. Doing so, a new hidden state is obtained and then passed on to the next time step.



How Does GRU Work?

Gated Recurrent Unit (GRU) networks process sequential data, such as time series or natural language, bypassing the hidden state from one time step to the next. The hidden state is a vector that captures the information from the past time steps relevant to the current time step. The main idea behind a GRU is to allow the network to decide what information from the last time step is relevant to the current time step and what information can be discarded.

Points to Remember

- A GRU network is a modification of the RNN network. GRU cell consists of two gates: the update gate and the reset gate.
- An update gate determines what model will pass information to the next GRU cell. A reset gate identifies the unnecessary information and decides what information to be laid off from the GRU network.
- A candidate's hidden state is calculated from the reset gate. This is used to determine the information stored from the past. This is generally called the memory component in a GRU cell.
- The new hidden state is calculated using the output of the reset gate, update gate, and the candidate hidden state. The weights associated with the various gates, input and hidden state are learnt by a concept called Backpropagation.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

GRU vs LSTM

	GRU	LSTM
Structure	Simpler structure with two gates (update and reset gate)	More complex structure with three gates (input, forget, and output gate)
Parameters	Fewer parameters (3 weight matrices)	More parameters (4 weight matrices)
Training	Faster to train	Slow to train
Space Complexity	In most cases, GRU tend to use fewer memory resources due to its simpler structure and fewer parameters, thus better suited for large datasets or sequences.	LSTM has a more complex structure and a larger number of parameters, thus might require more memory resources and could be less effective for large datasets or sequences.
Performance	Generally performed similarly to LSTM on many tasks, but in some cases, GRU has been shown to outperform LSTM and vice versa. It's better to try both and see which works better for your dataset and task.	LSTM generally performs well on many tasks but is more computationally expensive and requires more memory resources. LSTM has advantages over GRU in natural language understanding and machine translation tasks.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

UNIT - V DEEP REINFORCEMENT & UNSUPERVISED LEARNING

(Page No. 159-187)

1. About Deep Reinforcement Learning Pg. 161

- Reinforcement Learning
- The RL Process
  - The Reward Hypothesis: The Central Idea of Reinforcement Learning
  - Observations/States Space
  - Action Space
    - Discrete Space
    - Continuous Space
  - Type of Tasks
    - Episodic Task
    - Continuing Task
  - Exploration/ Exploitation Trade-Off
- The Two Main Approaches for Solving RL Problems
  - The Policy π: The Agent's Brain
  - Policy-Based Methods
    - Deterministic Policy
    - Stochastic Policy
  - Value-Based Methods
- The "Deep" in Reinforcement Learning
- Applications of Deep Reinforcement Learning
- Challenges of Deep Reinforcement Learning

Difference between Deep Learning and Reinforcement Learning Pg. 166

2. Q-Learning Pg. 167

- What is Q-Learning?
- Important Terms in Q-Learning
- What is the Bellman Equation?
- How to make a Q-Table?
- Major Difference between Q-Learning and Deep Q-Learning

3. Deep Q-Network (DQN) Pg. 171

- DQN
- Pre-Processing the Input and Temporal Limitation

4. Policy Gradient Methods Pg. 173

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

- An Overview of Policy Gradients
- The Advantages of Policy-Gradient Methods
- The Disadvantages of Policy-Gradient Methods

5. Actor-Critic Algorithm Pg. 174

- Pseudocode of Actor-Critic Algorithm

6. About Autoencoding Pg. 175

- Autoencoding
- Architecture of Autoencoders
- Types of Autoencoders
- Application of Autoencoders

7. Convolutional Auto Encoding Pg. 178

8. Variational Auto Encoding. Pg. 179

9. Generative Adversarial Networks Pg. 180

- What is a Generative Model?
- What are GANs?
- Why GANs was Developed?
- Components of GAN
- Steps for Training GAN
- Different Types of GAN Models
- Advantages of GANs
- Disadvantages of GANs

10. Autoencoders for Feature Extraction Pg. 182

- Autoencoder for Regression

11. Auto Encoders for Classification Pg. 184

12. Denoising Autoencoders Pg. 185

13. Sparse Autoencoders Pg. 186

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

1.About Deep Reinforcement Learning

Reinforcement Learning

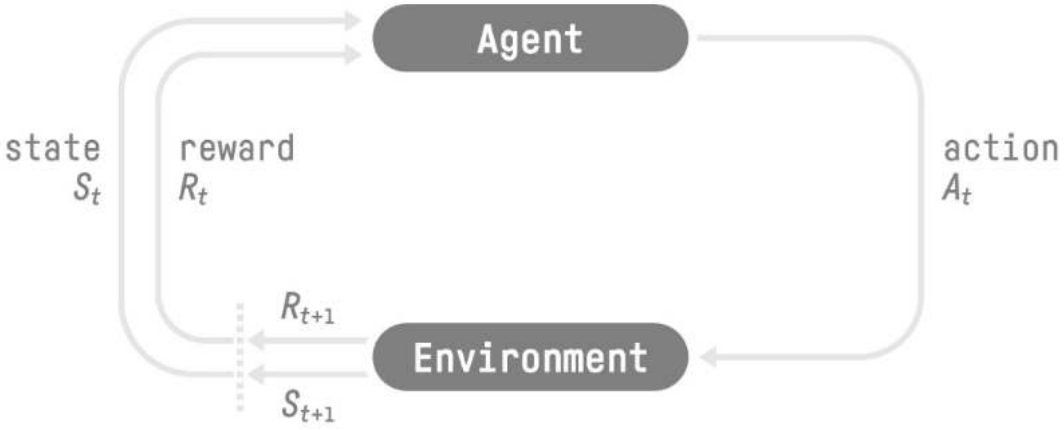
The idea behind Reinforcement Learning is that an agent (an AI) will learn from the environment by interacting with it (through trial and error) and receiving rewards (negative or positive) as feedback for performing actions. Learning from interaction with the environment comes from our natural experiences.

For instance, imagine putting your little brother in front of a video game he never played, a controller in his hands, and letting him alone. Your brother will interact with the environment (the video game) by pressing the right button (action). He got a coin, that's a +1 reward. It's positive, he just understood that in this game he must get the coins. But then, he presses right again and he touches an enemy, he just died -1 reward. By interacting with his environment through trial and error, your little brother understood that he needed to get coins in this environment but avoid the enemies. Without any supervision, the child will get better and better at playing the game. That's how humans and animals learn, through interaction. Reinforcement Learning is just a computational approach of learning from action.

"Reinforcement learning is a framework for solving control tasks (also called decision problems) by building agents that learn from the environment by interacting with it through trial and error and receiving rewards (positive or negative) as unique feedback."

The RL Process

To understand the RL process, let's imagine an agent learning to play a platform game:



- Our Agent receives **state S_0** from the **Environment** — we receive the first frame of our game (Environment).
- Based on that **state S_0**, the Agent takes **action A_0** — our Agent will move to the right.
- Environment goes to a **new state S_1**— new frame.
- The environment gives some **reward R_1** to the Agent — we're not dead (Positive Reward +1).

This RL loop outputs a sequence of state, action, reward and next state. The agent's goal is to maximize its cumulative reward, called the expected return.

The RL process is called the **Markov Decision Process (MDP)**. Markov Property implies that our agent needs only the current state to decide what action to take and not the history of all the states and actions they took before.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

The Reward Hypothesis: The Central Idea of Reinforcement Learning

The **reward** is fundamental in RL because it's the only feedback for the agent. Thanks to it, our agent knows if the action taken was good or not.

Why is the goal of the agent to maximize the expected return?

Because RL is based on the reward hypothesis, which is that all goals can be described as the maximization of the expected return (expected cumulative reward). That's why in Reinforcement Learning, to have the best behaviour, we need to maximize the expected cumulative reward.

Observations/States Space

Observations/States are the information our agent gets from the environment. In the case of a video game, it can be a frame (a screenshot). In the case of the trading agent, it can be the value of a certain stock, etc.

There is a differentiation to make between observation and state:

- State s: is a complete description of the state of the world (there is no hidden information). In a fully observed environment. With a chess game, we are in a fully observed environment, since we have access to the whole check board information.
- Observation o: is a partial description of the state. In a partially observed environment. In Super Mario Bros, we are in a partially observed environment. We receive an observation since we only see a part of the level.

Action Space

The Action space is the set of all possible actions in an environment. The actions can come from a discrete or continuous space:

- *Discrete Space*: the number of possible actions is finite. In Super Mario Bros, we have a finite set of actions since we have only 4 directions and jump.
- *Continuous Space*: the number of possible actions is infinite. A Self Driving Car agent has an infinite number of possible actions since it can turn left 20°, 21,1°, 21,2°, honk, turn right 20°…

Type of Tasks

A task is an instance of a Reinforcement Learning problem. We can have two types of tasks: episodic and continuing.

- *Episodic Task* - In this case, we have a starting point and an ending point (a terminal state). This creates an episode: a list of States, Actions, Rewards, and new States. For instance, think about Super Mario Bros: an episode begins at the launch of a new Mario Level and ending when you're killed or you reached the end of the level.
- *Continuing Task* - These are tasks that continue forever (no terminal state). In this case, the agent must learn how to choose the best actions and simultaneously interact with the environment. For instance, an agent that does automated stock trading. For this task, there is no starting point and terminal state. The agent keeps running until we decide to stop them.

Exploration/ Exploitation Trade-Off

- Exploration is exploring the environment by trying random actions in order to find more information about the environment. You go every day to the same one that you know is good and take the risk to miss another better restaurant.
- Exploitation is exploiting known information to maximize the reward. Try restaurants you never went to before, with the risk of having a bad experience but the probable opportunity of a fantastic experience.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Remember, the goal of our RL agent is to *maximize the expected cumulative reward*. However, we can fall into a common trap.

Let's take an example:
In this game, our mouse can have an infinite amount of small cheese (+1 each). But at the top of the maze, there is a gigantic sum of cheese (+1000). However, if we only focus on exploitation, our agent will never reach the gigantic sum of cheese. Instead, it will only exploit the nearest source of rewards, even if this source is small (exploitation). But if our agent does a little bit of exploration, it can discover the big reward (the pile of big cheese).

This is what we call the exploration/exploitation trade-off. We need to balance how much we explore the environment and how much we exploit what we know about the environment. Therefore, we must define a rule that helps to handle this trade-off.

The Two Main Approaches for Solving RL Problems
In other terms, how to build an RL agent that can select the actions that maximize its expected cumulative reward?

The Policy π: The Agent's Brain

The Policy π is the brain of our Agent, it's the function that tell us what action to take given the state we are. So, it defines the agent's behaviour at a given time. Think of policy as the brain of our agent, the function that will tells us the action to take given a state. This Policy is the function we want to learn, our goal is to find the optimal policy π, the policy that * maximizes expected return when the agent acts according to it. We find this π through training. *

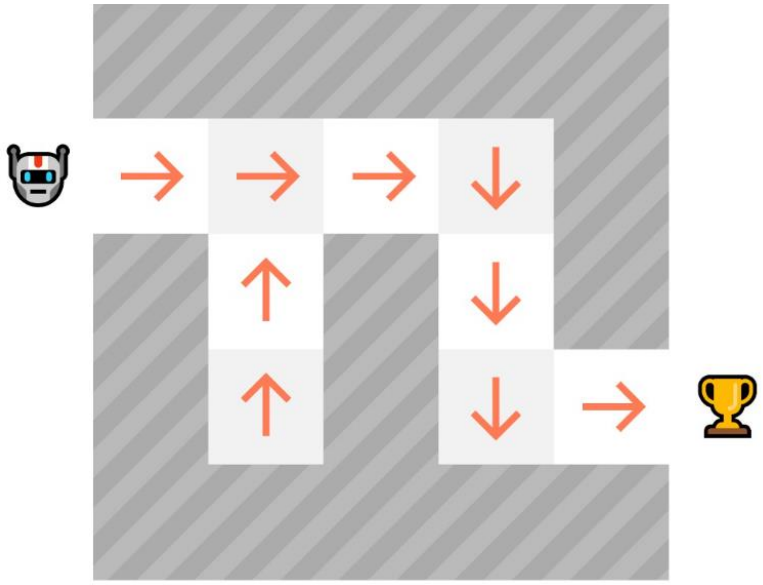
There are two approaches to train our agent to find this optimal policy π*:

- Directly, by teaching the agent to learn which action to take, given the state is in: *Policy-Based Methods*.
- Indirectly, teach the agent to learn which state is more valuable and then take the action that leads to the more valuable states: *Value-Based Methods*.

Policy-Based Methods

In Policy-Based Methods, we learn a policy function directly. This function will map from each state to the best corresponding action at that state. Or a probability distribution over the set of possible actions at that state.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

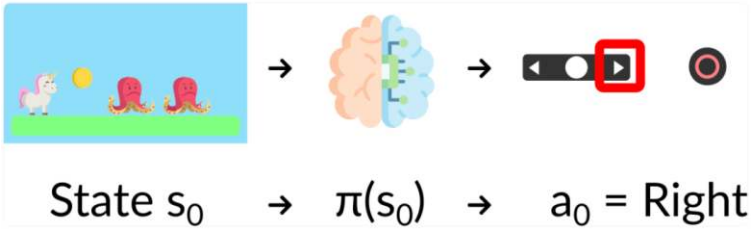


As we can see here, the policy (deterministic) **directly indicates the action to take for each step.**

We have two types of policy:

Deterministic Policy: a policy at a given state will always return the same action.

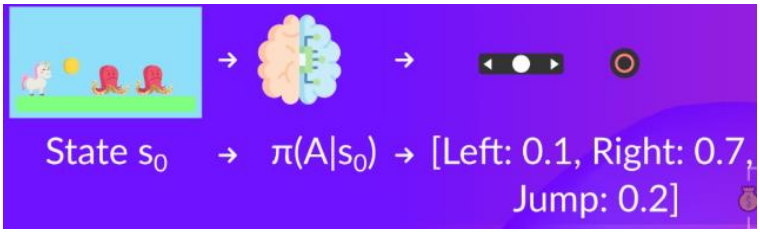
$$a = \pi(s)$$



Stochastic Policy: output a probability distribution over actions. Given an initial state, our stochastic policy will output probability distributions over the possible actions at that state.

$$\pi(a|s) = P[A|s]$$

Probability Distribution over the set of actions given the state



Value-Based Methods

In Value-Based Methods, instead of training a policy function, we train a value function that maps a state to the expected value of being at that state. The value of a state is the expected discounted return the agent can get if it starts in that state, and then act according to our policy. "Act according to our policy" just means that our policy is "going to the state with the highest value".

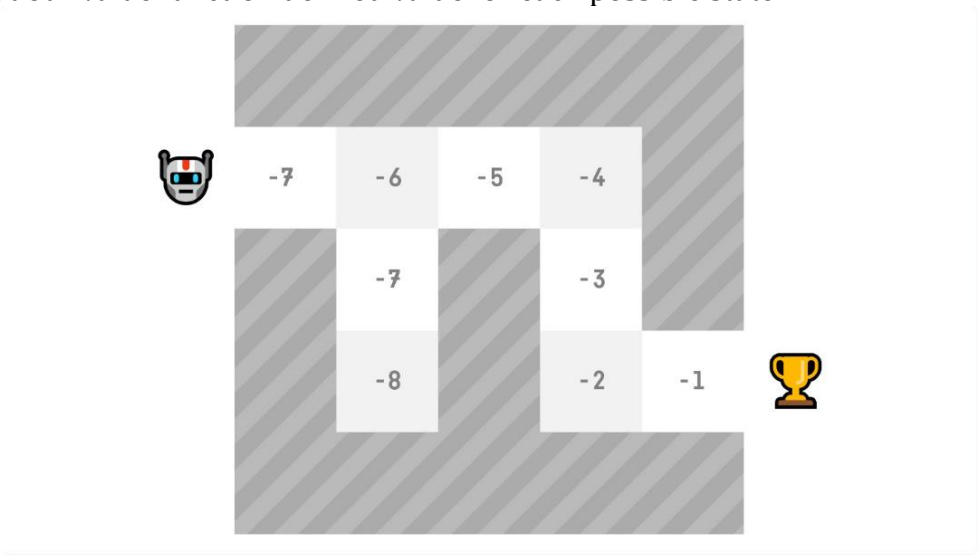
$$v_\pi(s) = \mathbb{E}_\pi\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \mid S_t = s\right]$$

Value function — Expected discounted return — Starting at state s

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Here we see that our value function defined value for each possible state.



Thanks to our value function, at each step our policy will select the state with the biggest value defined by the value function: -7, then -6, then -5 (and so on) to attain the goal.

In brief, Reinforcement Learning is the science of using experiences for making optimal decisions. The process involves the following simple steps:

1. Observing the environment.
2. Making decisions on how to act using some strategy.
3. Acting according to the decisions.
4. Receiving a penalty or reward.
5. Learning from experiences and refining the strategy.
6. Iterating until an optimal strategy is found.

The "Deep" in Reinforcement Learning

Deep Reinforcement Learning introduces deep neural networks to solve Reinforcement Learning problems i.e., to estimate the action to take (policy-based) or to estimate the value of a state (value-based) — hence the name "deep". Deep Reinforcement Learning is the combination of Reinforcement Learning with Deep Learning techniques to solve challenging sequential decision-making problems. The use of deep learning is most useful in problems with high-dimensional state space.

This means that with deep learning, Reinforcement Learning is able to solve more complicated tasks with lower prior knowledge because of its ability to learn different levels of abstractions from data. To use reinforcement learning successfully in situations approaching real-world complexity, however, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experience to new situations. This makes it possible for machines to mimic some human problem-solving capabilities, even in high-dimensional space, which only a few years ago was difficult to conceive.

Through a series of trial and error, a machine keeps learning, making this technology ideal for dynamic environments that keep changing. Although reinforcement learning has been around for decades, it was much more recently combined with deep learning, which yielded phenomenal results. The "deep" portion of reinforcement learning refers to multiple (deep) layers of artificial neural networks that replicate the structure of a human brain. Deep learning requires large amounts of training data and significant computing power. Over the last few years, the volumes of data have exploded while the costs for computing power have dramatically reduced, which has enabled the explosion of deep learning applications.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Applications of Deep Reinforcement Learning

Deep RL techniques have demonstrated their ability to tackle a wide range of problems that were previously unsolved.

- ✓ Gaming
- ✓ Robot control
- ✓ Self-driving cars
- ✓ Healthcare
- ✓ Other: In terms of applications, many areas are likely to be impacted by the possibilities brought by deep Reinforcement Learning, such as finance, business management, marketing, resource management, education, smart grids, transportation, science, engineering, or art. In fact, Deep RL systems are already in production environments. For example, Facebook uses Deep Reinforcement Learning for pushing notifications and for faster video loading with smart prefetching.

Challenges of Deep Reinforcement Learning

Many challenges appear when moving from a simulated setting to solving real-world problems.

- ❖ *Limited freedom of the agent*: In practice, even in the case where the task is well-defined (with explicit reward functions), a fundamental difficulty lies in the fact that it is often not possible to let the agent interact freely and sufficiently in the actual environment, due to safety, cost or time constraints.
- ❖ *Reality gap*: There may be situations, where the agent is not able to interact with the true environment but only with an inaccurate simulation of it. The reality gap describes the difference between the learning simulation and the effective real-world domain.
- ❖ *Limited observations*: For some cases, the acquisition of new observations may not be possible anymore (e.g., the batch setting). Such scenarios occur, for example, in medical trials or tasks with dependence on weather conditions or trading markets such as stock markets.

How those challenges can be addressed:

- *Simulation*: For many cases, a solution is the development of a simulator that is as accurate as possible.
- *Algorithm Design*: The design of the learning algorithms and their level of generalization has a great impact.
- *Transfer Learning*: Transfer learning is a crucial technique to utilize external expertise from other tasks to benefit the learning process of the target task.

Difference between Deep Learning and Reinforcement Learning

The following table highlights the major differences between Deep Learning and Reinforcement Learning –

Basis of Comparison	Deep Learning	Reinforcement learning
Origin	The concept of Deep Learning was introduced in the year 1986 by Rina Dechter.	Reinforcement Learning was developed in the late 1980s by Richard Bellman.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Basis of Comparison	Deep Learning	Reinforcement learning
Utilization	In the areas of speech and picture recognition, the dimension reduction task, and the pretraining for deep networks.	Specifically, in the fields of robotics, computer gaming, telecommunications, AI in healthcare, and elevator scheduling.
Data Existence	Data set that is already present and necessary for learning.	Because it is exploratory in nature, it does not require an existing data collection for the purpose of learning.
Comparison with Human Brain	Reinforcement learning is a type of artificial intelligence that can be enhanced by the use of feedback, making it more comparable to the capabilities of the human brain than deep learning.	Deep learning is focused mostly on recognition and has a weaker connection to interactive learning.
Method of Instruction	Deep learning can perform the desired action by first doing an analysis of previously collected data, and then applying the knowledge gained to a fresh collection of data.	The answer can be altered through the use of reinforcement learning, which adapts to continual input.
Applications	Image and speech recognition, as well as deep network pre-training and dimension reduction are all examples of applications for deep learning.	In contrast, reinforcement learning is applied in situations where optimal control is required when dealing with external stimuli. Some examples of this include robotics, elevator scheduling, telecommunications, computer games, and artificial intelligence in healthcare.
Also called	Deep learning is also referred to as hierarchical learning or deep structured learning.	Reinforcement learning has no other commonly used terms.

2.Q Learning

What is Q-Learning?

Q-Learning is a Reinforcement learning policy that will find the next best action, given a current state. It chooses this action at random and aims to maximize the reward.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

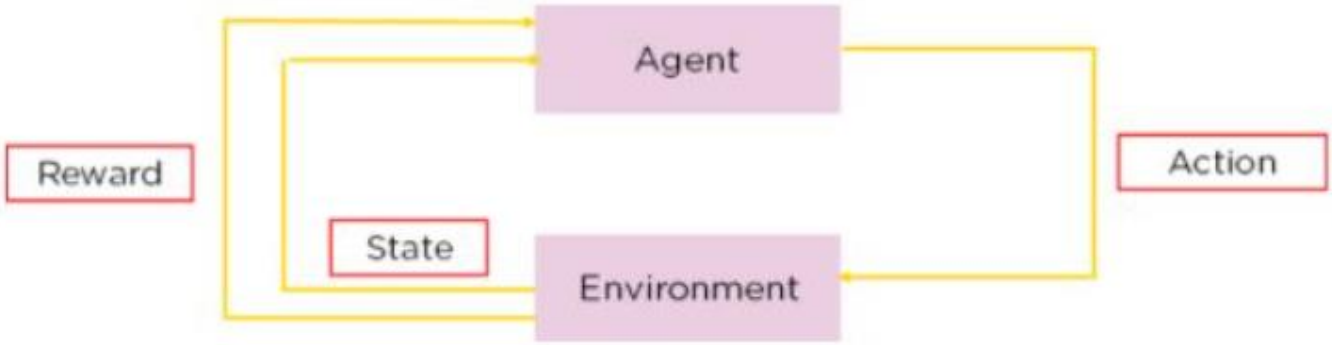


Figure 3: Components of Q-Learning

Q-learning is a model-free, off-policy reinforcement learning that will find the best course of action, given the current state of the agent. Depending on where the agent is in the environment, it will decide the next action to be taken. The objective of the model is to find the best course of action given its current state. To do this, it may come up with rules of its own or it may operate outside the policy given to it to follow. This means that there is no actual need for a policy, hence we call it off-policy.

Model-free means that the agent uses predictions of the environment's expected response to move forward. It does not use the reward system to learn, but rather, trial and error.

An example of Q-learning is an Advertisement recommendation system. In a normal ad recommendation system, the ads you get are based on your previous purchases or websites you may have visited. If you've bought a TV, you will get recommended TVs of different brands. Using Q-learning, we can optimize the ad recommendation system to recommend products that are frequently bought together. The reward will be if the user clicks on the suggested product.

Figure 4: Ad Recommendation System

Figure 5: Ad Recommendation System with Q-Learning

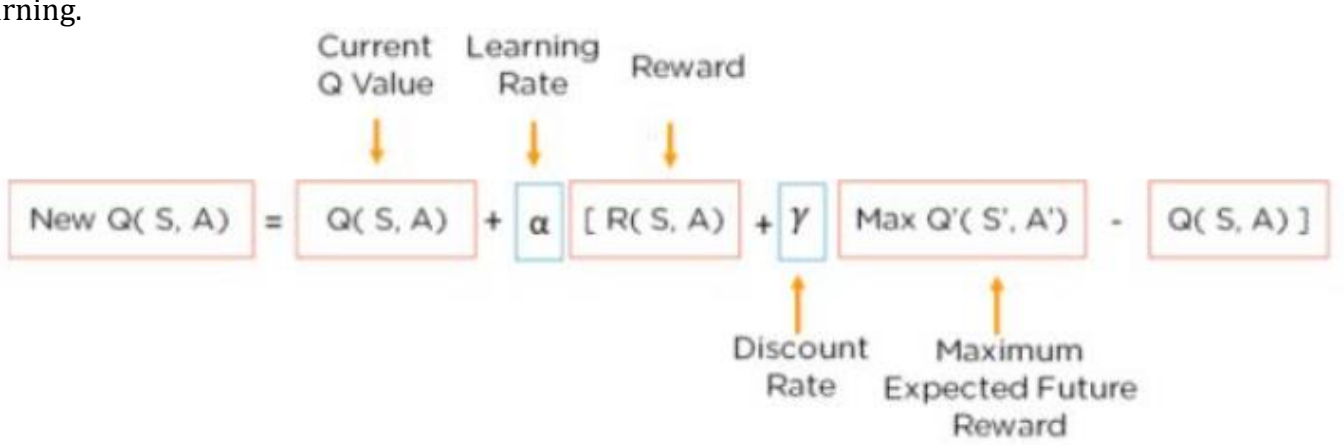
Important Terms in Q-Learning

- States: The State, S, represents the current position of an agent in an environment.
- Action: The Action, A, is the step taken by the agent when it is in a particular state.
- Rewards: For every action, the agent will get a positive or negative reward.
- Episodes: When an agent ends up in a terminating state and can't take a new action.
- Q-Values: Used to determine how good an Action, A, taken at a particular state, S, is. Q (A, S).
- Temporal Difference: A formula used to find the Q-Value by using the value of current state and action and previous state and action.

This document is available free of charge on studocu
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

What is the Bellman Equation?

The Bellman Equation is used to determine the value of a particular state and deduce how good it is to be in/take that state. The optimal state will give us the highest optimal value. The equation is given below. It uses the current state, and the reward associated with that state, along with the maximum expected reward and a discount rate, which determines its importance to the current state, to find the next state of our agent. The learning rate determines how fast or slow, the model will be learning.



How to make a Q-Table?

While running our algorithm, we will come across various solutions and the agent will take multiple paths. How do we find out the best among them? This is done by tabulating our findings in a table called a Q-Table. A **Q-Table** helps us to find the best action for each state in the environment. We use the Bellman Equation at each state to get the expected future state and reward and save it in a table to compare with other states.

Let's create a q-table for an agent that has to learn to run, fetch and sit on command. The steps taken to construct a q-table are:

Step 1: Create an initial Q-Table with all values initialized to 0.
When we initially start, the values of all states and rewards will be 0. Consider the Q-Table shown below which shows a dog simulator learning to perform actions:

Action	Fetching	Sitting	Running
Start	0	0	0
Idle	0	0	0
Wrong Action	0	0	0
Correct Action	0	0	0
End	0	0	0

Figure 7: Initial Q-Table

Step 2: Choose an action and perform it. Update values in the table
This is the starting point. We have performed no other action as of yet. Let us say that we want the agent to sit initially, which it does. The table will change to:

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Action	Fetching	Sitting	Running
Start	0	1	0
Idle	0	0	0
Wrong Action	0	0	0
Correct Action	0	0	0
End	0	0	0

Figure 8: Q-Table after performing an action

Step 3: Get the value of the reward and calculate the value Q-Value using Bellman Equation
For the action performed, we need to calculate the value of the actual reward and the Q (S, A) value.

Action	Fetching	Sitting	Running
Start	0	1	0
Idle	0	0	0
Wrong Action	0	0	0
Correct Action	0	34	0
End	0	0	0

Figure 9: Updating Q-Table with Bellman Equation

Step 4: Continue the same until the table is filled or an episode ends
The agent continues taking actions and for each action, the reward and Q-value are calculated and it updates the table.

Action	Fetching	Sitting	Running
Start	5	7	10
Idle	2	5	3
Wrong Action	2	6	1
Correct Action	54	34	17
End	3	1	4

Figure 10: Final Q-Table at end of an episode

Major Difference between Q-Learning and Deep Q-Learning

The primary reason for developing Deep Q-Learning was to handle environments that involve continuous action and states. The rudimentary Q-Learning algorithm can be used for small and discrete environments. This is because it works by maintaining a Q-table where the row encodes specific states and the columns encode the various actions that the agent can take in the environment. In a continuous environment, Q-learning can still be worked with by discretizing the states. If multiple variables are to be defined in any possible state in the environment, the Q-table becomes ridiculously large and impractical. The reason is apparent; the greater number of rows and columns, the more time agents take to explore every state and update values. This is not a feasible solution but Deep Q-networks as it uses a deep neural network to approximate the Q-table. Deep Q Learning uses the Q-learning idea and takes it one step further. Instead of using a Q-table, we use a Neural Network that takes a state and approximates the Q-values for each action based on that state.

This document is available free of charge on studocu
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

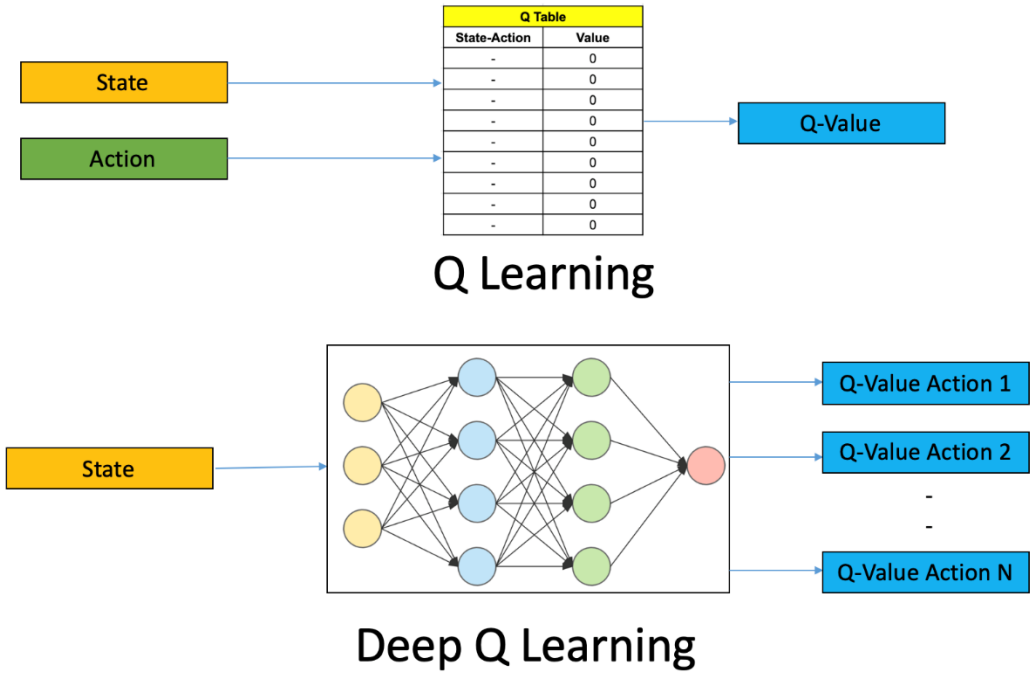
3. Deep Q-Network (DQN)

Deep Q Networks (DQN) are neural networks (and/or related tools) that utilize deep Q learning in order to provide models such as the simulation of intelligent video game play. Rather than being a specific name for a specific neural network build, Deep Q Networks may be composed of convolutional neural networks and other structures that use specific methods to learn about various processes.

DQN

In deep Q-learning, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output. The method of deep Q learning typically uses something called general policy iteration, described as the conjunction of policy evaluation and policy iteration, to learn policies from high dimensional sensory input.

In a general sense, deep Q networks train on inputs that represent active players in areas or other experienced samples and learn to match those data with desired outputs. In other words, with deep Q learning, the AI player gets to be more like a human player in learning to achieve desired outcomes.
The comparison between Q-learning & deep Q-learning is wonderfully illustrated below:



Steps involved in Reinforcement Learning using Deep Q-Learning Networks (DQNs)

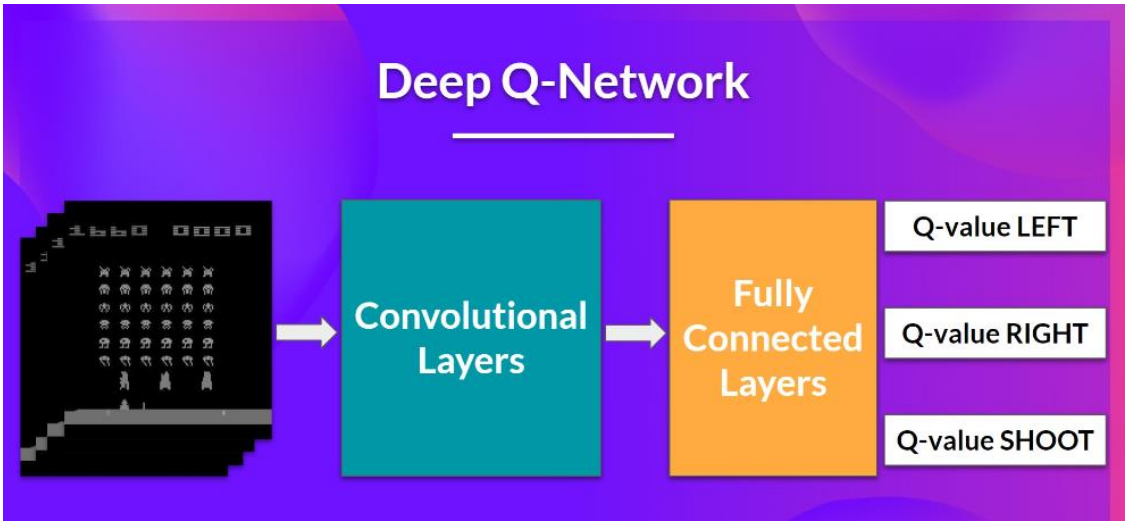
- All the past experience is stored by the user in memory
- The next action is determined by the maximum output of the Q-network
- The loss function here is mean squared error of the predicted Q-value and the target Q-value – Q*. This is basically a regression problem. However, we do not know the target or actual value here as we are dealing with a reinforcement learning problem. Going back to the Q-value update equation derived from the Bellman equation. we have:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

The section in green represents the target. We can argue that it is predicting its own value, but since R is the unbiased true reward, the network is going to update its gradient using backpropagation to finally converge.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

This is the architecture of our Deep Q-Learning network:

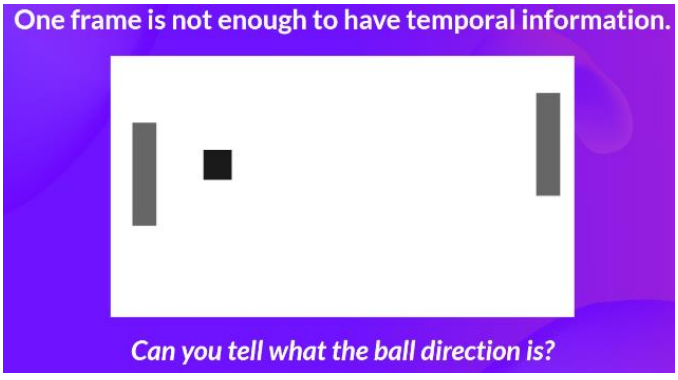


As input, we take a stack of 4 frames passed through the network as a state and output a vector of Q-values for each possible action at that state. Then, like with Q-Learning, we just need to use our epsilon-greedy policy to select which action to take. When the Neural Network is initialized, the Q-value estimation is terrible. But during training, our Deep Q-Network agent will associate a situation with appropriate action and learn to play the game well.

Pre-Processing the Input and Temporal Limitation

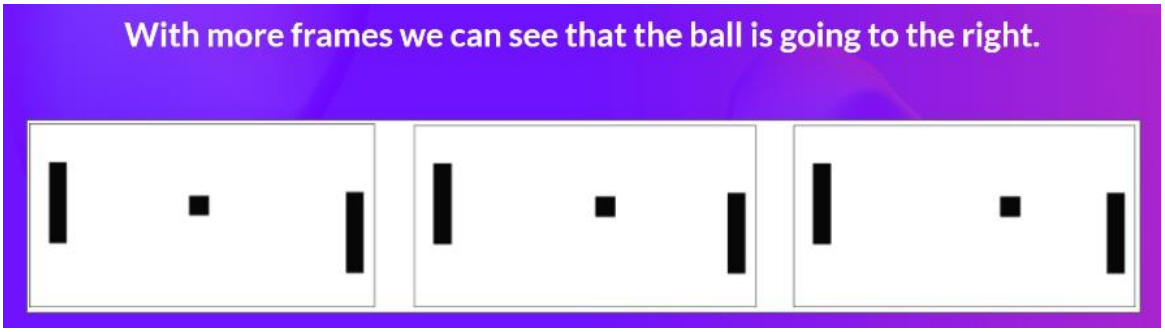
We need to pre-process the input. It's an essential step since we want to reduce the complexity of our state to reduce the computation time needed for training. To achieve this, we reduce the state space to 84x84 and grayscale it. We can do this since the colours in Atari environments don't add important information. This is an essential saving since we reduce our three-color channels (RGB) to 1. We can also crop a part of the screen in some games if it does not contain important information. Then we stack four frames together.

Why do we stack four frames together? We stack frames together because it helps us handle the problem of temporal limitation. Let's take an example with the game of Pong. When you see this frame:



Can you tell me where the ball is going? No, because one frame is not enough to have a sense of motion! But what if I add three more frames? Here you can see that the ball is going to the right.

This document is available free of charge on studocu
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)



That's why, to capture temporal information, we stack four frames together. Then, the stacked frames are processed by three convolutional layers. These layers allow us to capture and exploit spatial relationships in images. But also, because frames are stacked together, you can exploit some temporal properties across those frames. Finally, we have a couple of fully connected layers that output a Q-value for each possible action at that state. So, we see that Deep Q-Learning is using a neural network to approximate, given a state, the different Q-values for each possible action at that state.

4. Policy Gradient Methods

Policy-Gradient is a subclass of Policy-Based Methods, a category of algorithms that aims to optimize the policy directly without using a value function using different techniques. The difference with Policy-Based Methods is that Policy-Gradient methods are a series of algorithms that aim to optimize the policy directly by estimating the weights of the optimal policy using Gradient Ascent.

An Overview of Policy Gradients

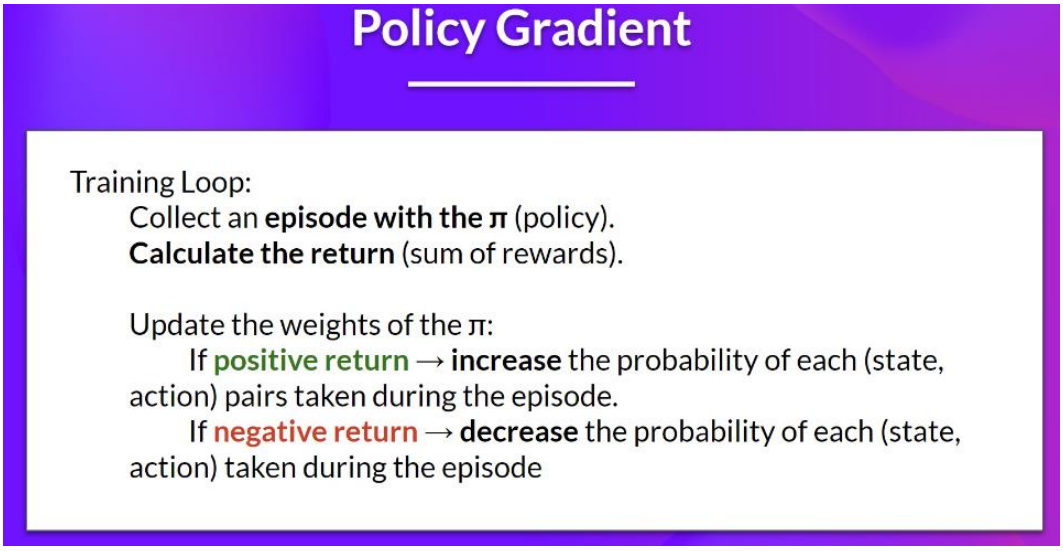
Why do we optimize the policy directly by estimating the weights of an optimal policy using Gradient Ascent in Policy Gradients Methods?

Remember that reinforcement learning aims to find an optimal behaviour strategy (policy) to maximize its expected cumulative reward. We also need to remember that a policy is a function that given a state, outputs, a distribution over actions (in our case using a stochastic policy). Our goal with Policy-Gradients is to control the probability distribution of actions by tuning the policy such that good actions (that maximize the return) are sampled more frequently in the future.

Let's take a simple example: We collect an episode by letting our policy interact with its environment. We then look at the sum of rewards of the episode (expected return). If this sum is positive, we consider that the actions taken during the episodes were good: Therefore, we want to increase the P(a|s) (probability of taking that action at that state) for each state-action pair.

The Policy Gradient algorithm (simplified) looks like this:

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)



The Advantages of Policy-Gradient Methods

There are multiple advantages over Deep Q-Learning methods. Let's see some of them:

- ✓ The simplicity of the integration: we can estimate the policy directly without storing additional data (action values).
- ✓ Policy gradient methods can learn a stochastic policy while value functions can't. This has two consequences:
  a. We don't need to implement an exploration/exploitation trade-off by hand. Since we output a probability distribution over actions, the agent explores the state space without always taking the same trajectory.
  b. We also get rid of the problem of perceptual aliasing. Perceptual aliasing is when two states seem (or are) the same but need different actions.

The Disadvantages of Policy-Gradient Methods

Naturally, Policy Gradient methods have also some disadvantages:

- Policy gradients converge a lot of time on a local maximum instead of a global optimum.
- Policy gradient goes faster, step by step: it can take longer to train (inefficient).
- Policy gradient can have high variance (solution baseline).

5. Actor-Critic Algorithm

Actor-Critic methods are temporal difference (TD) learning methods that represent the policy function independent of the value function. A policy function (or policy) returns a probability distribution over actions that the agent can take based on the given state. A value function determines the expected return for an agent starting at a given state and acting according to a particular policy forever after.

In the Actor-Critic method, the policy is referred to as the actor that proposes a set of possible actions given a state, and the estimated value function is referred to as the critic, which evaluates actions taken by the actor based on the given policy.

- The "Critic" estimates the value function. This could be the action-value (the Q value) or state-value (the V value).
- The "Actor" updates the policy distribution in the direction suggested by the Critic (such as with policy gradients).

This document is available free of charge on studocu
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

and both the Critic and Actor functions are parameterized with neural networks.

In a simple term, Actor-Critic is a Temporal Difference (TD) version of Policy gradient. It has two networks: *Actor and Critic*. The actor decided which action should be taken and critic inform the actor how good was the action and how it should adjust. The learning of the actor is based on policy gradient approach. In comparison, critics evaluate the action produced by the actor by computing the value function.

The basic idea behind actor-critic methods is that there are two deep neural networks. The actor network approximates the agent's policy: a probability distribution that tells us the probability of selecting a (continuous) action given some state of the environment. The critic network approximates the value function: the agent's estimate of future rewards that follow the current state. These two networks interact to shift the policy towards more profitable states, where profitability is determined by interacting with the environment. This requires no prior knowledge of how our environment works, or any input regarding rules of the game. All we have to do is let the algorithm interact with the environment and watch as it learns.

Pseudocode of Actor-Critic Algorithm

1. Sample {s_t, a_t} using the policy πθ from the actor-network.
2. Evaluate the advantage function A_t. It can be called as TD error δt. In Actor-critic algorithm, advantage function is produced by the critic-network.

$$A_{\pi_\theta}(s_t, a_t) = r(s_t, a_t) + V_{\pi_\theta}(s_{t+1}) - V_{\pi_\theta}(s_t)$$

3. Evaluate the gradient using the below expression:

$$\nabla J(\theta) \approx \sum_{t=0}^{T-1} \nabla_\theta log \pi_\theta(a_t, s_t) A_{\pi_\theta}(s_t, a_t)$$

4. Update the policy parameters, θ

$$\theta = \theta + \alpha \nabla J(\theta)$$

5. Update the weights of the critic-based value-based RL(Q-learning). δt is equivalent to advantage function.

$$w = w + \alpha \delta_t$$

6. Repeat 1 to 5 until we find the optimal policy πθ.

Actor-Critic Methods are very useful reinforcement learning techniques most useful for applications in robotics as they allow software to output continuous, rather than discrete actions. This enables control of electric motors to actuate movement in robotic systems, at the expense of increased computational complexity.

6. About Autoencoding

Autoencoding

"Autoencoding" is a data compression algorithm where the compression and decompression functions are 1) data-specific, 2) lossy, and 3) learned automatically from examples rather than engineered by a human. Additionally, in almost all contexts where the term "autoencoder" is used, the compression and decompression functions are implemented with neural networks.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

1) Autoencoders are *data-specific*, which means that they will only be able to compress data similar to what they have been trained on. This is different from, say, the MPEG-2 Audio Layer III (MP3) compression algorithm, which only holds assumptions about "sound" in general, but not about specific types of sounds. An autoencoder trained on pictures of faces would do a rather poor job of compressing pictures of trees, because the features it would learn would be face-specific.

2) Autoencoders are *lossy*, which means that the decompressed outputs will be degraded compared to the original inputs (similar to MP3 or JPEG compression). This differs from lossless arithmetic compression.

3) Autoencoders are *learned automatically* from data examples, which is a useful property: it means that it is easy to train specialized instances of the algorithm that will perform well on a specific type of input. It doesn't require any new engineering, just appropriate training data.

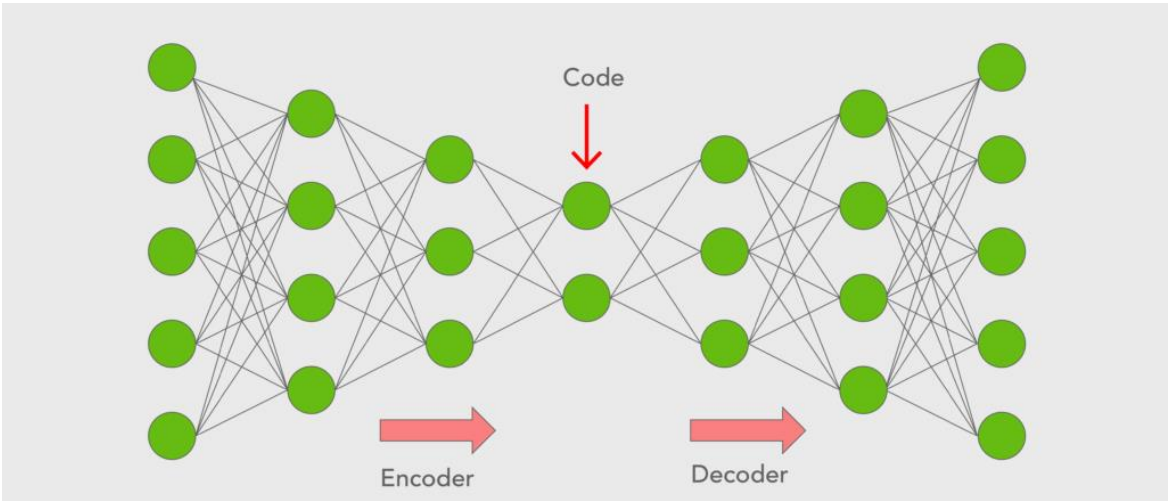
Autoencoders are a specific type of feedforward neural networks where the input is the same as the output. They compress the input into a lower-dimensional code and then reconstruct the output from this representation. The code is a compact “summary” or “compression” of the input, also called the latent-space representation.

Architecture of Autoencoders

Autoencoder is a type of neural network where the output layer has the same dimensionality as the input layer. In simpler words, the number of output units in the output layer is equal to the number of input units in the input layer. An autoencoder replicates the data from the input to the output in an unsupervised manner and is therefore sometimes referred to as a replicator neural network.

An autoencoder consists of *3 components*: encoder, code and decoder. The encoder compresses the input and produces the code, the decoder then reconstructs the input only using this code.

- *Encoder:* An encoder is a feedforward, fully connected neural network that compresses the input into a latent space representation and encodes the input image as a compressed representation in a reduced dimension. The compressed image is the distorted version of the original image.
- *Code:* This part of the network contains the reduced representation of the input that is fed into the decoder.
- *Decoder:* Decoder is also a feedforward network like the encoder and has a similar structure to the encoder. This network is responsible for reconstructing the input back to the original dimensions from the code.



This document is available free of charge on studocu
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

First, the input goes through the encoder where it is compressed and stored in the layer called Code, then the decoder decompresses the original input from the code. The main objective of the autoencoder is to get an output identical to the input.

Note that the decoder architecture is the mirror image of the encoder. This is not a requirement but it's typically the case. The only requirement is the dimensionality of the input and output must be the same. To build an autoencoder, you need three things: an encoding function, a decoding function, and a distance function between the amount of information loss between the compressed representation of your data and the decompressed representation (i.e., a "loss" function). The encoder and decoder will be chosen to be parametric functions (typically neural networks), and to be differentiable with respect to the distance function, so the parameters of the encoding/decoding functions can be optimized to minimize the reconstruction loss, using Stochastic Gradient Descent.

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

- *Code size:* number of nodes in the middle layer. Smaller size results in more compression.
- *Number of layers:* the autoencoder can be as deep as we like. In the figure above we have 2 layers in both the encoder and decoder, without considering the input and output.
- *Number of nodes per layer:* the autoencoder architecture we're working on is called a stacked autoencoder since the layers are stacked one after another. Usually stacked autoencoders look like a "sandwich". The number of nodes per layer decreases with each subsequent layer of the encoder, and increases back in the decoder. Also, the decoder is symmetric to the encoder in terms of layer structure.
- *Loss function:* we either use Mean Squared Error (MSE) or binary crossentropy. If the input values are in the range [0, 1] then we typically use crossentropy, otherwise we use the Mean Squared Error.

Types of Autoencoders

There are many types of autoencoders and some of them are listed below

- Convolutional Autoencoder
- Variational Autoencoders - encodes information onto a distribution, enabling us to use it for new data generation.
- Denoising autoencoders - designed to remove noise from data or images.
- Deep autoencoders
- Sparse Autoencoders - uses sparsity to create an information bottleneck.
- Under Complete Autoencoders - the most basic and widely used type, frequently referred to as an Autoencoder.
- Contractive Autoencoders

When should we not use autoencoders?

An autoencoder could misclassify input errors that are different from those in the training set or changes in underlying relationships that a human would notice. Another drawback is you may eliminate the vital information in the input data.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

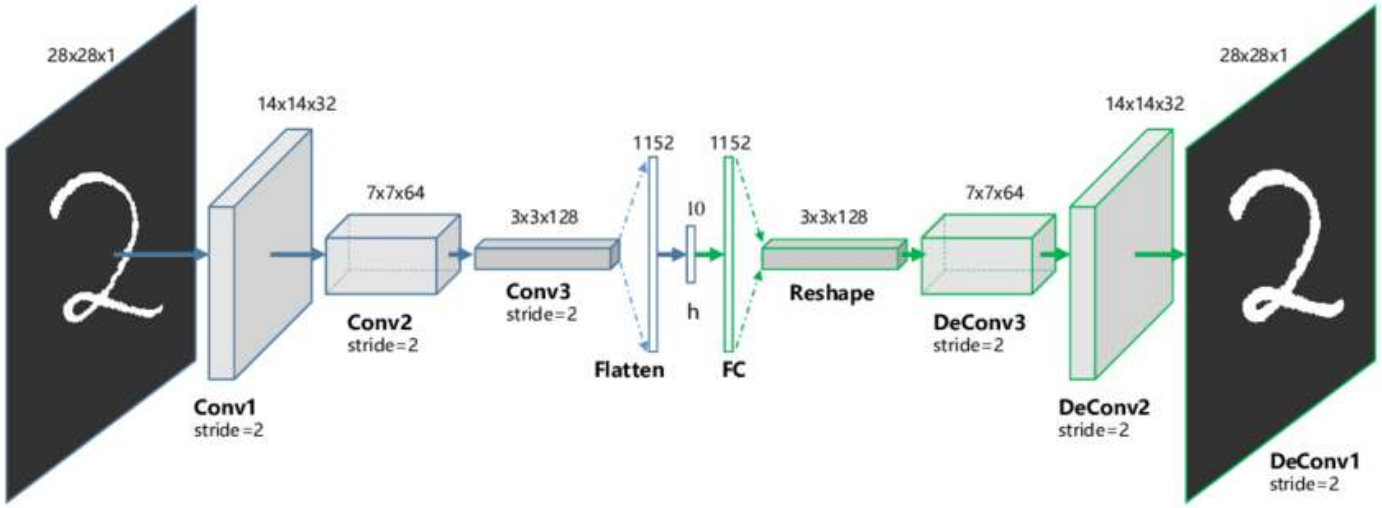
Application of Autoencoders

- ✓ *File Compression:* Primary use of Autoencoders is that they can reduce the dimensionality of input data which we in common refer to as file compression. Autoencoders works with all kinds of data like Images, Videos, and Audio, this helps in sharing and viewing data faster than we could do with its original file size.
- ✓ *Image Denoising* - Autoencoders are very good at denoising images. When an image gets corrupted or there is a bit of noise in it, we call this image a noisy image. To obtain proper information about the content of the image, we perform image denoising.
- ✓ *Dimensionality Reduction* - The autoencoders convert the input into a reduced representation which is stored in the middle layer called code. This is where the information from the input has been compressed and by extracting this layer from the model, each node can now be treated as a variable. Thus, we can conclude that by trashing out the decoder part, an autoencoder can be used for dimensionality reduction with the output being the code layer.
- ✓ *Feature Extraction* - Encoding part of Autoencoders helps to learn important hidden features present in the input data, in the process to reduce the reconstruction error. During encoding, a new set of combinations of original features is generated.
- ✓ *Image Generation* - Variational Autoencoder (VAE) is a Generative Model, used to generate images that have not been seen by the model yet. The idea is that given input images like images of face or scenery, the system will generate similar images. The use is to:
  - generate new characters of animation
  - generate fake human images
- ✓ *Image Colourisation* - One of the applications of autoencoders is to convert a black and white picture into a coloured image. Or we can convert a coloured image into a grayscale image.

7. Convolutional Auto Encoding

Convolutional Autoencoder is a variant of Convolutional Neural Networks that are used as the tools for unsupervised learning of convolution filters. They are generally applied in the task of image reconstruction to minimize reconstruction errors by learning the optimal filters. Once they are trained in this task, they can be applied to any input in order to extract features. Convolutional Autoencoders are general-purpose feature extractors differently from general autoencoders that completely ignore the 2D image structure. In autoencoders, the image must be unrolled into a single vector and the network must be built following the constraint on the number of inputs.

The block diagram of a Convolutional Autoencoder is given in the below figure.



This document is available free of charge on studocu

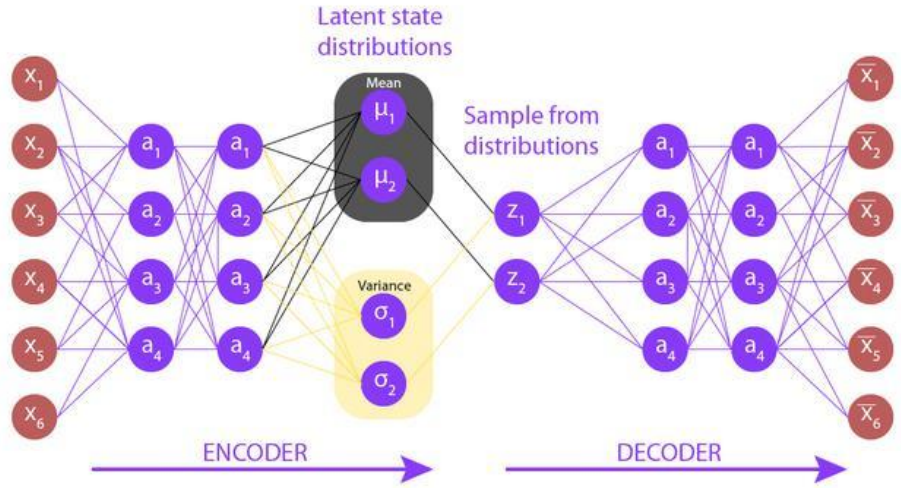
Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Convolutional Autoencoders (CAE) learn to encode the input in a set of simple signals and then reconstruct the input from them. In addition, we can modify the geometry or generate the reflectance of the image by using CAE. In this type of autoencoder, encoder layers are known as convolution layers and decoder layers are also called deconvolution layers. The deconvolution side is also known as up sampling or transpose convolution.

8. Variational Auto Encoding

Variational Autoencoder was proposed in 2013 by Knigma and Welling at Google and Qualcomm. It has many applications such as data compression, synthetic data creation etc. A Variational Autoencoder can be defined as being an autoencoder whose training is regularised to avoid overfitting and ensure that the latent space has good properties that enable generative process.

Just as a standard autoencoder, a variational autoencoder is an architecture composed of both an encoder and a decoder and that is trained to minimise the reconstruction error between the encoded-decoded data and the initial data. However, in order to introduce some regularisation of the latent space, we proceed to a slight modification of the encoding-decoding process: instead of encoding an input as a single point, we encode it as a distribution over the latent space.



Variational autoencoder is different from autoencoder in a way such that it provides a statistic manner for describing the samples of the dataset in latent space. Therefore, in variational autoencoder, the encoder outputs a probability distribution in the bottleneck layer instead of a single output value.

The model is then trained as follows:

- first, the input is encoded as distribution over the latent space
- second, a point from the latent space is sampled from that distribution
- third, the sampled point is decoded and the reconstruction error can be computed
- finally, the reconstruction error is backpropagated through the network

Variational autoencoders (VAEs) are models that address a specific problem with standard autoencoders. When you train an autoencoder, it learns to represent the input just in a compressed form called the latent space or the bottleneck. However, this latent space formed after training is not necessarily continuous and, in effect, might not be easy to interpolate. Variational autoencoders deal with this specific topic and express their latent attributes as a probability distribution, forming a continuous latent space that can be easily sampled and interpolated.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

9. Generative Adversarial Networks

What is a Generative Model?

What does "generative" mean in the name "Generative Adversarial Network"? "Generative" describes a class of statistical models that contrasts with discriminative models.
Informally:

- Generative models can generate new data instances.
- Discriminative models discriminate between different kinds of data instances.

A generative model could generate new photos of animals that look like real animals, while a discriminative model could tell a dog from a cat. GANs are just one kind of generative model.

More formally, given a set of data instances X and a set of labels Y:

- Generative models capture the joint probability p(X, Y), or just p(X) if there are no labels.
- Discriminative models capture the conditional probability p(Y | X).

A generative model includes the distribution of the data itself, and tells you how likely a given example is. For example, models that predict the next word in a sequence are typically generative models (usually much simpler than GANs) because they can assign a probability to a sequence of words. A discriminative model ignores the question of whether a given instance is likely, and just tells you how likely a label is to apply to the instance. Note that this is a very general definition. There are many kinds of generative model. GANs are just one kind of generative model.

What are GANs?

Generative Adversarial Networks (GANs) were developed in 2014 by Ian Goodfellow and his teammates. GAN is basically an approach to generative modeling that generates a new set of data based on training data that look like training data. GANs have two main blocks (two neural networks) which compete with each other and are able to capture, copy, and analyse the variations in a dataset. The two models are usually called Generator and Discriminator. To understand the term GAN let's break it into separate three parts

- Generative – To learn a generative model, which describes how data is generated in terms of a probabilistic model. In simple words, it explains how data is generated visually.
- Adversarial – The training of the model is done in an adversarial setting.
- Networks – use deep neural networks for training purposes.

Why GANs was Developed?

Machine learning algorithms and neural networks can easily be fooled to misclassify things by adding some amount of noise to data. After adding some amount of noise, the chances of misclassifying the images increase. Hence the small rise that, is it possible to implement something that neural networks can start visualizing new patterns like sample train data. Thus, GANs were built that generate new fake results similar to the original.

Components of GAN

A generative adversarial network (GAN) has two parts:

- The generator learns to generate plausible data. The generated instances become negative training examples for the discriminator.
- The discriminator learns to distinguish the generator's fake data from real data. The discriminator penalizes the generator for producing implausible results.

When training begins, the generator produces obviously fake data, and the discriminator quickly learns to tell that it's fake:

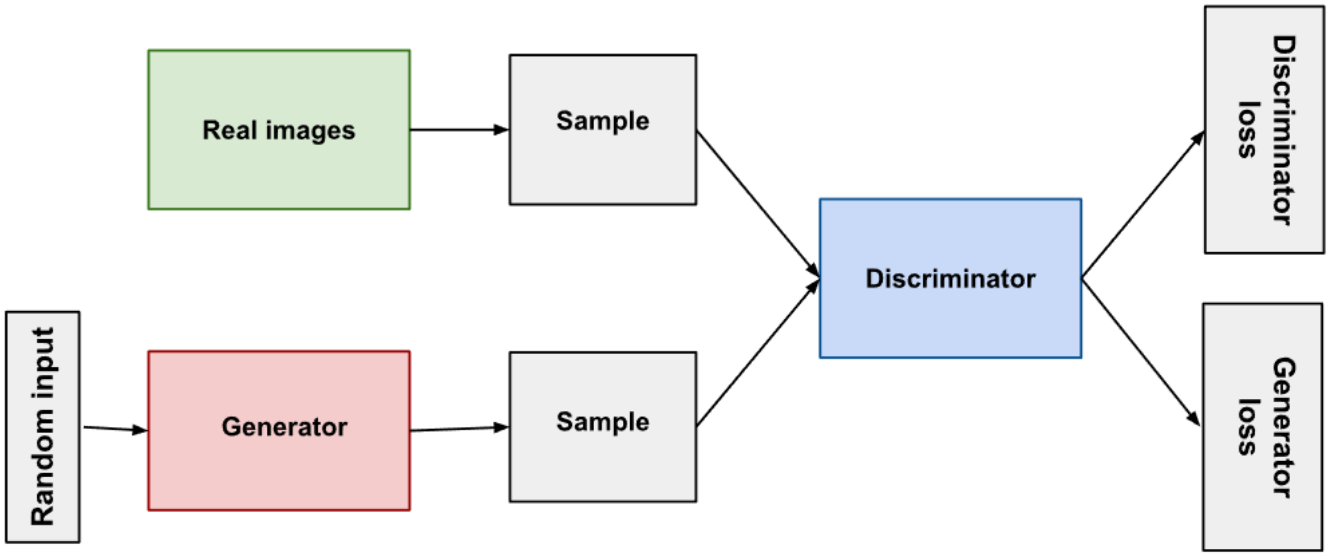
This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

As training progresses, the generator gets closer to producing output that can fool the discriminator:

Finally, if generator training goes well, the discriminator gets worse at telling the difference between real and fake. It starts to classify fake data as real, and its accuracy decreases.

Here's a picture of the whole system:



Both the generator and the discriminator are neural networks. The generator output is connected directly to the discriminator input. Through backpropagation, the discriminator's classification provides a signal that the generator uses to update its weights.

Steps for Training GAN

1. Define the problem
2. Choose the architecture of GAN
3. Train discriminator on real data
4. Generate fake inputs for the generator
5. Train discriminator on fake data
6. Train generator with the output of the discriminator

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

Different Types of GAN Models

1. Vanilla GAN
2. Conditional GAN (CGAN)
3. Deep Convolutional GAN (DCGAN)
4. Laplacian Pyramid GAN (LAPGAN)
5. Super Resolution GAN (SRGAN)

Advantages of GANs

- ✓ *Synthetic data generation*: GANs can generate new, synthetic data that resembles some known data distribution, which can be useful for data augmentation, anomaly detection, or creative applications.
- ✓ *High-quality results*: GANs can produce high-quality, photorealistic results in image synthesis, video synthesis, music synthesis, and other tasks.
- ✓ *Unsupervised learning*: GANs can be trained without labelled data, making them suitable for unsupervised learning tasks, where labelled data is scarce or difficult to obtain.
- ✓ *Versatility*: GANs can be applied to a wide range of tasks, including image synthesis, text-to-image synthesis, image-to-image translation, anomaly detection, data augmentation, and others.

Disadvantages of GANs

- *Training Instability*: GANs can be difficult to train, with the risk of instability, mode collapse, or failure to converge.
- *Computational Cost*: GANs can require a lot of computational resources and can be slow to train, especially for high-resolution images or large datasets.
- *Overfitting*: GANs can overfit the training data, producing synthetic data that is too similar to the training data and lacking diversity.
- *Bias and Fairness*: GANs can reflect the biases and unfairness present in the training data, leading to discriminatory or biased synthetic data.
- *Interpretability and Accountability*: GANs can be opaque and difficult to interpret or explain, making it challenging to ensure accountability, transparency, or fairness in their applications.

10. Autoencoders for Feature Extraction

An autoencoder is a neural network model that seeks to learn a compressed representation of an input. They are an unsupervised learning method, although technically, they are trained using supervised learning methods, referred to as self-supervised. They are typically trained as part of a broader model that attempts to recreate the input.

For example:

```
X = model.predict(X)
```

The design of the autoencoder model purposefully makes this challenging by restricting the architecture to a bottleneck at the midpoint of the model, from which the reconstruction of the input data is performed.

There are many types of autoencoders, and their use varies, but perhaps the more common use is as a learned or automatic feature extraction model.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

In this case, once the model is fit, the reconstruction aspect of the model can be discarded and the model up to the point of the bottleneck can be used. The output of the model at the bottleneck is a fixed length vector that provides a compressed representation of the input data.

Input data from the domain can then be provided to the model and the output of the model at the bottleneck can be used as a feature vector in a supervised learning model, for visualization, or more generally for dimensionality reduction.

Next, let's explore how we might develop an autoencoder for feature extraction on a regression predictive modeling problem.

Autoencoder for Regression

Developing an autoencoder to learn a compressed representation of the input features for a regression predictive modeling problem.

First, let's define a regression predictive modeling problem.

We will use the make_regression () scikit-learn function to define a synthetic regression task with 100 input features (columns) and 1,000 examples (rows). Importantly, we will define the problem in such a way that most of the input variables are redundant (90 of the 100 or 90 percent), allowing the autoencoder later to learn a useful compressed representation.

Next, we will develop a Multilayer Perceptron (MLP) autoencoder model.

The model will take all of the input columns, then output the same values. It will learn to recreate the input pattern exactly. The autoencoder consists of two parts: the encoder and the decoder. The encoder learns how to interpret the input and compress it to an internal representation defined by the bottleneck layer. The decoder takes the output of the encoder (the bottleneck layer) and attempts to recreate the input.

Once the autoencoder is trained, the decode is discarded and we only keep the encoder and use it to compress examples of input to vectors output by the bottleneck layer. Prior to defining and fitting the model, we will split the data into train and test sets and scale the input data by normalizing the values to the range 0-1, a good practice with MLPs. We will define the encoder to have one hidden layer with the same number of nodes as there are in the input data with batch normalization and ReLU activation.

This is followed by a bottleneck layer with the same number of nodes as columns in the input data, e.g., no compression. The decoder will be defined with the same structure. It will have one hidden layer with batch normalization and ReLU activation. The output layer will have the same number of nodes as there are columns in the input data and will use a linear activation function to output numeric values. The model will be fit using the efficient Adam version of stochastic gradient descent and minimizes the mean squared error, given that reconstruction is a type of multi-output regression problem. We can plot the layers in the autoencoder model to get a feeling for how the data flows through the model.

Next, we can train the model to reproduce the input and keep track of the performance of the model on the holdout test set. After training, we can plot the learning curves for the train and test sets to confirm the model learned the reconstruction problem well. Finally, we can save the encoder model for use later, if desired.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

11. Autoencoders for Classification

Developing an autoencoder to learn a compressed representation of the input features for a classification predictive modeling problem. First, let's define a classification predictive modeling problem. We will use the make_classification () scikit-learn function to define a synthetic binary (2-class) classification task with 100 input features (columns) and 1,000 examples (rows). Importantly, we will define the problem in such a way that most of the input variables are redundant (90 of the 100 or 90 percent), allowing the autoencoder later to learn a useful compressed representation.

Next, we will develop a Multilayer Perceptron (MLP) autoencoder model. The model will take all of the input columns, then output the same values. It will learn to recreate the input pattern exactly.

The autoencoder consists of two parts: the encoder and the decoder. The encoder learns how to interpret the input and compress it to an internal representation defined by the bottleneck layer. The decoder takes the output of the encoder (the bottleneck layer) and attempts to recreate the input.

Once the autoencoder is trained, the decoder is discarded and we only keep the encoder and use it to compress examples of input to vectors output by the bottleneck layer.

In this first autoencoder, we won't compress the input at all and will use a bottleneck layer the same size as the input. This should be an easy problem that the model will learn nearly perfectly and is intended to confirm our model is implemented correctly. Prior to defining and fitting the model, we will split the data into train and test sets and scale the input data by normalizing the values to the range 0-1, a good practice with MLPs. We will define the encoder to have two hidden layers, the first with two times the number of inputs (e.g., 200) and the second with the same number of inputs (100), followed by the bottleneck layer with the same number of inputs as the dataset (100).

To ensure the model learns well, we will use batch normalization and leaky ReLU activation. The decoder will be defined with a similar structure, although in reverse. It will have two hidden layers, the first with the number of inputs in the dataset (e.g., 100) and the second with double the number of inputs (e.g., 200). The output layer will have the same number of nodes as there are columns in the input data and will use a linear activation function to output numeric values.

The model will be fit using the efficient Adam version of stochastic gradient descent and minimizes the mean squared error, given that reconstruction is a type of multi-output regression problem. We can plot the layers in the autoencoder model to get a feeling for how the data flows through the model. Next, we can train the model to reproduce the input and keep track of the performance of the model on the hold-out test set. After training, we can plot the learning curves for the train and test sets to confirm the model learned the reconstruction problem well. Finally, we can save the encoder model for use later, if desired. As part of saving the encoder, we will also plot the encoder model to get a feeling for the shape of the output of the bottleneck layer, e.g., a 100-element vector.

Tying this all together, the complete example of an autoencoder for reconstructing the input data for a classification dataset without any compression in the bottleneck layer.

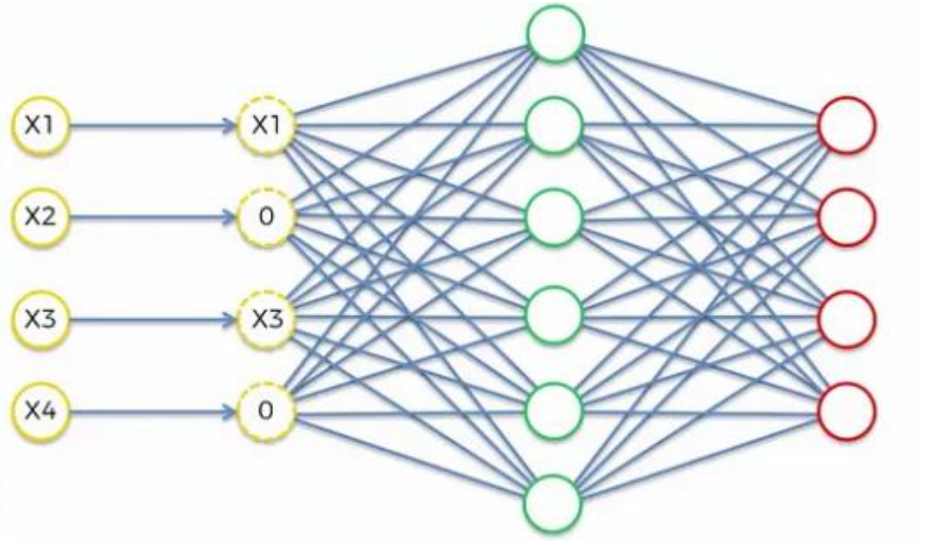
This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

12. Denoising Autoencoders

Denoising autoencoders are a stochastic version of standard autoencoders that reduces the risk of learning the identity function. Autoencoders are a class of neural networks used for feature selection and extraction, also called dimensionality reduction. In general, the more hidden layers in an autoencoder, the more refined this dimensional reduction can be. However, if an autoencoder has more hidden layers than inputs there is a risk the algorithm only learns the identity function during training, the point where the output simply equals the input, and then becomes useless.

Denoising autoencoders attempt to get around this risk of identity-function affiliation by introducing noise, i.e., randomly corrupting input so that the autoencoder must then "denoise" or reconstruct the original input. Denoising autoencoders are similar to regular autoencoders in that they take an input and produce an output. However, they differ because they don't have the input image as their ground truth. Instead, they use a noisy version. It is because removing image noise is difficult when working with images. You'd have to do it manually. But with a denoising autoencoder, we feed the noisy idea into our network and let it map it into a lower-dimensional manifold where filtering out noise becomes much more manageable. The loss function usually used with these networks is L2 or L1 loss. Denoising Autoencoders are an important and crucial tool for feature selection and extraction



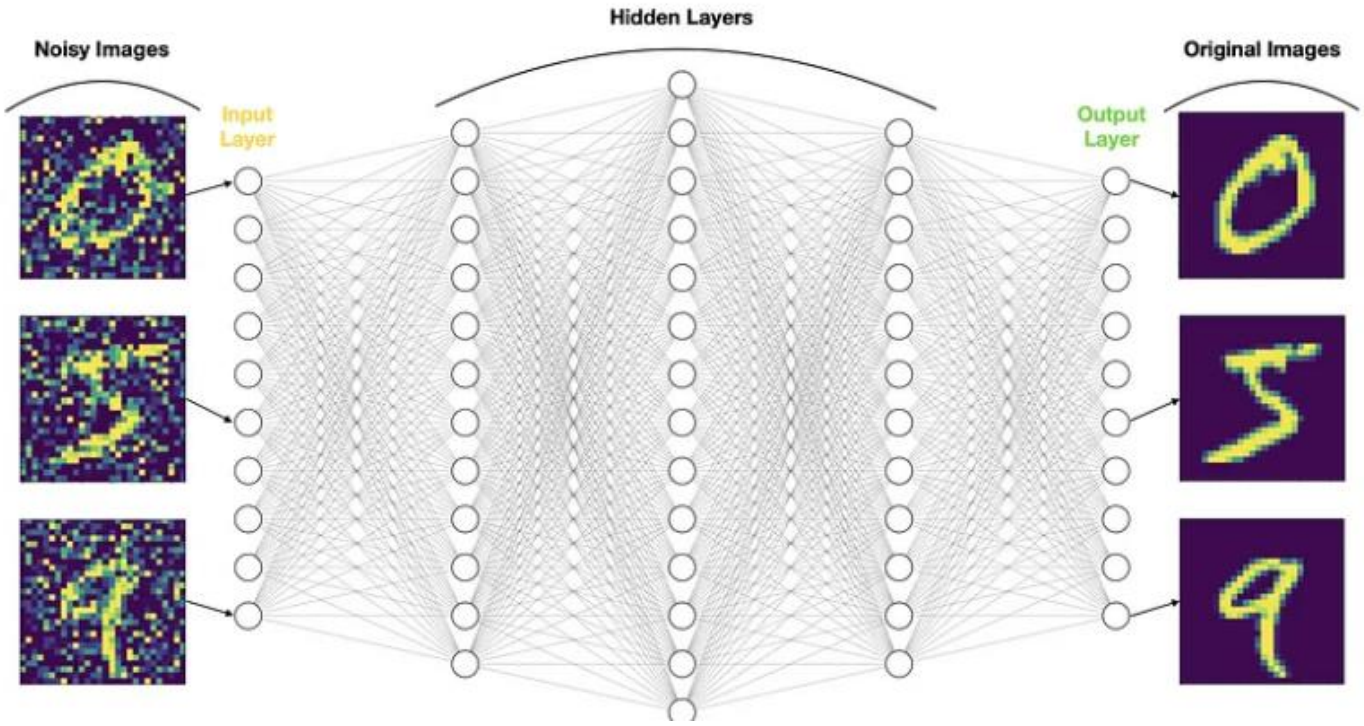
Denoising autoencoders add some noise to the input image and learn to remove it. Thus, avoiding to copy the input to the output without learning features about the data. These autoencoders take a partially corrupted input while training to recover the original undistorted input. The model learns a vector field for mapping the input data towards a lower-dimensional manifold which describes the natural data to cancel out the added noise. By this means, the encoder will extract the most important features and learn a more robust representation of the data.

The **purpose of a DAE** is to remove noise. You can also think of it as a customised denoising algorithm tuned to your data. Note the emphasis on the word customised. Given that we train a DAE on a specific set of data, it will be optimised to remove noise from similar data. For example, if we train it to remove noise from a collection of images, it will work well on similar images but will not be suitable for cleaning text data.

Unlike Undercomplete AE, we may use the same or higher number of neurons within the hidden layer, making the DAE overcomplete.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

The second difference comes from not using identical inputs and outputs. Instead, the outputs are the original data (e.g., images), while the inputs contain data with some added noise.



Above is an example of a DAE setup to denoise MNIST handwritten digits.

13. Sparse Autoencoders

A sparse autoencoder is one of a range of types of autoencoder artificial neural networks that work on the principle of unsupervised machine learning. Autoencoders are a type of deep network that can be used for dimensionality reduction – and to reconstruct a model through backpropagation.

Autoencoders seek to use items like feature selection and feature extraction to promote more efficient data coding. Autoencoders often use a technique called backpropagation to change weighted inputs, in order to achieve dimensionality reduction, which in a sense scale down the input for corresponding results. A sparse autoencoder is one that has small numbers of simultaneously active neural nodes.

Sparse autoencoders are controlled by changing the number of nodes at each hidden layer. Since it is impossible to design a neural network with a flexible number of nodes at its hidden layers, sparse autoencoders work by penalizing the activation of some neurons in hidden layers. It means that a penalty directly proportional to the number of neurons activated is applied to the loss function.

As a means of regularizing the neural network, the sparsity function prevents more neurons from being activated. There are two types of regularizer used:

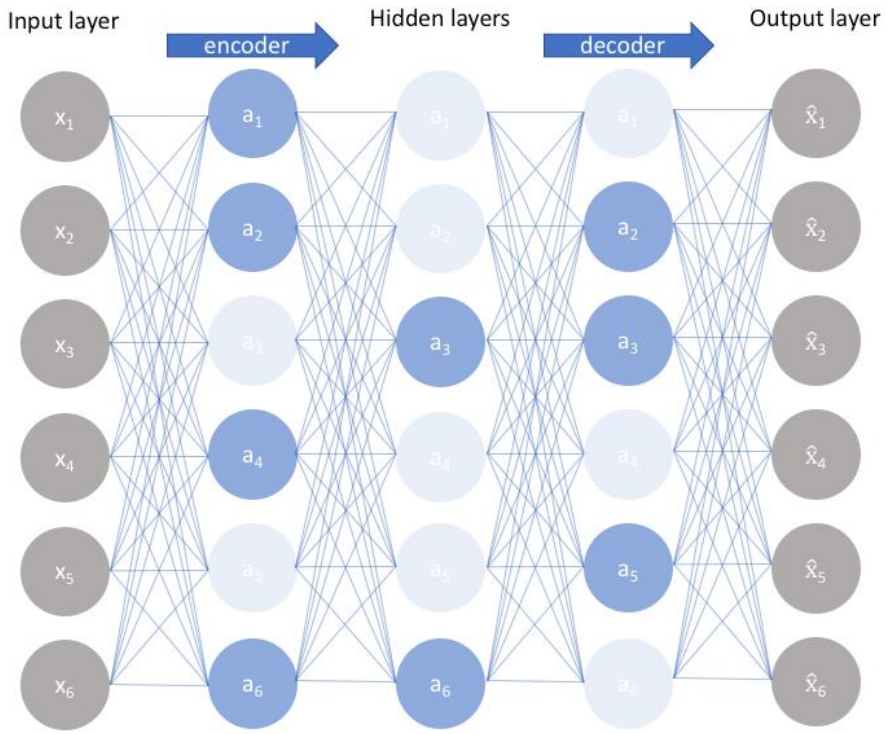
- The ***L1 Loss method*** is a general regularizer we can use to add magnitude to the model.
- The ***KL-divergence method*** considers the activations over a collection of samples at once rather than summing them as in the L1 Loss method. We constrain the average activation of each neuron over this collection.

This document is available free of charge on studocu

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)

A sparse autoencoder is simply an autoencoder whose training criterion involves a sparsity penalty. In most cases, we would construct our loss function by penalizing activations of hidden layers so that only a few nodes are encouraged to activate when a single sample is fed into the network.

The intuition behind this method is that, for example, if a man claims to be an expert in mathematics, computer science, psychology, and classical music, he might be just learning some quite shallow knowledge in these subjects. However, if he only claims to be devoted to mathematics, we would like to anticipate some useful insights from him. And it's the same for autoencoders we're training — fewer nodes activating while still keeping its performance would guarantee that the autoencoder is actually learning latent representations instead of redundant information in our input data.



There are actually two different ways to construct our sparsity penalty: L1 regularization and KL-divergence. Due to the sparsity of L1 regularization, sparse autoencoder actually learns better representations and its activations are more sparse which makes it perform better than original autoencoder without L1 regularization.

Downloaded by Amit Tiwari (amit.tiwari@tcetmumbai.in)